

To my lifelong companion Fátima, and to Oliver and Jakub

Pavel

To Nico and Leuntje van Rijn, for teaching me  
what is important in life

Jan

To my parents and also to Manuela, Quica, Manel, and Artur

Carlos

To Ada, Elias, Kobe, and Veerle, for reminding me  
how wonder-full the world is

Joaquin



---

## Preface

The first edition of this book was published in 2009, that is at the moment of writing already more than 10 years ago. As this area has progressed substantially, we decided to prepare the second edition. Our aim was to incorporate the most important advances, so that the new version would present an up-to-date account of the area and be useful to researchers, postgraduate students, and practitioners active in this area.

What are the major changes? First, if we just compare the number of chapters of the two editions, we note that it has doubled. So did roughly the number of pages.

We note that, at the time the first edition was written, the term AutoML was not yet around. So obviously we had to cover it in the new edition and also clarify its relationship to metalearning. Also, the automation of the design methods of chains of operations – nowadays referred to as pipelines or workflows – was in its infancy. So obviously we felt the need to update the existing material to keep up with this development.

In recent years the research areas of AutoML and metalearning have attracted a great deal of attention from not only researchers, but also many artificial intelligence companies, including, for instance, Google and IBM. The issue of how one can exploit metalearning to improve AutoML systems is one of the crucial questions that many researchers are trying to answer nowadays.

This book looks also into the future. As is usually the case, better understanding of some areas allows us to pose new research questions. We took care to include some in the respective chapters.

The authors of the first edition were Pavel Brazdil, Christophe Giraud-Carrier, Carlos Soares, and Ricardo Vilalta. With the extensive developments in the area, we wanted to strengthen the team by inviting Joaquin Vanschoren and Jan N. van Rijn to join the project. Unfortunately, in the end, Christophe and Ricardo were unavailable to work on the new edition. Nevertheless, the authors of the second edition are very grateful for their contributions at the onset of the project.

## How This Book Was Put Together

This book comprises two parts. Part I (chapters 2–7) discusses the basic concepts and architecture of metalearning and AutoML systems, while Part II (chapters 8–15) discusses various extensions. Part III (chapters 16–18) discusses metadata organization and management (e.g., metadata repositories) and ends with concluding remarks.

### Part I – Basic Concepts and Architecture

Chapter 1 starts by explaining the basic concepts used in this book, such as machine learning, metalearning, automatic machine learning, among others. It then continues with an overview of the basic architecture of a metalearning system and serves as an introduction to the rest of the book. All co-authors of the book collaborated on this chapter.

Chapter 2 focuses on ranking approaches that exploit metadata, as these are relatively easy to construct, but can still be very useful in practical applications. This chapter was written by P. Brazdil and J. N. van Rijn.<sup>1</sup> Chapter 3, written by P. Brazdil and J. N. van Rijn, is dedicated to the topic of evaluation of metalearning and AutoML systems. Chapter 4 discusses different dataset measures that play an important role as *metafeatures* in metalearning systems. This chapter was written by P. Brazdil and J. N. van Rijn. Chapter 5, written by P. Brazdil and J. N. van Rijn, can be seen as a continuation of Chapter 2. It discusses various metalearning approaches including, for instance, pairwise comparisons that were proposed in the past. Chapter 6 discusses hyperparameter optimization. It covers both basic search methods and also more advanced ones introduced in the area of automated machine learning (AutoML). This chapter was written by P. Brazdil, J. N. van Rijn, and J. Vanschoren. Chapter 7 discusses the problem of automating the construction of workflows or pipelines, representing sequences of operations. This chapter was written by P. Brazdil, but it reused some material from the first edition which was prepared by C. Giraud-Carrier.

### Part II – Advanced Techniques and Methods

Part 2 (chapters 8–15) continues with the topics in Part I, but covers different extensions of the basic methodology. Chapter 8, written by P. Brazdil and J. N. van Rijn, is dedicated to the topic of the design of configuration spaces and how to plan experiments. The two subsequent chapters discuss the specific topic of ensembles. Chapter 9, written by C. Giraud-Carrier, represents an invited chapter in this book. It describes different ways of organizing a set of base-level algorithms into ensembles. The authors of the second edition did not see any need to change this material, and so it is kept as it appeared in the first edition.

---

<sup>1</sup>Parts of Chapters 2 and 3 of the first edition, written by C. Soares and P. Brazdil, were reused and adapted for this chapter.

Chapter 10 continues with the topic of ensembles and shows how metalearning can be exploited in the construction of ensembles (ensemble learning). This chapter was written by C. Soares and P. Brazdil. The subsequent chapters are dedicated to rather specific topics. Chapter 11, written by J. N. van Rijn, describes how one can use metalearning to provide algorithm recommendations in data stream settings. Chapter 12, written by R. Vilalta and M. Meskhi, covers the transfer of meta-models and represents the second invited chapter of this book. It represents a substantial update of the similar chapter in the first edition, which was written by R. Vilalta. Chapter 13, written by M. Huisman, J. N. van Rijn, and A. Plaat, discusses metalearning in deep neural networks and represents the third invited chapter of this book. Chapter 14 is dedicated to the relatively new topic of automating Data Science. This chapter was drafted by P. Brazdil and incorporates various contributions and suggestions of his co-authors. The aim is to discuss various operations normally carried out within Data Science and to consider whether automation is possible and whether meta-knowledge can be exploited in this process. The aim of Chapter 15 is also to look into the future and consider whether it is possible to automate the design of more complex solutions. This chapter was written by P. Brazdil. These may involve not only pipelines of operations, but also more complex control structures (e.g., iteration), and automatic changes in the underlying representation.

### **Part III – Organizing and Exploiting Metadata**

Part III covers some practical issues and includes the final three chapters (16–18). Chapter 16, written by J. Vanschoren and J. N. van Rijn, discusses repositories of metadata, and in particular the repository known under the name *OpenML*. This repository includes machine-usable data on many machine learning experiments carried in the past and the corresponding results. Chapter 17, written by J. N. van Rijn and J. Vanschoren, shows how the metadata can be explored to obtain further insights in machine learning and metalearning research and thereby obtain new effective practical systems. Chapter 18 ends the book with brief concluding remarks about the role of metaknowledge and also presents some future challenges. The first version was elaborated by P. Brazdil, but includes various contributions of other co-authors, in particular of J. N. van Rijn and C. Soares.

### **Acknowledgements**

We wish to express our gratitude to all those who have helped in bringing this project to fruition.

We acknowledge the support of grant 612.001.206 from the Dutch Research Council (NWO) for granting the ‘Open Access Books’ funding, making the book publicly available.

P. Brazdil is grateful to the University of Porto, Faculty of Economics, R&D institution INESC TEC, and one of its centres, namely the *Laboratory of Artificial*

*Intelligence and Decision Support (LLAAD)*, for their continued support. The work carried out was partly supported by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project UIDB/50014/2020.

J. N. van Rijn is grateful to the University of Freiburg, the Data Science Institute (DSI) at Columbia University, and Leiden Institute of Advanced Computer Science (LIACS) at Leiden University, for their support throughout this project.

C. Soares expresses his gratitude to the University of Porto and to the Faculty of Engineering for their support.

J. Vanschoren is grateful to the Eindhoven University of Technology and to the Data Mining Group for their support.

We are greatly indebted to many of our colleagues for many useful discussions and suggestions that are reflected in this book: Salisu M. Abdulrahman, Bernd Bischl, Hendrik Blockeel, Isabelle Guyon, Holger Hoos, Geoffrey Holmes, Frank Hutter, João Gama, Rui Leite, Andreas Mueller, Bernhard Pfahringer, Ashwin Srinivasan, and Martin Wistuba.

We also acknowledge the influence of many other researchers resulting from various encounters and discussions, in person or by email: Herke van Hoof, Peter Flach, Pavel Kordík, Tom Mitchell, Katharina Morik, Sergey Muravyov, Aske Plaat, Ricardo Prudêncio, Luc De Raedt, Michele Sebag, and Kate Smith-Miles.

We wish to thank also many others with whom we have collaborated: Pedro Abreu, Mitra Baratchi, Bilge Celik, Vítor Cerqueira, André Correia, Afonso Costa, Tiago Cunha, Katharina Eggenberger, Matthias Feurer, Pieter Gijsbers, Carlos Gomes, Taciana Gomes, Hendrik Jan Hoogeboom, Mike Huisman, Matthias König, Lars Kotthoff, Jorge Kanda, Aaron Klein, Walter Kusters, Marius Lindauer, Marta Mercier, Péricles Miranda, Felix Mohr, Mohammad Nozari, Sílvia Nunes, Catarina Oliveira, Marcos L. de Paula Bueno, Florian Pfisterer, Fábio Pinto, Peter van der Putten, Sahi Ravi, Adriano Rivolli, André Rossi, Cláudio Sá, Prabhant Singh, Arthur Sousa, Bruno Souza, Frank W. Takes, and Jonathan K. Vis. We are grateful also to the OpenML community, for their efforts to make machine learning research reproducible.

We are also grateful to our editor, Ronan Nugent from Springer, for his patience and encouragement throughout this project.

We wish to express our thanks to Manuel Caramelo for his careful proof-reading of the draft of the whole book and for suggesting many corrections.

Porto, Eindhoven, Leiden

March 2021

*Pavel Brazdil*

*Jan N. van Rijn*

*Carlos Soares*

*Joaquin Vanschoren*

---

# Contents

---

## Part I Basic Concepts and Architecture

---

<b>1 Introduction</b>	3
.....	
<b>2 Metalearning Approaches for Algorithm Selection I (Exploiting Rankings)</b>	19
.....	
<b>3 Evaluating Recommendations of Metalearning/AutoML Systems</b>	39
.....	
<b>4 Dataset Characteristics (Metafeatures)</b>	53
.....	
<b>5 Metalearning Approaches for Algorithm Selection II</b>	77
.....	
<b>6 Metalearning for Hyperparameter Optimization</b>	103
.....	
<b>7 Automating Workflow/Pipeline Design</b>	123
.....	

---

## Part II Advanced Techniques and Methods

---

<b>8 Setting Up Configuration Spaces and Experiments</b>	143
.....	
<b>9 Combining Base-Learners into Ensembles</b>	169
<i>Christophe Giraud-Carrier</i> .....	

<b>10 Metalearning in Ensemble Methods</b>	189
<b>11 Algorithm Recommendation for Data Streams</b>	201
<b>12 Transfer of Knowledge Across Tasks</b> <i>Ricardo Vilalta and Mikhail M. Meskhi</i>	219
<b>13 Metalearning for Deep Neural Networks</b> <i>Mike Huisman, Jan N. van Rijn, and Aske Plaat</i>	237
<b>14 Automating Data Science</b>	269
<b>15 Automating the Design of Complex Systems</b>	283
<hr/>	
<b>Part III Organizing and Exploiting Metadata</b>	
<hr/>	
<b>16 Metadata Repositories</b>	297
<b>17 Learning from Metadata in Repositories</b>	311
<b>18 Concluding Remarks</b>	329
<b>Index</b>	339



**Basic Concepts and Architecture**



# Introduction

**Summary.** This chapter starts by describing the organization of the book, which consists of three parts. Part I discusses some basic concepts, including, for instance, what metalearning is and how it is related to automatic machine learning (AutoML). This continues with a presentation of the basic architecture of metalearning/AutoML systems, discussion of systems that exploit algorithm selection using prior metadata, methodology used in their evaluation, and different types of meta-level models, while mentioning the respective chapters where more details can be found. This part also includes discussion of methods used for hyperparameter optimization and workflow design. Part II includes the discussion of more advanced techniques and methods. The first chapter discusses the problem of setting up configuration spaces and conducting experiments. Subsequent chapters discuss different types of ensembles, metalearning in ensemble methods, algorithms used for data streams and transfer of meta-models across tasks. One chapter is dedicated to metalearning for deep neural networks. The last two chapters discuss the problem of automating various data science tasks and trying to design systems that are more complex. Part III is relatively short. It discusses repositories of metadata (including experimental results) and exemplifies what can be learned from this metadata by giving illustrative examples. The final chapter presents concluding remarks.

## 1.1 Organization of the Book

This book comprises three parts. In Part I (Chaps. 2–7) we sketch the basic concepts and architecture of metalearning systems, especially focusing on which types of “metaknowledge” can be collected by observing the performance of different models on prior tasks and how this can be used within metalearning approaches to learn new tasks more efficiently. Since this type of metalearning is closely related to automated machine learning (AutoML), we also cover this topic in some depth, but with a specific focus on how we can improve AutoML through metalearning.

Part II (Chaps. 8–15) covers different extensions of these basic ideas to more specific tasks. First, we discuss some methods that can be used in the design of configuration spaces that affect the search of metalearning and AutoML systems. Then, we show how metalearning can be used to build better ensembles and to recommend algorithms for streaming data. Next, we discuss how to transfer information from previously learned models to new tasks, using transfer learning and few-shot learning in neural networks.

The final two chapters are dedicated to the problem of automating data science and the design of complex systems.

Part III (Chaps. 16–18) provides practical advice on how to organize metadata in repositories and how this can be leveraged in machine learning research. The last chapter includes our closing remarks on and presents future challenges.

## 1.2 Basic Concepts and Architecture (Part I)

### 1.2.1 Basic Concepts

#### Role of machine learning

We are surrounded by data. On a daily basis, we are confronted by many forms of it. Companies try to market their products with commercials in the form of billboards and online advertisements. Large sensor networks and telescopes measure complex processes, happening around us on Earth or throughout the universe. Pharmaceutical institutions record the interactions between types of molecules, in search of new medications for new diseases.

All this data is valuable, as it enables us to characterize different situations, learn to separate them into different groups, and incorporate this into a system that can help us make decisions. We can thus identify fraudulent transactions from financial data, develop new medical drugs based on clinical data, or speculate about the evolution of celestial bodies in our universe. This process involves *learning*.

The scientific community has elaborated many techniques for analyzing and processing data. A traditional scientific task is modeling, where the aim is to describe the given complex phenomenon in a simplified way, in order to learn something from it. Many data modeling techniques have been developed for that purpose based on various intuitions and assumptions. This area of research is called *Machine Learning*.

#### Role of metalearning

As was shown, we cannot assume that there is one algorithm that works for all sorts of data, as each algorithm has its own area of expertise. Selecting the proper algorithm for a given task and dataset is key to obtaining an adequate model. This, in itself, can be seen as a learning task.

This process of learning across tasks has generally been called *metalearning*. Over the past decades, however, various machine learning researchers have used this term in many different ways, covering concepts such as meta-modeling, learning to learn, continuous learning, ensemble learning, and transfer learning. This large and growing body of work has clearly demonstrated that metalearning can make machine learning drastically more efficient, easier, and more trustworthy.

The area of metalearning is a very active research field. Many new and interesting research directions are emerging that address this general goal in novel ways. This book is meant to provide a snapshot of the most established research to date. As the area has grown a lot, the material had to be organized and structured into cohesive units. For instance, dataset characteristics are discussed in a separate chapter (Chapter 4), even though they play an important role in many other chapters.

## Definition of metalearning

Let us start with a definition of metalearning, as it is viewed in this book:

Metalearning is the study of principled methods that exploit metaknowledge to obtain efficient models and solutions by adapting machine learning processes.

The *metaknowledge* referred to above typically includes any sort of information obtained from previous tasks, such as descriptions of prior tasks, the pipelines and neural architectures tried, or the resulting models themselves. In many cases, it also includes knowledge that is obtained *during* the search for the best model on a new task, and that can be leveraged to guide the search for better learning models. Lemke et al. (2015) describe this from a systems point of view:

A metalearning system must include a learning subsystem, which adapts with experience. Experience is gained by exploiting meta-knowledge extracted: a) in a previous learning episode on a single dataset and/or b) from different domains or problems.

Currently, the aim of many is to exploit the metadata gathered both on the past and the target dataset.

## Metalearning versus automated machine learning (AutoML)

One important question is: what is the difference between a metalearning system and an AutoML system? Although this is a rather subjective matter for which different answers may be given, here we present the definition of AutoML of Guyon et al. (2015):

AutoML refers to all aspects of automating the machine learning process, beyond model selection, hyperparameter optimization, and model search, . . .

Many AutoML systems make use of experience obtained from previously seen datasets. As such, many AutoML systems are, according to the definition above, also metalearning systems. In this book, we focus on techniques that involve metalearning, as well as AutoML systems that often use metalearning.

## On the origins of the term *metalearning*

The pioneering work of Rice (1976) is discussed in Chapter 1. This work was not widely known in the machine learning community until much later. In the 1980s Larry Rendell published various articles on *bias management* (this topic is discussed in Chapter 8). One of his articles (Rendell et al., 1987) includes the following text:

The VBMS [Variable Bias Management System] can perform **meta-learning**. Unlike most other learning systems, the VBMS learns at different levels. In the process of learning a concept the system will also acquire knowledge about induction problems, biases, and the relationship between the two. Thus the system will not only learn concepts, but will also learn about the relationship between problems and problem-solving techniques.

P. Brazdil encountered the term *meta-interpreter* in connection with the work of Kowalski (1979) at the University of Edinburgh in the late 70's. In 1988 he organized a workshop on *Machine Learning, Meta-Reasoning and Logics* (Brazdil and Konolige, 1990). The introduction of this book includes the following passage:

For some meta-knowledge represents knowledge that talks about other (object level) knowledge. The purpose of meta-knowledge is mainly to control inference. In the second school of thought, **meta-knowledge** has a somewhat different role: it is used to control the process of knowledge acquisition and knowledge reformulation (learning).

Metalearning was explored in project StatLog (1990–93) (Michie et al., 1994).

### 1.2.2 Problem types

The scientific literature typically distinguishes the following problem types, many of which will be referred to throughout the book. The general aim of metalearning systems is to learn from the usage of prior models (how they were constructed and how well they performed) in order to *to model a target dataset better*. If the base-level task is classification, this implies that the system can predict the value of the target variable, i.e. the class value in this case. Ideally, it does this better (or more efficiently) by leveraging information besides the training data itself.

**Algorithm selection (AS):** Given a set of algorithms and a dataset (target dataset), determine which algorithm is most appropriate to model the target dataset.

**Hyperparameter optimization (HPO):** Given an algorithm with specific *hyperparameters* and a target dataset, determine the best hyperparameter settings of the given algorithm to model the target dataset.

**Combined algorithm selection and hyperparameter optimization (CASH):** Given a set of algorithms, each with its own set of hyperparameters, and a target dataset, determine which algorithm to use and how to set its hyperparameters to model the target dataset. Some CASH systems address also the more complex pipeline synthesis task discussed next.

**Workflow (pipeline) synthesis:** Given a set of algorithms, each with its own set of hyperparameters, and a target dataset, design a workflow (pipeline) consisting of a one or more algorithms to model the target dataset. The inclusion of a particular algorithm and its hyperparameter settings in the workflow can be seen as a CASH problem.

**Architecture search and/or synthesis:** This problem type can be seen as a generalization of the problem type above. In this setting the individual constituents do not need to be organized in a sequence, as it is done in workflows (pipelines). The architecture can include, for instance, partially ordered or tree-like structures. The neural network architecture design can be seen as a problem that falls into this category.

**Few-shot learning:** Given a target dataset with few examples and various datasets that are very similar, but include many examples, retrieve a model that has been pre-trained on prior datasets and fine-tune it to perform well on the target dataset.

We note that algorithm selection problems are defined on a *discrete* set of algorithms, while the hyperparameter optimization problem and CASH problem are typically defined on *continuous* configuration spaces, or heterogeneous spaces with both discrete

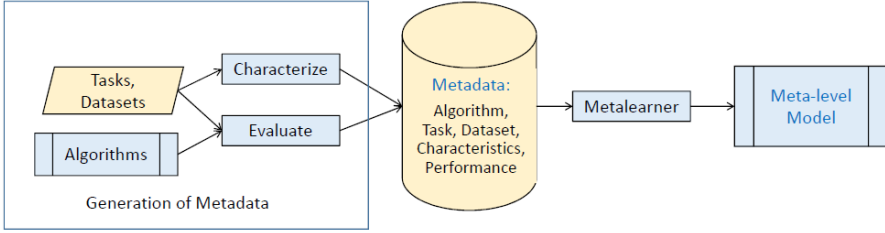


Fig. 1.1: Generating a meta-level model

and continuous variables. Techniques for algorithm selection can also easily be applied to discretized versions of the latter.

In this book, we follow the following convention that has been broadly adopted by the machine learning community. A *hyperparameter* is a user-defined parameter that determines the behavior of a particular machine learning algorithm; for instance, the level of pruning in a decision tree, or the learning rate in neural networks are hyperparameters. A (model) *parameter* is a parameter that has been learned based on the training data. For instance, the weights of a neural network model are regarded as model parameters.

### 1.2.3 Basic architecture of metalearning and AutoML systems

The algorithm selection problem was first formulated by Rice (1976). He has observed that it is possible to relate the performance of algorithms to *dataset characteristics/features*. In other words, dataset features are often quite good predictors of performance of algorithms. This can be exploited to identify the best performing algorithm for a given target dataset. It has since then been applied to many application domains, also beyond machine learning (Smith-Miles, 2008).

A general architecture for metalearning systems that address the algorithm selection problem is shown in Figure 1.1. First, *metadata* is collected that encodes information on previous learning episodes. This includes descriptions of the *tasks* that we solved before. For instance, these could be classification tasks where we build classifiers for a given dataset, or reinforcement learning tasks defined by different learning environments. *Characterizations* of these tasks are often very useful for reasoning how new tasks could be related to prior tasks. The metadata also includes *algorithms* (e.g. machine learning pipelines or neural architectures) that were previously used for learning these tasks, and information about performance resulting from *evaluations*, showing how well that worked. In some cases, we can store the trained models as well, or measurable properties of these models. Such metadata from many previous (or current) tasks could be combined in a database, or a “memory” of prior experience that we want to build on.

We can leverage this metadata in many different ways. For instance, we could use metadata directly inside metalearning algorithms, or use it to train a meta-level model. Such a meta-model can be used inside a metalearning system, as illustrated in Figure 1.2. Based on the characteristics of a novel “target” task, the meta-model could construct or recommend new algorithms to try, and use the observed performance to update the

algorithms or recommendations until some stopping condition, usually a time budget or performance constraint, has been satisfied. In some cases, no task characteristics are included, and the metalearning system learns from prior experience and observations on the new task alone.

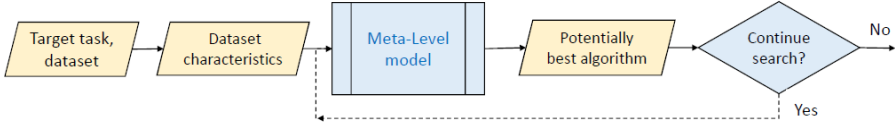


Fig. 1.2: Using a meta-level model to predict the best algorithm

### 1.2.4 Algorithm selection using metadata from prior datasets (Chaps. 2,5)

Chapters 2 and 5 discuss methods that exploit performance metadata of algorithms on previous tasks to recommend algorithms for a target dataset. These recommendations can take the form of rankings of candidate algorithms (Chap. 2) or meta-models that predict the suitability of algorithms on new tasks (Chap. 5).

In Chapter 2 we describe a relatively simple approach, the *average ranking method*, which is often used as a baseline method. The average ranking algorithm uses meta-knowledge acquired in previous tasks to identify the potentially best base-level algorithms for the current task.

This approach requires that an appropriate evaluation measure, such as *accuracy*, is set beforehand. In this chapter we also describe a method that builds this ranking based on a combination of accuracy and runtime, yielding good anytime performance.

Chapter 5 discusses more advanced methods. The methods in both chapters are designed to work on discrete problems.

### 1.2.5 Evaluation and comparisons of different systems (Chap. 3)

When working with a given metalearning system, it is important to know whether we can trust its recommendations and how its performance compares with other competing approaches. Chapter 3 discusses a typical approach that is commonly used to evaluate metalearning systems and make comparisons.

In order to obtain a good estimate of the performance of a metalearning system, it needs to be evaluated on many datasets. As the performance of algorithms may vary substantially across these datasets, many systems normalize the performance values first to make comparisons meaningful. Section 3.1 discusses some of the most common normalization methods used in practice.

Assuming that a metalearning system outputs a sequence of algorithms to test, we can study how similar this sequence is from the ideal sequence. This can be determined by looking at a degree of correlation between the two sequences. Section 3.2 describes this in more detail.



We note that the above approach compares the predictions made by meta-level model with the meta-target (i.e., correct ordering of algorithms). A disadvantage of this approach is that it does not show directly the effects in terms of base-level performance. This problem can be avoided by considering the appropriate base-level performance of different metalearning systems and how this evolves with time. If the ideal performance is known, it is possible to calculate the value of *performance loss*, which is the difference between the actual performance and the ideal value. The loss curve shows how the loss evolves with time. In some systems the maximum available time (i.e., time budget) is given beforehand. Different systems and their variants can then be compared by looking at how the loss evolves with time. More details can be found in Section 3.3.

Section 3.4 also presents some useful measures, such as *loose accuracy* and *discounted cumulative gain* that are often used when comparing sequences. The final section (Section 3.5) describes the methodology that is commonly used in comparisons involving several metalearning/AutoML systems.

### 1.2.6 Role of dataset characteristics/metafeatures (Chap. 4)

We note that in the proposal of Rice (1976), dataset characteristics play a crucial role. They have been used in many metalearning systems since then. Typically, they help to restrict the search for the potentially best algorithm. If the characteristics are not available, or if they are difficult to define in a given domain, the search can proceed nevertheless. The basic approaches based on rankings or pairwise comparisons discussed in Chapters 2 and 5 can be used without any dataset characteristics. This is an advantage, as indeed, in many domains, it is difficult to come up with a sufficient number of informative characteristics that enable to discriminate a large number of very similar algorithm configurations.

One important group of characteristics is the group of *performance-based characteristics*. This group includes, for instance, *sampling landmarks*, representing the performance of particular algorithms on samples of data. These can be obtained in virtually all domains.

There is no doubt that some characteristics are, in general, important. Consider, for instance, the basic characteristic of the target variable. If it is numeric, it suggests that a suitable regression algorithm should be used, while if it is categorical, a classification algorithm should be used. A similar argument can be given when we encounter balanced/unbalanced data. This characteristic conditions the choice of the right approach.

Chapter 4 discusses various dataset characteristics, organized by task type, such as classification, regression, or time series. Various chapters in this book discuss how the datasets characteristics are effectively used in different metalearning approaches (e.g., Chaps. 2, 5).

### 1.2.7 Different types of meta-level models (Chap. 5)

Several types of meta-level models were used in the past:

- regression model,
- classification model,
- relative performance model.

Chapter 5 discusses the details of such models. These are used in various approaches discussed throughout the book.

The regression model uses a suitable regression algorithm which is trained on the metadata and can then be used to predict the performance of a given set of base-level algorithms. The predictions can be used to order the base-level algorithms and hence identify the best one.

Regression models play an important role also in the search for the best hyperparameter configuration, particularly if these are numeric and continuous. For example, the method called *sequential model-based optimization* discussed in Chapter 6 uses a regression algorithm on a meta-level to model the loss function, and to identify promising hyperparameter settings.

A classification model identifies which of the base-level algorithms are *applicable* to the target classification task. This implies that these algorithms are likely to obtain a relatively good performance on the target task. We note that this metalearning task is applied to a discrete domain.

If we were to use probabilistic classifiers at the meta-level which provide apart from the class (e.g., applicable or non-applicable), also numeric values related to the probability of classification, then these values can be used to identify the potentially best possible base-level algorithm or to explore a ranking in further search.

The relative performance model is based on an assumption that it is not necessary to have the details about the actual performance of algorithms, if the aim is to identify good-performing algorithms. All that is needed is the information regarding their relative performance. Relative performance models can either use rankings or pairwise comparisons. In all these settings it is possible to use search to identify the potentially best algorithm for the target dataset.

## 1.2.8 Hyperparameter optimization (Chap. 6)

Chapter 6 describes various approaches for the hyperparameter optimization and combined algorithm selection and hyperparameter optimization problems.

This chapter differs from the Chapters 2 and 5 in one important aspect: it discusses methods that use performance metadata obtained mainly on the target dataset. The metadata is used to construct relatively simple and fast-to-test models of the target algorithm configuration (algorithm with appropriate hyperparameter settings) that can be queried. The aim of these queries is to identify the best configuration to test, which is the one with the highest estimate of performance (e.g., accuracy). This type of search is referred to as *model-based search*.

The situation is not entirely clear-cut, though. As is shown, the metadata gathered on past datasets can also be useful and improve the performance of model-based search.

## 1.2.9 Automatic methods for workflow design (Chap. 7)

Many tasks require a solution which does not involve a single base-level algorithm, but rather several algorithms. The term *workflow* (or *pipeline*) is often used to represent such sequences. In general, the set may be only partially ordered.

When designing workflows (pipelines), the number of configurations can grow dramatically. This is because each item in the workflow can in principle be substituted by an appropriate base-level operation and there may be several such operations available. The problem is exacerbated by the fact that a sequence of two or more operators can in general be executed in any order, unless instructions are given to the contrary. This

creates a problem, as for  $N$  operators there are  $N!$  possible orderings. So, if a set of operators should be executed in a specific order, explicit instructions need to be given to that effect. If the order is irrelevant, the system should also be prevented from experimenting with alternative orderings. All alternative workflows and their configurations (including all possible hyperparameter settings) constitute the so-called *configuration space*.

Chapter 7 discusses various means that have been used to restrict the design options and thus reduce the size of the configuration space. These include, for instance, ontologies and context-free grammars. Each of these formalisms has its merits and shortcomings.

Many platforms have resorted to planning systems that use a set of operators. These can be designed to be in accordance with given ontologies or grammars. This topic is discussed in Section 7.3.

As the search space may be rather large, it is important to leverage prior experience. This topic is addressed in Section 7.4, which discusses *rankings of plans* that have proved to be useful in the past. So, the workflows/pipelines that have proved successful in the past, can be retrieved and used as plans for future tasks. So, it is possible to exploit both planning and metalearning.

## 1.3 Advanced Techniques and Methods (Part II)

### 1.3.1 Setting up configuration spaces and experiments (Chap. 8)

One of the challenges that metalearning and AutoML research faces nowadays is that the number of algorithms (workflows in general) and their configurations is so large that it can cause problems when searching through this space for an acceptable solution. Also, it is not possible any more to have a complete set of experimental results (complete metadata). So, several questions arise:

1. Is the configuration space adequate for the set of tasks of interest? This question is addressed in Section 8.3.
2. Which parts of the configuration space are relevant, and which parts are less relevant? This question is addressed in Section 8.4.
3. Can we reduce the configuration space to make metalearning more effective? This question is addressed in Section 8.5.

Considering this from the perspective of the algorithm selection framework, these questions are concerned with the algorithm space.

In order to successfully learn, also several aspects from the problem space become important. We address the following questions:

1. Which datasets do we need to be able to transfer knowledge to new datasets? This question is addressed in Section 8.7.
2. Do we need complete metadata, or does incomplete metadata suffice? This question is already partly addressed in Chapter 2, and is further elaborated on in Section 8.8.
3. Which experiments need to be scheduled first to obtain adequate metadata? This question is addressed in Section 8.9.

### 1.3.2 Automatic methods for ensembles and streams

#### Combining base-learners into ensembles (Chap. 9)

Ensembles of classification or regression models represent an important area of machine learning. They have become popular because they tend to achieve high performance when compared with single models. This is why we devote one chapter to this topic in this book. We start by introducing ensemble learning and present an overview of some of its most well-known methods, including bagging, boosting, stacking, and cascade generalization, among others.

#### Metalearning in ensemble methods (Chap. 10)

There is a growing number of approaches integrating metalearning methods – in the sense used in this book – in ensemble learning approaches.<sup>1</sup> In Chapter 10 we discuss some of those approaches. We start by providing a general perspective and then we analyze them in detail, concerning the ensemble method used, the metalearning approach employed, and finally the metadata involved.

We show that ensemble learning presents many opportunities for research on metalearning, with very interesting challenges, namely in terms of the size of the configuration space, the definition of the competence regions of the models, and the dependency between them. Given the complexity of ensemble learning systems, one final challenge is to apply metalearning to understand and explain their behavior.

#### Algorithm recommendation for data streams (Chap. 11)

Real-time analysis of data streams is a key area of data mining research. Many real-world collected data is in fact a stream where observations come in one by one, and algorithms processing these are often subject to time and memory constraints. Examples of this are stock prices, human body measurements, and other sensor measurements. The nature of data can change over time, effectively outdated models that we have built in the past.

This has been identified by the scientific community, and hence many machine learning algorithms have been adapted or are specifically designed to work on data streams. Some examples of this are *Hoeffding trees*, *online boosting*, and *leveraging bagging*. Also, the scientific community provided the so-called *drift detectors*, mechanisms that identify when the created model is no longer applicable. Once again, we are faced with an algorithm selection problem, which can be solved with metalearning.

In this chapter, we discuss three approaches on how techniques from within the scope of this book have been used to address this problem. First, we discuss metalearning approaches that divide the streams into various intervals, calculate metafeatures over these parts, and use a meta-model for each part of the stream to select which classifier to use. Second, we discuss ensemble approaches, that use the performance on recent data to determine which ensemble members are still up to date. In some sense, these methods are much simpler to apply in practice, as they do not rely on a basis of metalearning, and consistently outperform the metalearning approaches. Third, we discuss approaches

---

<sup>1</sup>In ensemble learning literature the term *metalearning* is used to refer to certain ensemble learning approaches (Chan and Stolfo, 1993), where it has a somewhat different meaning from the one used in this book.

that are built upon the notion of recurring concepts. Indeed, it is reasonable to assume some sort of seasonality in data, and models that have become outdated can become relevant again at some point later in time. This section describes systems that facilitate this type of data. Finally, this chapter closes with open research questions and directions for future work.

### 1.3.3 Transfer of meta-models across tasks (Chap. 12)

Many people hold the view that learning should not be viewed as an isolated task that starts from scratch with every new problem. Instead, a learning algorithm should exhibit the ability to exploit the results of previous learning processes to new tasks. This area is often referred to as *transfer of knowledge across tasks*, or simply *transfer learning*. The term *learning to learn* is also sometimes used in this context.

Chapter 12 is dedicated to transfer learning, whose aim is to improve learning by detecting, extracting, and exploiting certain information across tasks. This chapter is an *invited chapter* written by Ricardo Vilalta and Mikhail M. Meskhi, with the aim of complementing the material of this book.

The authors discuss different approaches available to transfer knowledge across tasks, namely representational transfer and functional transfer. The term *representational transfer* is used to denote cases when the target and source models are trained at different times and the transfer takes place after one or more source models have already been trained. In this case there is an explicit form of knowledge transferred directly to the target model or to a meta-model that captures the relevant part of the knowledge obtained in past learning episodes.

The term *functional transfer* is used to denote cases where two or more models are trained simultaneously. This situation is sometimes referred to as *multi-task learning*. In this case the models share (possibly a part of) their internal structure during learning. More details on this can be found in Section 12.2.

The authors address the question regarding what exactly can be transferred across tasks and distinguish instance-, feature-, and parameter-based transfer learning (see Section 12.3). Parameter-based transfer learning describes a case when the parameters found on the source domain can be used to initialize the search on the target domain. We note that this kind of strategy is also discussed in Chapter 6 (Section 6.7).

As neural networks play an important role in AI, one whole section (Section 12.3) is dedicated to the issue of transfer in neural networks. So, for instance, one of the approaches involves transfer of a part of network structure. This section also describes a *double loop architecture*, where the base-learner iterates over the training set in an inner loop, while the metalearner iterates over different tasks to learn metaparameters in an outer loop. This section also describes transfer in kernel methods and in parametric Bayesian models. The final section (Section 12.4) describes a theoretical framework.

### 1.3.4 Metalearning for deep neural networks (Chap. 13)

Deep learning methods are attracting a lot of attention recently, because of their successes in various application domains, such as image or speech recognition. As training is generally slow and requires a lot of data, metalearning can offer a solution to this problem. Metalearning can help to identify the best settings for hyperparameters, as well as parameters, concerned, for instance, with weights of a neural network model.

Most metalearning techniques involve a learning process at two levels, as was already pointed out in the previous chapter. At the *inner level*, the system is presented with a new task and tries to learn the concepts associated with this task. This adaptation is facilitated by the knowledge that the agent has accumulated from other tasks, at the *outer level*.

The authors categorize this field of metalearning in three groups: metric-, model-, and optimization-based techniques, following previous work. After introducing notation and providing background information, this chapter describes key techniques of each category, and identifies the main challenges and open questions. An extended version of this overview is also available outside this book (Huisman et al., 2021).

### 1.3.5 Automating data science and design of complex systems

#### Automating data science (AutoDS) (Chap. 14)

It has been observed that in data science a greater part of the effort usually goes into various preparatory steps that precede model-building. The actual model-building step typically requires less effort. This has motivated researchers to examine how to automate the preparatory steps and gave rise to methodology that is known under the name *CRISP-DM model* (Shearer, 2000).

The main steps of this methodology include problem understanding and definition of the current problem/task, obtaining the data, data preprocessing and various transformations of data, model building, and its evaluation and automating report generation. Some of these steps can be encapsulated into a workflow, and so the aim is to design a workflow with the best potential performance.

The model-building step, including hyperparameter optimization, is discussed in Chapter 6. More complex models in the form of workflows are discussed in Chapter 7. The aim of Chapter 14 is to focus on the other steps that are not covered in these chapters.

The area related to the definition of the current problem (task) involves various steps. In Section 14.1 we argue that the problem understanding of a domain expert needs to be transformed into a description that can be processed by the system. The subsequent steps can be carried out with the help of automated methods. These steps include generation of task descriptors (e.g., keywords) that help to determine the task type, the domain, and the goals. This in turn allows us to search for and retrieve domain-specific knowledge appropriate for the task at hand. This issue is discussed in Section 14.2.

The operation of obtaining the data and its automation may not be a trivial matter, as it is necessary to determine whether the data already exists or not. In the latter case a plan needs to be elaborated regarding how to get it. Sometimes it is necessary to merge data from different sources (databases, OLAP cube, etc.). Section 14.3 discusses these issues in more detail.

The area of preprocessing and transformation has been explored more by various researchers in the AutoDS community. Methods exist for selection of instances and/or elimination of outliers, discretization and various other kinds of transformations. This area is sometimes referred to as *data wrangling*. These transformations can be learned by exploiting existing machine learning techniques (e.g., learning by demonstration). More details can be found in Section 14.3.

One other important area of data science involves decisions regarding the appropriate level of detail to be used in the application. As has been shown, data may be summarized by appropriate aggregation operations, such as *Drill Down/Up* operations in a given OLAP cube. Categorical data may also be transformed by introducing new high-level features. This process involves determining the right level of granularity. Efforts can be made to automate this, but more work is needed before it is possible to offer practical solutions to companies. More details about this issue can be found in Section 14.4.

## Automating the design of complex systems (Chap. 15)

In this book we have dealt with the problem of automating the design of KDD workflows and other data science tasks. A question arises regarding whether the methods can be extended to somewhat more complex tasks. Chapter 15 discusses these issues, but the focus in this book is on symbolic approaches.

We are well aware that many successful applications nowadays, particularly in vision and NLP, exploit deep neural networks (DNNs), CNNs, and RNNs. Despite this, we believe that symbolic approaches continue to be relevant. We believe that this is the case for the following reasons:

- DNNs typically require large training data to work well. In some domains not many examples are available (e.g., occurrences of rare diseases). Also, whenever the examples are supplied by a human (as, for instance, in data wrangling discussed in Chapter 14), we wish the system to be capable of inducing the right transformation on the basis of a modest number of examples. Many systems in this area exploit symbolic representations (e.g., rules), as it is easy to incorporate background knowledge, which is often also in the form of rules.
- It seems that, whenever AI systems need to communicate with humans, it is advantageous to resort to symbolic concepts which can easily be transferred between a human and a system.
- As human reasoning includes both symbolic and subsymbolic parts, it is foreseeable that future AI systems will follow this line too. So it is foreseeable that the two reasoning systems will coexist in a kind of functional symbiosis. Indeed, one current trend involves so-called *explainable AI*.

The structure of this chapter is as follows. Section 15.1 discusses more complex operators that may be required when searching for a solution of more complex tasks. This includes, for instance, conditional operators and operators for iterative processing.

Introduction of new concepts is addressed in Section 15.2. It discusses changes of granularity by the introduction of new concepts. It reviews various approaches explored in the past, such as, constructive induction, propositionalization, reformulation of rules, among others. This section draws attention to some new advances, such as feature construction in deep NNs.

There are tasks that cannot be learned in one go, but rather require a sub-division into subtasks, a plan for learning the constituents, and joining the parts together. This methodology is discussed in Section 15.3. Some tasks require an iterative process in the process of learning. More details on this can be found in Section 15.4. There are problems whose tasks are interdependent. One such problem is analyzed in Section 15.5.

## 1.4 Repositories of Experimental Results (Part III)

### 1.4.1 Repositories of metadata (Chap. 16)

Throughout this book, we discuss the benefits of using knowledge about past datasets, classifiers, and experiments. All around the globe, thousands of machine learning experiments are being executed on a daily basis, generating a constant stream of empirical information on machine learning techniques. Having the details of experiments freely available to others is important, as it enables to reproduce the experiments and verify that the conclusions are correct and use this knowledge to extend the work further. It can thus speed up progress in science.

This chapter starts with a review of online repositories where researchers can share data, code, and experiments. In particular, it covers OpenML, an online platform for sharing and organizing machine learning data automatically and in fine detail. OpenML contains thousands of datasets and algorithms, and millions of experimental results on these experiments. In this chapter we describe the basic philosophy behind it, and its basic components: datasets, tasks, flows, setups, runs, and benchmark suites. OpenML has API bindings in various programming languages, making it easy for users to interact with the API in their native language. One hallmark feature of OpenML is the integration into various machine learning toolboxes, such as Scikit-learn, Weka, and mlR. Users of these toolboxes can automatically upload all the results that they obtained, leading to a large repository of experimental results.

### 1.4.2 Learning from metadata in repositories (Chap. 17)

Having a vast set of experiments, collected and organized in a structured way, allows us to conduct various types of experimental studies. Loosely based upon a categorization of Vanschoren et al. (2012), we present three types of experiments that explore OpenML metadata to investigate certain issues to be described shortly: experiments on a single dataset, on multiple datasets, and experiments involving specific dataset or algorithm characterization.

As for the experiments on a single dataset, Section 17.1 shows how the OpenML metadata can be used for simple benchmarking and, in particular, to assess the impact of varying the settings of a specific hyperparameter. As for the experiments on multiple datasets, Section 17.2 shows how the OpenML metadata can be used to assess the benefit of hyperparameter optimization, and also, the differences on predictions between algorithms. Finally, for the experiments involving specific characteristics, Section 17.3 shows how the OpenML metadata can be used to investigate and answer certain scientific hypotheses, such as on what type of datasets are linear models adequate, and for what type of datasets feature selection is useful. Furthermore, we present studies whose aim is to establish the relative importance of hyperparameters across datasets.

### 1.4.3 Concluding remarks (Chap. 18)

The last chapter of the book (Chap. 18) presents the concluding remarks with respect to the whole book. It includes two sections. As metaknowledge has a central role in many approaches discussed in this book, we analyze this issue in more detail here. In particular, we are concerned with the issue of what kind of metaknowledge is used in different metalearning/AutoML tasks, such as algorithm selection, hyperparameter optimization,



and workflow generation. We draw attention to the fact that some metaknowledge is acquired (learned) by the systems, while other is given (e.g., different aspects of the given configuration space). More details on this can be found in Section 18.1.

Section 18.2 discusses future challenges, such as better integration of metalearning and AutoML approaches and what kind of guidance could be provided by the system for the task of configuring metalearning/AutoML systems to new settings. This task involves (semi-)automatic reduction of configuration spaces to make the search more effective. The last part of this chapter discusses various challenges which we encounter when trying to automate different steps of data science.

## References

- Brazdil, P. and Konolige, K. (1990). *Machine Learning, Meta-Reasoning and Logics*. Kluwer Academic Publishers.
- Chan, P. and Stolfo, S. (1993). Toward parallel and distributed learning by meta-learning. In *Working Notes of the AAAI-93 Workshop on Knowledge Discovery in Databases*, pages 227–240.
- Guyon, I., Bennett, K., Cawley, G., Escalante, H. J., Escalera, S., Ho, T. K., Macià, N., Ray, B., Saeed, M., Statnikov, A., et al. (2015). Design of the 2015 ChaLearn AutoML challenge. In *2015 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE.
- Huisman, M., van Rijn, J. N., and Plaat, A. (2021). A survey of deep meta-learning. *Artificial Intelligence Review*.
- Kowalski, R. (1979). *Logic for Problem Solving*. North-Holland.
- Lemke, C., Budka, M., and Gabrys, B. (2015). Metalearning: a survey of trends and technologies. *Artificial Intelligence Review*, 44(1):117–130.
- Michie, D., Spiegelhalter, D. J., and Taylor, C. C. (1994). *Machine Learning, Neural and Statistical Classification*. Ellis Horwood.
- Rendell, L., Seshu, R., and Tcheng, D. (1987). More robust concept learning using dynamically-variable bias. In *Proceedings of the Fourth International Workshop on Machine Learning*, pages 66–78. Morgan Kaufmann Publishers, Inc.
- Rice, J. R. (1976). The algorithm selection problem. *Advances in Computers*, 15:65–118.
- Shearer, C. (2000). The CRISP-DM model: the new blueprint for data mining. *J Data Warehousing*, 5:13–22.
- Smith-Miles, K. A. (2008). Cross-disciplinary perspectives on meta-learning for algorithm selection. *ACM Computing Surveys (CSUR)*, 41(1):6:1–6:25.
- Vanschoren, J., Blockeel, H., Pfahringer, B., and Holmes, G. (2012). Experiment databases: a new way to share, organize and learn from experiments. *Machine Learning*, 87(2):127–158.



## Metalearning Approaches for Algorithm Selection I (Exploiting Rankings)

**Summary.** This chapter discusses an approach to the problem of algorithm selection, which exploits the performance metadata of algorithms (workflows) on prior tasks to generate recommendations for a given target dataset. The recommendations are in the form of rankings of candidate algorithms. The methodology involves two phases. In the first one, rankings of algorithms/workflows are elaborated on the basis of historical performance data on different datasets. These are subsequently aggregated into a single ranking (e.g. average ranking). In the second phase, the average ranking is used to schedule tests on the target dataset with the objective of identifying the best performing algorithm. This approach requires that an appropriate evaluation measure, such as *accuracy*, is set beforehand. In this chapter we also describe a method that builds this ranking based on a combination of accuracy and runtime, yielding good anytime performance. While this approach is rather simple, it can still provide good recommendations to the user. Although the examples in this chapter are from the classification domain, this approach can be applied to other tasks besides algorithm selection, namely hyperparameter optimization (HPO), as well as the combined algorithm selection and hyperparameter optimization (CASH) problem. As this approach works with discrete data, continuous hyperparameters need to be discretized first.

### 2.1 Introduction

This chapter follows the basic scheme discussed in the introduction which was illustrated in Figures 1.1 and 1.2. However, we focus on a method that exploits a specific kind of metadata that captures performance results of algorithms (workflows) on past datasets, namely rankings. Ranking methods usually rely on some form of metadata, that is, knowledge about how a discrete set of algorithms have performed on a set of historical datasets. This chapter discusses a standard approach that is capable of converting this metadata into a static ranking. The ranking helps users by suggesting an order of algorithms to apply when confronted with a new dataset. The approach is rather simple, but can still provide excellent recommendations to the user. Therefore, we have decided to discuss this approach before various other approaches described in subsequent chapters. Although it can be applied to various domains, all the examples in this chapter are from the classification domain.

The ranking approach can be applied to the algorithm selection (AS) task, hyperparameter optimization (HPO), as well as the combined algorithm selection and hyperpa-

parameter optimization (CASH) problem. Note that this approach works always with discrete data. Therefore, when addressing the HPO or CASH problems using this approach, continuous hyperparameters need to be discretized first.

## Organization of this chapter

Section 2.2 discusses a rather general topic, which is concerned with different forms of recommendation. The system can recommend just a single item, or several items, or a ranked list of items.

Section 2.3 explains the methodology used to construct a ranking of algorithms, based on the available metadata. The methodology involves two phases. In the first one, rankings of algorithms/workflows are elaborated on the basis of historical performance data on different datasets. These are subsequently aggregated into a single ranking (e.g. average ranking). The details are described in Subsection 2.3.1. In the second phase, the average ranking is used to schedule tests on the target dataset with the objective of identifying the best performing algorithm. The details are explained in Subsection 2.3.2. This procedure represents a kind of standard and has been used in many metalearning and AutoML research papers.

Section 2.4 describes a method that incorporates a measure that combines both accuracy and runtime. Indeed, as the aim is to identify well-performing algorithms as soon as possible, this method tests fast algorithms first, and proceeds to slower ones later. The final section (2.5) describes various extensions of the basic approach.

## 2.2 Different Forms of Recommendation

Before explaining the approach that exploits rankings, let us analyze different types of recommendation that a system can provide. The system can recommend the user to apply/explore:

1. Best algorithm in a set,
2. Subset of the top algorithms,
3. Linear ranking,
4. Quasi-linear (weak) ranking,
5. Incomplete ranking.

Although *complete* and *incomplete* rankings can also be referred to as *total* and *partial* rankings, we prefer to use the former terminology here. Table 2.1 illustrates what characterizes each case. In our example, it is assumed that the given portfolio of algorithms includes  $\{a_1, a_2, \dots, a_6\}$ . Figure 2.1 complements this information. Hasse diagrams provide a simple visual representation of rankings (Pavan and Todeschini, 2004), where each node represents an algorithm and directed edges represent the relation “significantly better than”. The figure on the left (part (a)) shows an example of a complete linear ranking. The figure in the center (part (b)) shows an example of complete quasi-linear ranking. The figure on the right (part (c)) shows an example of an incomplete linear ranking. Each of these figures corresponds to the rankings in rows 3 to 5 of Table 2.1. More details about each form are provided in the following subsections.

Table 2.1: Examples of different forms of recommendation

1. Best in a set	$a_3$												
2. Subset	$\{a_3, a_1, a_5\}$												
	Rank												
	<table border="1"> <tr> <th>1</th> <th>2</th> <th>3</th> <th>4</th> <th>5</th> <th>6</th> </tr> <tr> <td><math>a_3</math></td> <td><math>a_1</math></td> <td><math>a_5</math></td> <td><math>a_6</math></td> <td><math>a_4</math></td> <td><math>a_2</math></td> </tr> </table>	1	2	3	4	5	6	$a_3$	$a_1$	$a_5$	$a_6$	$a_4$	$a_2$
1	2	3	4	5	6								
$a_3$	$a_1$	$a_5$	$a_6$	$a_4$	$a_2$								
3. Linear and complete ranking													
4. Quasi-linear and complete ranking	$\overline{a_3 a_1} \ a_5 \ \overline{a_6 a_4} \ a_2$												
5. Linear and incomplete ranking	$a_3 \ a_1 \ a_5 \ a_6$												

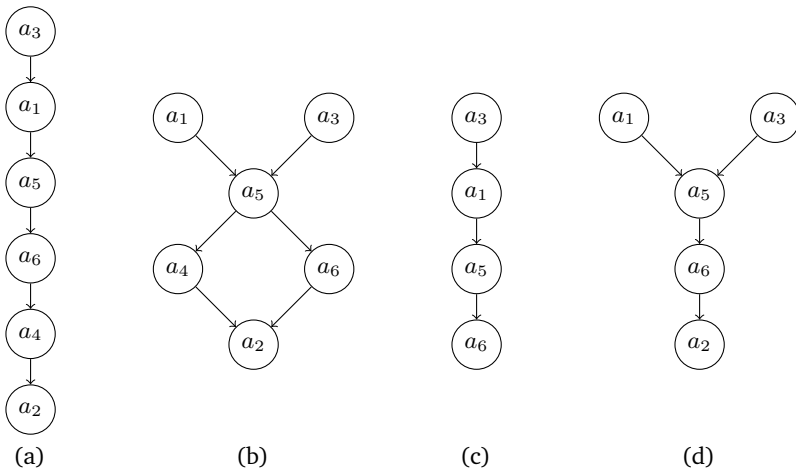


Fig. 2.1: Representation of rankings using Hasse diagrams: (a) complete linear ranking; (b) complete quasi-linear ranking; (c) incomplete linear ranking; (d) incomplete quasi-linear ranking

### 2.2.1 Best algorithm in a set

The first form consists of identifying the algorithm that is expected to obtain the best performance in the set of base-level algorithms (Pfahring et al., 2000; Kalousis, 2002). Note that this is formally a ranking of size one. Intuitively, most data scientists make use of such ranking, when applying their favorite algorithm first on a new dataset. One way of doing this is by identifying the best algorithm for each dataset. Then the information gathered needs to be aggregated. One possibility is to use the algorithm that was the best one on most datasets.

We note that this is similar to selecting one of the top items in the linear ranking. As this strategy does not use search, there is a possibility that the algorithm selected may not be the truly best one. Consequently, we may get a substandard result.

### 2.2.2 Subset of the top algorithms

Methods that use this form of recommendation suggest a (usually small) subset of algorithms that are expected to perform well on the given problem (Todorovski and Džeroski, 1999; Kalousis and Theoharis, 1999; Kalousis, 2002). One way of determining this subset is by identifying the best algorithm for each dataset. If the best algorithm is tied with others, we may simply select the first one in the order as they appear. Then the information gathered can be aggregated by taking the union of all the algorithms identified. So, supposing that the training datasets include  $n$  datasets, we will end up with a subset with at most  $n$  elements. We note that this method ignores all algorithms that achieve a comparable performance to the best algorithm. Consequently, there is a chance that the subset selected may not include the truly best one.

To increase the chances, we may use a more elaborate strategy. It involves identifying the best algorithm for each dataset and all other algorithms that also perform equally well. The notion of *performing well* on a given dataset is typically defined in relative terms. For example, having a model that makes predictions that are correct in 50% of the cases is considered very good on some datasets, whereas on others this might be considered mediocre. The following subsection discusses the details.

#### Identifying algorithms with comparable performance

Assuming that the best algorithm has been identified ( $a_*$ ) a question arises whether there are other algorithms (e.g.,  $a_c$ ) with comparable performance. One possibility is to carry out a suitable statistical test to determine whether the performance of some candidate algorithms is significantly worse than the performance of the best one (Kalousis and Theoharis, 1999; Kalousis, 2002). An algorithm is included among the “good” subset, if it is not significantly worse than the best one. In practice, researchers have applied both parametric test (e.g., t-test) and non-parametric tests (e.g., Wilcoxon signed-rank test (Neave and Worthington, 1992)). This approach requires that the tests of the algorithms are carried out using cross-validation (CV), as it requires information gathered in different folds of the CV procedure.

If the fold information is not available, it is possible to use an approximate method that may still provide a quite satisfactory solution. This approach involves establishing a *margin* relative to the performance of the best algorithm on that dataset. All the algorithms with a performance within the margin are considered to perform well too. In classification, the margin can be defined in the following way (Brazdil et al., 1994; Gama and Brazdil, 1995; Todorovski and Džeroski, 1999):

$$\left( e_{min}, e_{min} + k \sqrt{\frac{e_{min} (1 - e_{min})}{n}} \right), \quad (2.1)$$

where  $e_{min}$  is the error of the best algorithm,  $n$  is the number of examples, and  $k$  is a user-defined confidence level that affects the size of the margin. We note that this approach is based on an assumption that errors are normally distributed.

Both approaches are related, because the interval of confidence of the first method can be related to the margin used in the second one. Thus, any algorithm with a performance within this margin can be considered to be not significantly worse than the best one. This method results in a small subset of algorithms for each dataset (those that perform well).

## Aggregating subsets

The subsets generated in the previous step need to be aggregated. This can be done, for instance, by taking a union. The algorithms in the final subset can be ordered according to how many datasets they are involved in. If a particular algorithm  $a_i$  appears in several subsets, while  $a_j$  only once,  $a_i$  could be attributed a higher rank than  $a_j$ , as the probability that  $a_i$  will achieve better performance on the target dataset is higher when compared with  $a_j$ .

This approach has the advantage that the search phase involves more than one algorithm and, consequently, the chance that the truly best algorithm is included in it is higher.

This topic is related to the problem of reducing the set of algorithm in a given portfolio, which is discussed in Chapter 8.

### 2.2.3 Linear ranking

Rankings have been used by many researches in the past (Brazdil et al., 1994; Soares and Brazdil, 2000; Keller et al., 2000; Brazdil et al., 2003). Typically, the order indicated in the ranking is the order that should be followed in the experimentation phase. Many systems tend to use *linear and complete* ranking. It is shown in row 3 of Table 2.1 and also in Figure 2.1(a). It is referred to as *linear ranking* because the ranks are different for all algorithms. Additionally, it is a *complete ranking* because all the algorithms  $a_1, \dots, a_6$  have their rank defined (Cook et al., 2007).

This type of ranking has a disadvantage, as it cannot represent the case when two algorithms are tied on a given dataset (i.e., their performance is not significantly different).

### 2.2.4 Quasi-linear (weak) ranking

Whenever two or more algorithms are tied, a *quasi-linear* (sometimes also called *weak*) ranking can be used (Cook et al., 1996). An example is shown in Table 2.1 (row 4). The line above the algorithm names (as in  $\overline{a_3 a_1}$ ) indicates that the performance of the corresponding algorithms is not significantly different. An alternative representation is shown in Figure 2.1(b).

Quasi-linear rankings arise when there is not enough data permitting to distinguish their (relative) performance on the dataset at hand, or if the algorithms are truly indistinguishable. In this case, the problem can be resolved by assigning the same rank to all tied algorithms.

A meta-learning method that provides recommendations in the form of quasi-linear rankings is proposed in Brazdil et al. (2001). The method is an adaptation of the  $k$ -NN ranking approach discussed in the next section (2.3). It identifies algorithms with equivalent performance and includes only one of the algorithms in the recommendation.

### 2.2.5 Incomplete ranking

Both linear and quasi-linear rankings can be incomplete, as only some algorithms were used in the tests. So a question arises on what to do. In our view we need to distinguish the following two rather different situations. The first one arises when some algorithms

were excluded from consideration for a particular reason (e.g., they sometimes crash; they are difficult to use; they are rather slow etc.). In this case, we should just resort to the incomplete ranking, as if it were complete.

The second situation occurs when new algorithms were developed and so they need to be added to the existing algorithm set (portfolio). Obviously, it is necessary to run tests to extend the existing metadata. The metadata does not necessarily need to be complete. The topic of complete vs. incomplete metadata is discussed further in Chapter 8 (Section 8.8). If the metalearning method in question can work with incomplete metadata, a question arises regarding which tests should be conducted in preference to others. Chapter 8 (Section 8.9) describes some strategies developed in the area of *multi-armed bandits* that can be used for this purpose.

## 2.2.6 Searching for the best algorithm within a given budget

Rankings are particularly suitable for algorithm recommendation, because the metalearning system can be developed without any information about how many base-algorithms the user will try out. This number depends on the available computational resources (i.e., budget) and the importance of achieving good performance (e.g., accuracy) on the target problem. If time is the critical factor, only very few alternatives should be selected. On the other hand, if the critical factor is, say, accuracy, then more algorithms should be examined, as it increases the chance of obtaining the potentially best result. This was confirmed by various experimental studies (e.g., Brazdil et al. (2003)).

## 2.3 Ranking Models for Algorithm Selection

The approach described in this chapter is based on the following assumption: if the aim is to identify a well-performing algorithm, it is not as important to accurately predict their *true performance*, as it is to predict their *relative performance*. The task of algorithm recommendation can thus be defined as the task of ranking algorithms according to their predicted performance.

To address this problem with the help of machine learning, we follow the two-phase approach described in introductory Chapter 1. In the first phase, it is necessary to collect data describing the performance of algorithms, referred to as *performance metadata*. Some approaches also exploit certain characteristics of base-level tasks, referred to as *task/dataset metadata*. The metadata permits to generate a meta-level model. In the approach discussed in this chapter, the meta-level model is in the form of a ranked list of algorithms (workflows). More details about this process are provided in Subsection 2.3.1.

After the meta-level model has been generated, it is possible to advance to the second phase. The meta-level model can be used to obtain recommendation for the target dataset. More details about this are given in Subsection 2.3.2.

### 2.3.1 Generating a meta-model in the form of rankings

The process of generating a meta-model involves the following steps:

1. Evaluate all algorithms on all datasets.
2. Use dataset similarity to identify the relevant parts of metadata.



3. Use all performance results to elaborate a ranking of all algorithms, representing a meta-model.

This process is illustrated in Figure 2.2.

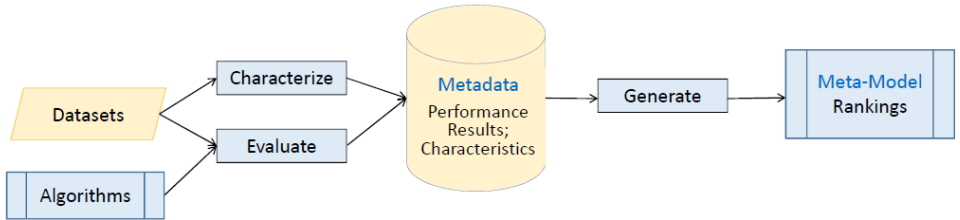


Fig. 2.2: Generating a meta-model in the form of a ranking

## Gathering performance results

This step consists of running tests to collect performance results (performance metadata). We assume that the performance results are stored in a *performance matrix*  $P$ , where rows represent datasets and columns algorithms. More precisely, the labels (names) of the rows are the names of the dataset used, i.e.,  $D = \{d_1, \dots, d_k\}$ . Similarly, the labels (names) of the columns are the algorithm names ( $A = \{a_1, \dots, a_n\}$ ). Each slot  $P(i, j)$  holds the performance of algorithm  $j$  on dataset  $i$  after the respective evaluation was carried out.

Let us clarify what kind of performance measures can be used here. In the classification domain, some common measures are accuracy, AUC, F1, microF1 and macroF1, among others, described in books on machine learning (ML) (e.g., Mitchell (1997); Hand et al. (2001)). In the examples in this chapter we use mainly *predictive accuracy*, which is defined as the proportion of test examples that were classified correctly by the model.

Details of this process are shown in Algorithm 2.1. To simplify the description, we assume that the initial performance matrix  $P_0$ , which is initially empty, is given. The aim is to generate a rank matrix  $R$ , which has a similar format to the performance matrix  $P$ , but instead of performance values it includes ranks. Table 2.3 shows an example of test results converted to ranks for 3 datasets and 10 algorithms.

The conversion to ranks is simple. The best algorithm is assigned rank 1, the runner-up is assigned rank 2, and so on.

Note that cross-validating each algorithm on each dataset is a costly procedure, and only feasible for a relatively small number of algorithms and datasets. In many studies, this information is assumed to be readily available, e.g., by using an existing source of such metadata, such as OpenML discussed in Chapter 16.

```

input :  $P_0$  (empty performance matrix)
output:  $R$  (ranking matrix)
begin
  |  $P \leftarrow P_0$ 
end
foreach row (dataset)  $i$  in  $P$  do
  | foreach column (algorithm)  $j$  in  $P$  do
  | | Evaluate the algorithm  $j$  on dataset  $i$  using cross-validation (CV):
  | |  $P(i, j) \leftarrow CV(j, i)$ 
  | end
end
foreach column (algorithm)  $j$  in  $P$  do
  | Convert the performance vector into a ranking:
  |  $R(:, j) \leftarrow rank(P(:, j))$ 
end

```

**Algorithm 2.1:** Constructing performance and rank matrices

## Aggregating performance results into a single ranking

This subsection includes a description of the process of aggregation of set of rankings obtained in different tests into a single aggregated ranking. The aggregation is done on the basis of a particular ranking criterion that can be chosen. Different criteria exist:

- average rank,
- median rank,
- rank based on significant wins and/or losses.

**Aggregating by average rank:** This method can be regarded as a variant of Borda's method (Lin, 2010). This method was inspired by Friedman's *M statistic* (Neave and Worthington, 1992). The method based on average ranks is referred to as the *average ranking (AR)* method. It requires that we have, for each dataset, a ranking of all algorithms based on performance results.

Let  $R_{i,j}$  be the rank of base-algorithm  $a_j$  ( $j = 1, \dots, n$ ) on dataset  $i$ , where  $n$  is the number of algorithms. The *average rank* for each  $a_j$  is

$$\bar{R}_j = \frac{\sum_{i=1}^n R_{i,j}}{k}, \quad (2.2)$$

where  $k$  represents the number of datasets. The final ranking is obtained by ordering the average ranks and assigning ranks to the algorithms accordingly. An example is given further on.

**Aggregating by median rank:** This method is similar to the one just described. Instead of calculating the mean rank using Eq. 2.2, it is necessary to obtain the median value. The method based on median ranks is referred to as the *median ranking (MR)* method. Cachada (2017) compared the two methods — AR and MR — on a set-up that included test results of 368 different workflows on 37 datasets. The results showed that MR achieved somewhat better results than AR, although the differences were not statistically significant.

Table 2.2: Classification algorithms

C5b	Boosted decision trees (C5.0)
C5r	Decision tree-based rule set (C5.0)
C5t	Decision tree (C5.0)
IB1	1-Nearest neighbor (MLC++)
LD	Linear discriminant
Lt	Decision trees with linear combination of attributes
MLP	Multilayer perceptron (Clementine)
NB	Naïve Bayes
RBFN	Radial basis function network (Clementine)
RIP	Rule sets (RIPPER)

Table 2.3: Example of an average ranking based on three datasets

Algorithm:	C5b	C5r	C5t	MLP	RBFN	LD	Lt	IB1	NB	RIP
byzantine	2	6	7	10	9	5	4	1	3	8
isolet	2	5	7	10	9	1	6	4	3	8
pendigits	2	4	6	7	10	8	3	1	9	5
Average rank scores $\bar{R}_i$	2.0	5.0	6.7	9.0	9.3	4.7	4.3	2.0	5.0	7.0
Average ranking	1.5	5.5	7	9	10	4	3	1.5	5.5	8

**Aggregating by the number of significant wins and/or losses:** This method establishes the rank of each algorithm  $a_i$  and takes into account the number of significant wins and/or losses over other algorithms. A *significant win* of algorithm  $a_i$  over algorithm  $a_j$  is defined as a performance difference that is statistically significant. This method was explored by various researchers in the past (Brazdil et al., 2003; Leite and Brazdil, 2010).

### Example: elaborating an average ranking

The use of the average ranking method for the problem of algorithm recommendation is illustrated here on an example. The metadata used captures the performance of 10 classification algorithms (see Table 2.2) and 57 datasets from the UCI repository (Asuncion and Newman, 2007). More information about the experimental set-up can be found elsewhere (Brazdil et al., 2003).

The goal here is to construct the ranking of the algorithms on the basis of rankings obtained on three datasets listed in Table 2.3. The corresponding average rank scores,  $\bar{R}_j$ , obtained by aggregating the individual rankings are shown in that table. The rank scores can be used to reorder the algorithms and, this way, obtain the recommended ranking (C5b, IB1 .. RBFN). This ranking provides guidance concerning the future experiments to be carried out on the target dataset.

We note that the average ranking contains two pairs of ties. One of them involves C5b and IB1, which share the first two ranks and hence have been attributed rank 1.5 in our table. A tie means that there is no evidence that either of the algorithms (in this case C5b and IB1) would achieve different performance, based on the metadata used. The user can carry our random selection, or else use some other criterion in the selection process (e.g., runtime).

A question that follows is whether the predicted (or recommended) ranking is an accurate prediction of the true ranking, i.e., of the relative performance of the algorithms on the target dataset. This issue is addressed in the next subsection (2.3.2) and also in Chapter 3. We observe that the two rankings are more or less similar. The largest error is made in the prediction of the ranks of LD and NB (four rank positions), but the majority of the errors are of two positions. Nevertheless, a proper evaluation methodology is necessary. That is, we need methods that enable us to quantify and compare the quality of rankings in a systematic way. This is explained in Section 2.3.3.

### 2.3.2 Using the ranking meta-model for predictions (top- $n$ strategy)

The meta-model discussed in the previous subsection can be used to provide a recommendation regarding which algorithm to select for the target dataset. This scheme is illustrated in Figure 2.3. Algorithm 2.2 provides more details about the method.

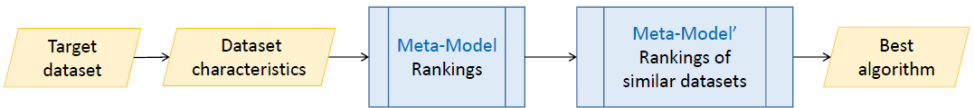


Fig. 2.3: Using the average ranking (AR) method for the prediction of the best algorithm

As the recommendation is in the form of a ranking, it is thus reasonable to expect that the order recommended will be followed by the user. The algorithm (workflow) ranked in the first position will most likely be considered first, followed by the one ranked second, and so on. This is done by cross-validating the algorithms in this order on the target dataset. After each cross-validation test, the performance is stored, and the algorithm with the highest stored performance is the winner. A question arises regarding how many algorithms the user should select.

A *top-n* execution can be used for this purpose (Brazdil et al., 2003). This method consists of simulating that the top  $n$  items will be selected. When studying the performance of the *top-n* scheme, we will normally let it run until the end. In other words, the parameter  $n$  will normally be set to the maximum value, corresponding to the number of algorithms. This has the advantage that we can inspect the results at different stages of execution. One other alternative to fixing  $n$  consists of fixing the time budget (see Section 2.4).

### Example

The method is illustrated by following the recommended ranking presented in Table 2.4 and carrying out tests on `waveform40` dataset. The table also presents the accuracy obtained by each algorithm and the corresponding runtime. The first item listed in this table represents the default classification accuracy on this dataset. As the dataset `waveform40` includes three classes, we can assume that the mean accuracy will be  $1/3$ , under the

```

input :  $A = \{a_1, \dots, a_n\}$  (list of algorithms ordered by ranks)
           $d_{new}$  (target dataset)
           $n$  (number of algorithms to test)
output:  $a^*$  (algorithm with the best performance)
           $p^*$  (performance of  $a^*$ )
           $t_{accum}$  (time used)

begin
   $a^* \leftarrow A[1]$  (initialize  $a^*$ )
  Evaluate the first algorithm and initialize values:
   $(p^*, t_{accum}) \leftarrow CV(A[1], d_{new})$ 
  foreach  $i \in \{2, \dots, n\}$  do
    Evaluate the  $i$ -th algorithm:
     $(p_c, t_c) \leftarrow CV(A[i], d_{new})$ 
    if  $p_c > p^*$  then
       $a^* \leftarrow A[i]$ 
    end
     $p^* \leftarrow \max(p_c, p^*)$ 
     $t_{accum} \leftarrow t_c + t_{accum}$ 
  end
end

```

**Algorithm 2.2:** Top- $n$  procedure

Table 2.4: Results of executing a given recommended ranking on waveform40 dataset

Recommended Ranking	Def	MLP	RBFN	LD	Lt	C5b	NB	RIP	C5r	C5t	IB1
	0	1	2	3	4	5	6	7	8	9	10
Accuracy	0.33	0.81	0.85	0.86	0.84	0.82	0.80	0.79	0.78	0.76	0.70
Runtime	0	99.70	441.52	1.73	9.78	44.91	3.55	66.18	11.44	4.05	34.91
Runtime accum.	0	99.7	541.2	542.9	552.7	597.6	601.2	667.4	678.8	682.9	717.8

assumption that the classes are equally probable. We assume that determining this takes virtually no time.

Figure 2.4 shows how the accuracy evolves with the number of algorithms executed ( $n$ ). The first algorithm executed is MLP, obtaining an accuracy of 81.4%. Once the next algorithm in the ranking (RBFN) is executed a significant increase in accuracy is obtained, reaching 85.1%. The execution of the next algorithm in the ranking, LD, yields a smaller increase (86.0%). The remaining algorithms do not alter this situation much. Note that when using the top- $n$  strategy, the performance never goes down. In order to understand this, we need to revisit what it actually does. It measures the highest obtained performance in cross-validation tests so far. As the set of cross-validated algorithms grows, this value cannot decrease.

Figure 2.5 shows the evolution of accuracy on runtime. This plot provides more information that is relevant for the assessment of the recommended ranking. It shows that, although the execution of RBFN provides a significant improvement in accuracy, it does so at the cost of a comparatively much larger runtime (441 s.). The plot also shows that, although the gain obtained with LD is smaller, the corresponding runtime is quite small (less than 2 s.).

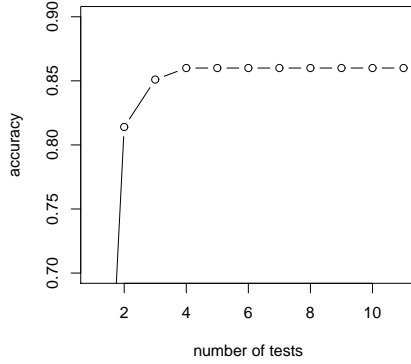


Fig. 2.4: Dependence of accuracy on number of tests with top-*n* execution

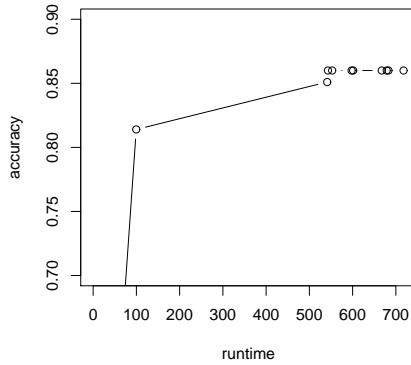


Fig. 2.5: Dependence of accuracy on runtime with top-*n* execution

Section 2.4 discusses another variant of the ranking method where runtime is incorporated into the algorithm. It is shown that this leads to marked improvements.

### 2.3.3 Evaluation of recommended rankings

An important question is how good/bad the recommendations are. Chapter 3 discusses the methodology that can be adopted to assess the quality of the recommendations generated by the system. It describes two different approaches. The first one aims to assess

the quality by comparing it with the correct ranking, representing a golden standard. The second one aims to assess the effects on base-level performance when the ranking is followed.

## 2.4 Using a Combined Measure of Accuracy and Runtime

Rankings can be based on any performance measure we might wish to consider. Measures that combine accuracy (or AUC, F1, etc.) and runtime are of particular interest. Indeed, beforehand we do not know for sure which algorithms will perform well on the target dataset, and therefore a lot of time can be wasted on slower algorithms. Ideally, we want to schedule first the CV tests of fast, but relatively well-performing algorithms, before other, slower ones.

As Abdulrahman et al. (2018) have shown, this can lead to substantial speed-ups when seeking the best algorithm for a given target dataset. The concept of a combined measure that combines accuracy and runtime is not new. Various authors have proposed such a measure, including, for instance, Brazdil et al. (2003) who proposed the measure *ARR*. However, as was shown later (Abdulrahman et al., 2018), this measure is not monotonic. The authors introduced a measure A3R (shown in Chapter 5) that does not suffer from this shortcoming. Here we use a simplified version of this function, referred to as A3R' (van Rijn et al., 2015), which is defined as follows:

$$A3R'_{a_j}{}^{d_i} = \frac{P_{a_j}^{d_i}}{(T_{a_j}^{d_i})^Q}, \quad (2.3)$$

where  $P_{a_j}^{d_i}$  represents the performance (e.g., accuracy) of algorithm  $a_j$  on dataset  $d_i$  and  $T_{a_j}^{d_i}$  the corresponding runtime. This function requires that a correct balance is established between the importance of accuracy and runtime. This is done by the parameter  $Q$ , which is in effect a scaling factor. Typically,  $Q$  would be a rather small number, such as  $1/64$ , representing in effect, the  $64^{\text{th}}$  root. This is motivated by the fact that runtimes vary much more than accuracies. It is not uncommon that one particular algorithm is three orders of magnitude slower (or faster) than another. Obviously, we do not want the time ratios to completely dominate the equation.

For instance, when the setting is  $Q = 1/64$ , an algorithm that is 1000 times slower would yield a denominator of 1.114. This would thus be equivalent to the faster reference algorithm only if its accuracy were 11.4% higher than the reference algorithm.

A question arises regarding what the best setting for the parameter  $Q$  is. Abdulrahman et al. (2018) have investigated this issue. They have considered various settings for  $Q$ , including  $1/4$ ,  $1/16$ ,  $1/64$ ,  $1/128$  and  $1/258$ . They have shown that the setting  $Q = 1/64$  was the best one, as it permitted to identify the good-performing algorithms earlier than the other options. This setting can be regarded as a useful default setting.

Average ranking is elaborated in the way described in Section 2.3. In addition to this, runtimes could be normalized for each dataset. Normalization is discussed in Chapter 3. For each dataset, the algorithms are ordered according to the performance measure chosen (here A3R) and ranks are assigned accordingly.

The average ranking is constructed by applying the Eq. 2.2. This upgrade has a rather dramatic effect on the loss curves, as can be seen in Figure 2.6 reproduced from Abdulrahman et al. (2018).

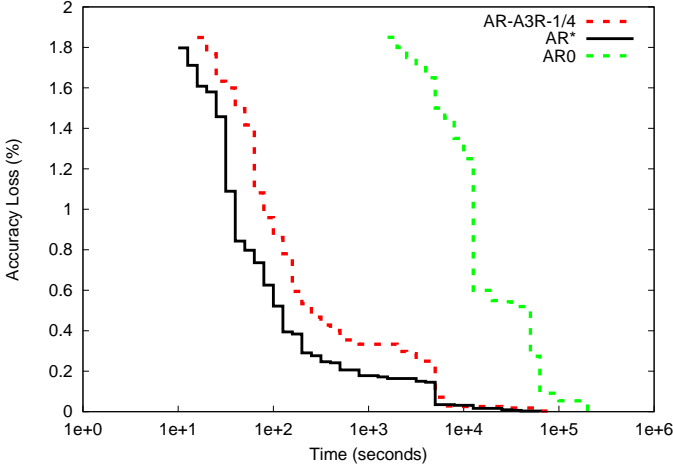


Fig. 2.6: Loss-time curves for A3R-based and accuracy-based average ranking

The loss curve of AR\*, corresponding to the A3R-based average ranking method with the parameter setting  $P = 1/64$ , obtains a much better curve than the version AR0, corresponding to the case when only accuracy matters. With AR\* the loss of 1% is achieved before reaching 100 seconds, while AR0 requires more than 10,000 seconds to obtain the same loss.

## 2.5 Extensions and Other Approaches

### 2.5.1 Using average ranking method to recommend workflows

Currently the attention of both researchers and practitioners is turning to the selection and configuration of workflows (pipelines) of operations. These typically include different preprocessing operations followed by the application of machine learning algorithms with appropriate hyperparameter configurations.

In this section we briefly mention the work of Cachada et al. (2017), which uses the variant AR\* to recommend a workflow for a new dataset. The workflows may include a particular feature selection method (*correlation feature selection*, CFS (Hall, 1999)) and a particular classification algorithm selected from a given (62 in total). About half of these are ensembles. Besides, the authors also use different versions of some classification algorithms (algorithms with different settings of hyperparameters).

The authors show that AR\* was able to select good-performing workflows. Their experiments also show that including feature selection and hyperparameter configurations as alternatives is, on the whole, beneficial. More details about this approach and other related approaches are given in Chapter 7.



### 2.5.2 Rankings may downgrade algorithms that are dataset experts

The rankings discussed in this chapter focus on algorithms that have high performance overall. Although this seems understandable, it also has a potential downside. Consider for example the following case, as shown in Table 2.5.

Table 2.5: Example metadataset, consisting of datasets  $d_1 \dots d_4$  and algorithms  $a_1 \dots a_4$ . The table on the left shows the performance values of each algorithm on each dataset. The table on the right shows the ranks of each algorithm on each dataset

	$d_1$	$d_2$	$d_3$	$d_4$		$d_1$	$d_2$	$d_3$	$d_4$
$a_1$	0.66	0.63	<b>0.95</b>	0.65	$a_1$	4	4	<b>1</b>	4
$a_2$	<b>0.90</b>	<b>0.81</b>	0.89	<b>0.84</b>	$a_2$	<b>1</b>	<b>1</b>	2	<b>1</b>
$a_3$	0.82	0.79	0.83	0.83	$a_3$	2	2	3	2
$a_4$	0.74	0.76	0.84	0.77	$a_4$	3	3	4	3

As can be seen, the complete ranking would be  $a_2, a_3, a_4, a_1$ , suggesting that  $a_1$  is the worst algorithm to test. Looking at it from a different perspective,  $a_1$  is, in fact, the only algorithm that manages to exceed the performance of  $a_2$  on one dataset ( $d_3$ ). In other words, when considering the performance on each dataset, algorithms  $a_2$  and  $a_1$  are the only algorithms that lie on the Pareto front.

The issue of how to identify and eliminate certain algorithms from a given set (which can be converted to a ranking) was addressed by Brazdil et al. (2001) and Abdulrahman et al. (2019). This approach is further detailed in Chapter 8 (Section 8.5).

Wistuba et al. (2015) investigated how to create a ranking of complementary algorithms. Pfisterer et al. (2018) showed that creating the optimal ranking based on metadata is an NP-complete problem, and proposed a greedy approach.

### 2.5.3 Approaches based on multi-criteria analysis with DEA

An alternative to designing a combined measure of two (or more) performance criteria is to use data envelopment analysis (DEA) (Charnes et al., 1978) for multi-criteria evaluation of learning algorithms (Nakhaeizadeh and Schnabl, 1997). One of the important characteristics of DEA is that the weights of the different criteria are determined by the method and not the user. However, this flexibility may not always be entirely suitable, and so Nakhaeizadeh and Schnabl (1998) have proposed a variant of DEA that enables to personalize the relative importance of different criteria. For instance, one user may prefer faster algorithms that generate interpretable models even if they are not so accurate.

### 2.5.4 Using dataset similarity to identify relevant parts of metadata

Section 2.3 described how to create a ranking model based on all metadata. However, not all metadata gathered in the experiments may be relevant to the task at hand. If

Table 2.6: Example of metadata with missing test results

<i>Alg.</i>	$D_1$	$D_2$	$D_3$	$D_4$	$D_5$	$D_6$
$a_1$	0.85	0.77		0.98		0.82
$a_2$		0.55	0.67	0.68	0.66	
$a_3$	0.63		0.55	0.89		0.46
$a_4$	0.45	0.52	0.34		0.44	0.63
$a_5$	0.78	0.87	0.61	0.34	0.42	
$a_6$		0.99		0.89		0.22

the metadata includes test results on datasets that are rather different from the current task, using it may have an adverse effect on performance. So a question arises on how to identify which metadata is relevant for the given task.

One common approach involves using dataset characteristics to identify a subset of the most similar datasets to the target dataset and use the metadata associated with these datasets only. The approach presented here is motivated by the following hypothesis. If datasets are similar, then the algorithm rankings obtained on those datasets will be similar too. Dataset characteristics are sometimes called, in this context, *metafeatures*. Dataset characteristics are discussed in detail in Chapter 4.

We want to stress that a ranking approach can often be used without this step with quite satisfactory results. However, if dataset characteristics are not considered, the target dataset does not affect the order in which the algorithms are tested. In other words, the method follows a fixed schedule. Although this may not affect the final algorithm identified, more time may be needed to identify it. Having a flexible schedule may bring advantages. One such flexible schedule, different from the one discussed here, is presented in Chapter 5 (Section 5.8), which discusses an approach referred to as *active testing*.

### 2.5.5 Dealing with incomplete rankings

In practice, it sometimes happens that a certain proportion of test results is missing. That is, the test results of some algorithms on some datasets may be missing. So, if this happens, the resulting ranking will be incomplete. An example of an incomplete ranking is shown in row 5 in Table 2.1 and also in Figure 2.1(c). Table 2.6 shows an example with six algorithms ( $a_1 \dots a_6$ ) and six datasets ( $D_1 \dots D_6$ ). Note that in each column two out of the six results are missing.

Given that incomplete test results often arise in practice, a question arises regarding what to do. One simple and obvious answer is to complete the results. However, this may not always be possible, as a particular algorithm may have simply failed to run, or the know-how regarding how to run it is no more available. Also, running experiments with ML algorithms may often require substantial computational resources.

So, the other possibility is to use the incomplete metadata in the process of identifying the potentially best algorithm for the target dataset. This issue was investigated by Abdulrahman et al. (2018). The authors have shown that the performance of the average ranking method  $AR^*$  that uses the combined measure of accuracy and runtime (discussed in Section 2.4), is not affected even by 50% of omissions in the metadata. This

has an important implication. It indicates that we do not need to carry out exhaustive testing to provide quite good metamodels.

Abdulrahman et al. (2018) have shown that the method for aggregating incomplete rankings needs to be modified. More details are provided in the following subsection.

### Aggregating incomplete rankings

Many diverse methods exist that can be used to aggregate incomplete rankings. According to Lin (2010), these can be divided into three categories: heuristic algorithms, Markov chain methods, and stochastic optimization methods. The last category includes, for instance, *cross-entropy Monte Carlo* (CEMC) methods.

Merging incomplete rankings may involve rankings of different size. Some approaches require that these rankings be completed before aggregation. Let us consider a simple example. Suppose ranking  $R_1$  represents four elements, namely  $(a_1, a_3, a_4, a_2)$ , while  $R_2$  represents just two elements  $(a_2, a_1)$ . Some approaches would require that the missing elements in  $R_2$  (i.e.,  $a_3, a_4$ ) be attributed a concrete rank (e.g., rank 3). For instance, this strategy is used in package *RankAggreg* of R (Pihur et al., 2009). This is not right, as one should not be forced to assume some information when in fact there is none.

Abdulrahman et al. (2018) have proposed a relatively simple method for aggregating incomplete rankings that avoids this shortcoming. The method is based on the following observation: If two rankings are of unequal length, the ranks in the shorter one provide much less information than the ranks in the longer ranking. It is quite easy to see why. A substandard algorithm may appear in the first position if it is compared with another similar algorithm. The authors provide experimental evidence that this method provides quite good results despite its simplicity.

## References

- Abdulrahman, S., Brazdil, P., van Rijn, J. N., and Vanschoren, J. (2018). Speeding up algorithm selection using average ranking and active testing by introducing runtime. *Machine Learning*, 107(1):79–108.
- Abdulrahman, S., Brazdil, P., Zainon, W., and Alhassan, A. (2019). Simplifying the algorithm selection using reduction of rankings of classification algorithms. In *ICSCA '19, Proceedings of the 2019 8th Int. Conf. on Software and Computer Applications, Malaysia*, pages 140–148. ACM, New York.
- Asuncion, A. and Newman, D. (2007). UCI machine learning repository.
- Brazdil, P., Gama, J., and Henery, B. (1994). Characterizing the applicability of classification algorithms using meta-level learning. In Bergadano, F. and De Raedt, L., editors, *Proceedings of the European Conference on Machine Learning (ECML94)*, pages 83–102. Springer-Verlag.
- Brazdil, P., Soares, C., and da Costa, J. P. (2003). Ranking learning algorithms: Using IBL and meta-learning on accuracy and time results. *Machine Learning*, 50(3):251–277.
- Brazdil, P., Soares, C., and Pereira, R. (2001). Reducing rankings of classifiers by eliminating redundant cases. In Brazdil, P. and Jorge, A., editors, *Proceedings of the 10th Portuguese Conference on Artificial Intelligence (EPIA2001)*. Springer.
- Cachada, M. (2017). Ranking classification algorithms on past performance. Master's thesis, Faculty of Economics, University of Porto.

- Cachada, M., Abdulrahman, S., and Brazdil, P. (2017). Combining feature and algorithm hyperparameter selection using some metalearning methods. In *Proc. of Workshop AutoML 2017, CEUR Proceedings Vol-1998*, pages 75–87.
- Charnes, A., Cooper, W., and Rhodes, E. (1978). Measuring the efficiency of decision making units. *European Journal of Operational Research*, 2(6):429–444.
- Cook, W. D., Golany, B., Penn, M., and Raviv, T. (2007). Creating a consensus ranking of proposals from reviewers’ partial ordinal rankings. *Computers & Operations Research*, 34(4):954–965.
- Cook, W. D., Kress, M., and Seiford, L. W. (1996). A general framework for distance-based consensus in ordinal ranking models. *European Journal of Operational Research*, 96(2):392–397.
- Gama, J. and Brazdil, P. (1995). Characterization of classification algorithms. In Pinto-Ferreira, C. and Mamede, N. J., editors, *Progress in Artificial Intelligence, Proceedings of the Seventh Portuguese Conference on Artificial Intelligence*, pages 189–200. Springer-Verlag.
- Hall, M. (1999). *Correlation-based feature selection for machine learning*. PhD thesis, University of Waikato.
- Hand, D., Mannila, H., and Smyth, P. (2001). *Principles of Data Mining*. MIT Press.
- Kalousis, A. (2002). *Algorithm Selection via Meta-Learning*. PhD thesis, University of Geneva, Department of Computer Science.
- Kalousis, A. and Theoharis, T. (1999). NOEMON: Design, implementation and performance results of an intelligent assistant for classifier selection. *Intelligent Data Analysis*, 3(5):319–337.
- Keller, J., Paterson, I., and Berrer, H. (2000). An integrated concept for multi-criteria ranking of data-mining algorithms. In Keller, J. and Giraud-Carrier, C., editors, *Proceedings of the ECML Workshop on Meta-Learning: Building Automatic Advice Strategies for Model Selection and Method Combination*, pages 73–85.
- Leite, R. and Brazdil, P. (2010). Active testing strategy to predict the best classification algorithm via sampling and metalearning. In *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI)*, pages 309–314.
- Lin, S. (2010). Rank aggregation methods. *WIREs Computational Statistics*, 2:555–570.
- Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill.
- Nakhaeizadeh, G. and Schnabl, A. (1997). Development of multi-criteria metrics for evaluation of data mining algorithms. In *Proceedings of the Fourth International Conference on Knowledge Discovery in Databases & Data Mining*, pages 37–42. AAAI Press.
- Nakhaeizadeh, G. and Schnabl, A. (1998). Towards the personalization of algorithms evaluation in data mining. In Agrawal, R. and Stolorz, P., editors, *Proceedings of the Third International Conference on Knowledge Discovery & Data Mining*, pages 289–293. AAAI Press.
- Neave, H. R. and Worthington, P. L. (1992). *Distribution-Free Tests*. Routledge.
- Pavan, M. and Todeschini, R. (2004). New indices for analysing partial ranking diagrams. *Analytica Chimica Acta*, 515(1):167–181.
- Pfahring, B., Bensusan, H., and Giraud-Carrier, C. (2000). Meta-learning by landmarking various learning algorithms. In Langley, P., editor, *Proceedings of the 17th International Conference on Machine Learning, ICML’00*, pages 743–750.
- Pfisterer, F., van Rijn, J. N., Probst, P., Müller, A., and Bischl, B. (2018). Learning multiple defaults for machine learning algorithms. *arXiv preprint arXiv:1811.09409*.
- Pihur, V., Datta, S., and Datta, S. (2009). RankAggreg, an R package for weighted rank aggregation. *BMC Bioinformatics*, 10(1):62.

- Soares, C. and Brazdil, P. (2000). Zoomed ranking: Selection of classification algorithms based on relevant performance information. In Zighed, D. A., Komorowski, J., and Zytkow, J., editors, *Proceedings of the Fourth European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD 2000)*, pages 126–135. Springer.
- Todorovski, L. and Džeroski, S. (1999). Experiments in meta-level learning with ILP. In Rauch, J. and Zytkow, J., editors, *Proceedings of the Third European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD99)*, pages 98–106. Springer.
- van Rijn, J. N., Abdulrahman, S., Brazdil, P., and Vanschoren, J. (2015). Fast algorithm selection using learning curves. In *International Symposium on Intelligent Data Analysis XIV*, pages 298–309.
- Wistuba, M., Schilling, N., and Schmidt-Thieme, L. (2015). Sequential model-free hyperparameter tuning. In *2015 IEEE International Conference on Data Mining*, pages 1033–1038.



## Evaluating Recommendations of Metalearning/AutoML Systems

**Summary.** This chapter discusses some typical approaches that are commonly used to evaluate metalearning and AutoML systems. This helps us to establish whether we can trust the recommendations provided by a particular system, and also provides a way of comparing different competing approaches. As the performance of algorithms may vary substantially across different tasks, it is often necessary to normalize the performance values first to make comparisons meaningful. This chapter discusses some common normalization methods used. As often a given metalearning system outputs a sequence of algorithms to test, we can study how similar this sequence is from the ideal sequence. This can be determined by looking at a degree of correlation between the two sequences. This chapter provides more details on this issue. One common way of comparing systems is by considering the effect of selecting different algorithms (workflows) on base-level performance and determining how the performance evolves with time. If the ideal performance is known, it is possible to calculate the value of *performance loss*. The loss curve shows how the loss evolves with time or what its value is at the maximum available time (i.e., the time budget) given beforehand. This chapter also describes the methodology that is commonly used in comparisons involving several metalearning/AutoML systems with recourse to statistical tests.

### 3.1 Introduction

In this chapter we describe the methodology that can be used to assess the quality of recommendations generated by a given metalearning/AutoML system.

In many tasks it is necessary to compare performance of base-level algorithms across different tasks (datasets). As some tasks may be more difficult than others, the performance may differ substantially across tasks. Hence it may be useful to rescale the performance values so that these could be compared. Different rescaling techniques are discussed in Section 3.3.

The recommendations of typical metalearning/AutoML system can be seen as suggestions regarding which particular algorithm (workflow) should be applied to the target dataset. Quite often, the suggestions are in the form of an ordered sequence of algorithms (or workflows), which can be considered as the *recommended ranking* generated.

The strategy of recommending a ranked list of items can be compared to the strategy used in information retrieval, where normally a list of potentially useful documents is

suggested to the user. The reason for this is simple: the first item in the list may not be relevant, and hence it is better to give the user other options. But let us come back to the topic of evaluation, where the following issues arise:

1. Is the performance of the particular metalearning/AutoML system satisfactory from the user's point of view?
2. How does the performance of the particular metalearning/AutoML system compare with other systems of this kind (competitors?)

The aim of this chapter is to address both issues above. As for the first issue, there are two ways of assessing the quality of the recommended ranking of algorithms (workflows). One is by comparing the recommended ranking with a golden standard. This issue is taken up in Section 3.5. The second one aims to assess the effects when the recommendations are followed. More details about this are given in Section 3.6.

## 3.2 Methodology for Evaluating Base-Level Algorithms

The topic of evaluation of machine learning (ML) algorithms is discussed in various textbooks on machine learning (see, e.g., Mitchell (1997); Kohavi (1995); Schaffer (1993)), so it is not discussed in great detail here. However, we need to present some basic concepts that are required to explain the methodology for evaluating metalearning/AutoML systems.

### 3.2.1 Generalization error

One of the important concepts in ML is the concept of *generalization error*. ML algorithms are normally designed to minimize this error. In other words, when presented with some dataset, one should not try to fit this data perfectly, but rather try to generate a model that would perform well on yet unseen data points. In the following section, we will briefly review how this can be measured.

### 3.2.2 Evaluation strategies

The following evaluation strategies can be used to measure the generalization on unseen data points: *holdout*, *cross-validation*, or *leave-one-out* evaluation (Kohavi, 1995; Schaffer, 1993).

**Holdout.** When applying the holdout evaluation, the original dataset is split into two sets, a training set, consisting of, e.g., 90%, and a holdout set consisting of the remaining 10%. The model is trained on the training set, and evaluated on the holdout set. The division into two subsets is controlled by a parameter, and its setting affects the estimate of the performance.

**Cross-validation (CV)** is often used to overcome the problem of the holdout method, by evaluating the model several times. In so-called 10-fold cross-validation the given algorithm would be evaluated 10 times. In each fold, it would be trained on a different portion of the original dataset containing 10% of instances, while the remaining 90% would be used for testing. So each fold would result in certain performance measures. In classification, for instance, this would normally be *accuracy*, *precision*, *recall*, *AUC*, etc. The values of a particular metric are often aggregated. So, for instance, from 10 accuracy values we can obtain the mean value of accuracy.



Leave-one-out (LOO-CV) can be considered as a special case of cross validation procedure. Let us denote the size of the dataset by  $|d|$ . So in this case, the number of folds (cycles) is equal to  $|d|$ . It trains the model  $|d|$  times on a train set of size  $|d| - 1$ , and evaluates it on the final data point. This way each data instance is used exactly once for testing. This method is often used when there are relatively few examples for both training and testing.

**Bootstrap evaluation** This strategy is used when the number of cases is small. The augmented dataset is created by generating a certain number of so-called *bootstrap samples* from the original dataset, where each original data point can be sampled with replacement (meaning that it can occur multiple times in the bootstrap). If a bootstrap of the same size as the original dataset was generated, naturally some data points have been omitted, since others will occur multiple times. These data points will form the test set.

The machine learning community has adopted cross-validation as the standard for evaluating algorithm performance. However, for large datasets (e.g., image datasets) a single holdout set is often used instead. In this book, we often use the term cross-validation evaluation.

In Section 3.4 we explain how these measures are extended for evaluating metalearning and AutoML systems. In the next section we address the issue of normalization of performance values, which is needed when we want to carry out comparisons across different datasets.

### 3.2.3 Loss function and loss

In machine learning literature, the term *loss* is often used instead of *error*. Loss is a numeric value that is obtained by applying a *loss function* to a given algorithm  $a$  with a certain configuration of hyperparameters  $\theta$  and a given dataset  $d$ . The loss function can be defined as follows:

$$L = \mathcal{L}(a_{\theta}^j, d_{train}, d_{valid}), \quad (3.1)$$

where  $d_{train}$  is the training portion of  $d$  and  $d_{valid}$  is the validation set, that is, a subset of  $d$  used for evaluation purposes. This distinction between validation and test set is clarified further on in Section 3.4.1.

Let us see how it can be formulated to provide more details about the processes involved. Let  $M(a_{\theta}^j, d_{train})$  represent the output of the trained model generated by  $a_{\theta}^j$  on  $d_{train}$ , the training portion of a given dataset  $d$ . Let

$$\hat{y}_{valid} = A(M(a_{\theta}^j, d_{train}), X_{valid}) \quad (3.2)$$

represent the application of the trained model to  $X_{valid}$ , that is, a part of  $d_{valid}$  that includes just the attribute values. This function returns a prediction of the base-level performance  $\hat{y}_{valid}$ . Then the loss  $L$  can be determined by comparing the predictions  $\hat{y}_{valid}$  with using true values  $y_{valid}$ , which is the part of  $d_{valid}$  that includes just the target variable:

$$L = \mathcal{L}(\hat{y}_{valid}, y_{valid}). \quad (3.3)$$

Sometimes it is convenient to use the short form of the loss function, shown earlier (Eq. 3.1), that includes just the input arguments.

### 3.3 Normalization of Performance for Base-Level Algorithms

We note that the range of performance values for different datasets may vary substantially. An accuracy of 90% may be quite high on a classification problem, but low on another one. If we want to compare the performance of systems across different datasets, it is important to rescale the values. Different approaches were proposed in the past:

- Substituting performance values by ranks
- Rescaling into 0–1 interval
- Mapping values into a normal distribution
- Rescaling into quantile values
- Normalizing by considering error margin

More details about each transformation are given in the following subsections.

#### Substituting performance values by ranks

This transformation is described in Chapter 2 (Section 2.2.1).

#### Rescaling into 0–1 interval

This transformation requires that we identify the best (maximal) performance  $P_{max}^d$  and the worst (minimal) performance  $P_{min}^d$  of an algorithm on dataset  $d$ . Regarding the worst performance, it is common to use the performance of the default classifier, which simply predicts the most frequent class. These two values determine an interval between 0 and 1, and all values are rescaled into this interval. The following equation shows how a particular performance value  $P^d$  can be rescaled into  $P'^d$ :

$$P'^d = \frac{P^d - P_{min}^d}{P_{max}^d - P_{min}^d}. \quad (3.4)$$

Values of  $P'^d$  close to 0 (1) now indicate that the performance is close to the lowest (highest) value measured for this dataset.

#### Mapping values into a normal distribution

Another possibility is to use a standard normalization method, which requires that we calculate the *mean* and the *standard deviation* of all performance values. Then all success rates (or other performance values) are normalized by subtracting the mean value and by dividing the result by the standard deviation. The advantage of this method is that the values have a rather clear interpretation. Higher negative values (i.e.,  $< -0.5$ ) indicate that the success rate is rather low. Values around 0 show that the success rate is not far from the average. High positive values (i.e.,  $> 0.5$ ) indicate that the performance is rather good.

The disadvantage of this method is that it assumes that the values are distributed normally. This may not hold in some applications.

#### Rescaling into quantile values

This method transforms all values into quantiles, which are cut points that divide the range of a probability distribution into continuous intervals with equal probabilities.

## Normalizing by considering error margin

Gama and Brazdil (1995) suggested that all performance values be expressed in terms of the *error margin* (Mitchell, 1997), which is calculated as follows:  $EM = \text{sqrt}(ER \times (1 - ER)/NT)$ , where  $ER$  represents the error rate and  $NT$  the number of examples in the test set. The errors are converted to values indicating how many EMs it is below the best error rate, or alternatively above the worst error rate. The disadvantage of this method is that it assumes that the values are distributed normally.

## 3.4 Methodology for Evaluating Metalearning and AutoML Systems

In this section we describe the evaluation methodology that needs to be considered when evaluating metalearning and AutoML systems. We distinguish two modes: in one, the evaluation is carried out just once, following the strategy (protocol) of hold-out. More details can be found in the next subsection. The other mode involves a cycle following the cross-validation (CV) (or leave-one-out (LOO)) strategy. More details about this are given in Subsection 3.4.2.

### 3.4.1 One-pass evaluation with hold-out

This scheme follows the strategy described in Section 3.2, which relies on the division of the given dataset into a training subset and test set. This division is determined by the user setting up the experiment. It is outside the scope of a given metalearning/AutoML system. When comparing the performance of various metalearning/AutoML systems, this division should be constant in all comparisons. Many metalearning/AutoML systems carry out such evaluation internally to determine which base-level algorithm should be recommended. Let us examine this in more detail.

### Objective of metalearning/AutoML systems

In the Introduction we have described various metalearning/AutoML problems, including algorithm selection (AS), hyperparameter optimization (HPO) and combined algorithm selection and hyperparameter optimization (CASH). Here we will consider CASH, as it subsumes the other two.

The formal definition of the CASH problem exploits the concept of *loss* discussed in Section 3.2.3. The following definition is based on the work of Thornton et al. (2013):

$$a_{\theta^*}^* = \arg \min_{a^j \in \mathcal{A}, \theta \in \Theta^j} \mathcal{L}(a_{\theta}^j, d_{train}, d_{valid}). \quad (3.5)$$

As we see, the objective of metalearning systems is to minimize the loss by exploring different alternative algorithms and hyperparameter settings. In this process, they also carry out evaluation so that they would come up with the best recommendation. This issue is addressed in the next subsection.

## Inner evaluation carried out by metalearning/AutoML systems

Many metalearning/AutoML systems include an inner loop which includes evaluation: they try out some base model, evaluate this model on a set of unseen data cases, and repeat this process again.

As these systems do not (and must not) use the test set in any way, an additional set of cases is required (Varma and Simon, 2006). So, the training set is further divided into an *inner training set*, on which the base models are trained, and a *validation set*, which is used to assess the performance of different variants (e.g., with different hyperparameter settings) and select the best one. This division can be determined by a parameter of the metalearning system, and can even be optimized without our intervention. The division of the dataset is illustrated in Figure 3.1.

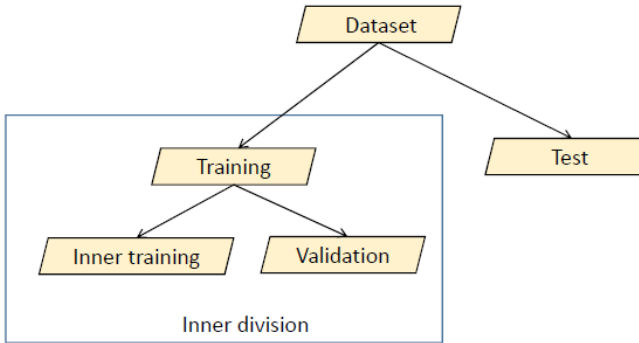


Fig. 3.1: Training, test, and validation set

The model that performs best on the validation set is selected by the metalearning/AutoML system. It is possible to retrain it on the whole training set (i.e., the inner training set plus the validation set), so that it would include as much data as possible to enhance its performance. Finally, the model is evaluated on the test set and its performance is reported to the user.

Note that the internal division can also be upgraded to include a cross-validation procedure in the inner evaluation concerned with selection. This has the advantage that different performance values obtained on different folds can be aggregated into a single value (e.g., mean) with lower dispersion. Consequently, the decisions made on the basis of this value tend to be more reliable. The disadvantage is that using CV takes more time than using hold-out, and hence the system takes more time to come up with a recommendation.

We note that this evaluation method provides a single result, as, in fact, it uses the methodology of hold-out on the meta-level. The use of cross-validation (CV) (or leave-one-out (LOO-CV), on the meta-level is discussed in the next subsection.

## Avoiding biased evaluation

When evaluating an AutoML system that includes metalearning capabilities, it is important to ensure that the dataset used for evaluation is not included in the meta-dataset accompanying the metalearning system, as this would lead to a biased estimate of performance. Some systems, such as Auto-sklearn, are shipped with a meta-dataset consisting of many common benchmark datasets. If the aim is to evaluate how this system performs on new datasets, obviously these should not be included in the meta-dataset accompanying the system.

However, if the aim is to simply use the metalearning system to solve practical problems, it does not matter if the new dataset belongs to the pool of existing meta-datasets, or if it is similar to the one in this pool. If this happens, one might expect that the system would have used its metalearning capability to do well in such situations.

### 3.4.2 Meta-level evaluation with cross-validation

The multiple pass evaluation with cross-validation requires that the base-level datasets are considered as *instances*. Then these can be divided into folds, as outlined in the description of the cross-validation (CV) strategy in Section 3.2. As the process is repeated for all folds, we obtain as many test results as there are folds. This information can be aggregated into aggregated values (e.g., averages of individual performance values).

We note that in many domains the meta-dataset can be relatively small and include a limited number of base-level datasets. Hence, the leave-one-out (LOO-CV) methodology is often adopted, as it is appropriate in such situations. As the case left out is, in this case, a base-level dataset, this strategy could be referred to as *leave-one-dataset-out*.

### Meta-level evaluation with table lookups

As evaluating base-level algorithms can be computationally expensive, it is common to use the following strategy: the performance of each algorithm on every dataset is recorded and stored in the corresponding meta-dataset. Whenever it is necessary to conduct the same evaluation again, previous results are retrieved using simply a table lookup.

The advantage of such an evaluation methodology is that it facilitates the task of conducting large-scale evaluation of metalearning systems. As the results of the experiments are pre-computed, the experiments can be carried out multiple times without computational overhead. Table lookups can be extended by surrogate models; an empirical performance model is used to predict the performance of configurations that were not explicitly examined. NAS-Bench-101 is a recent large-scale meta-dataset that can be used to retrieve experimental results.<sup>1</sup>

## 3.5 Evaluating Recommendations by Correlation

The quality of the recommended ranking of algorithms (workflows) is typically established through comparison with the *golden standard*, that is, the ideal ranking on the

<sup>1</sup>NASBench: A neural architecture search dataset and benchmark, <https://github.com/google-research/nasbench>

new (test) dataset(s). Sometimes, the ideal ranking is also referred to as the *true ranking*.

This evaluation protocol is applicable for evaluating metalearning systems that produce a ranking, such as the one presented in Chapter 2. In each leave-one-dataset-out cycle, the recommended ranking is compared against the ideal (true) ranking on the left-out dataset, and then the results are averaged for all cycles.

Different predicted rankings have different degrees of accuracy. For instance, given the golden standard (true ranking) is (1, 2, 3, 4), the ordering (2, 1, 3, 4) is intuitively a better prediction (i.e., is more accurate) than the ordering (4, 3, 2, 1). This is because the former ordering is more similar to the true ranking than the latter one.

Different measures can be used to evaluate how close (or how similar or how accurate) the recommended ranking is to the golden standard. This distance represents in effect a measure of *ranking accuracy*. A very common one is based on rank correlation used earlier (Sohn, 1999; Brazdil and Soares, 2000; Brazdil et al., 2003). Here we have opted for *Spearman's rank correlation* (Neave and Worthington, 1992), but Kendall's Tau correlation could have been used as well. Obviously, we want to obtain rankings that are highly correlated with the golden standard. This will enable us to assess the accuracy of a given ranking method.

Metalearning/AutoML method  $M_A$  will be considered *more accurate* than method  $M_B$  if it generates a ranked list of recommendations that is more similar to the true ranking than those obtained by method  $M_B$ .

## Spearman's rank correlation

The similarity between predicted and true rankings can be measured using Spearman's rank correlation coefficient (Spearman, 1904; Neave and Worthington, 1992) (see Eq. 3.6):

$$r_s(\hat{R}, R) = \frac{\sum_{i=1}^n (\hat{R}_i - \bar{\hat{R}}_i)(R_i - \bar{R}_i)}{\sqrt{(\sum_{i=1}^n (\hat{R}_i - \bar{\hat{R}}_i)^2 \sum_{i=1}^n (R_i - \bar{R}_i)^2)}}, \quad (3.6)$$

where  $\hat{R}_i$  represents the predicted ranks,  $\bar{\hat{R}}_i$  their mean value,  $R_i$  the true rank of item  $i$ , and  $n$  the number of items. In situations where there are no ties, the following formula (Eq. 3.7) can be used

$$r_s(\hat{R}, R) = 1 - \frac{6 \sum_{i=1}^n (\hat{R}_i - R_i)^2}{n^3 - n}. \quad (3.7)$$

An interesting property of Spearman's coefficient is that it is basically the sum of squared rank errors, which can be related to the normalized mean squared error measure, commonly used in regression (Torgo, 1999).

The sum is rescaled to yield more meaningful values: the value of 1 represents perfect agreement, and  $-1$  represents complete disagreement. A correlation of 0 means that the rankings are not related, which would be the expected score of a random ranking method.

The statistical significance of the values of  $r_s$  can be obtained from the corresponding table of critical values, which can be found in many textbooks on statistics (e.g., Neave and Worthington (1992)).

Table 3.1: Recommended and true ranking for the letter dataset

Algorithm	C5b	C5r	C5t	MLP	RBFN	LD	Lt	IB1	NB	RIP
Recommended	1.5	5.5	7	9	10	4	3	1.5	5.5	8
True	1	3	5	7	10	8	4	2	9	6

The use of Spearman’s rank correlation coefficient (Equation 3.6) to evaluate ranking accuracy is illustrated in Table 3.1.<sup>2</sup> The reader can verify that the value of Spearman’s correlation  $r_s$  is 0.707. According to the table of critical values of  $r_s$ , the value obtained is significant at a level of 2.5% (one-sided test).<sup>3</sup> Therefore, Spearman’s coefficient confirms that the recommended ranking is a good approximation to the true ranking.

### Weighted rank measure of correlation

Spearman’s rank correlation coefficient has the disadvantage that it treats all ranks equally. An alternative measure is a weighted rank measure of correlation which gives more importance to higher ranks than lower ones, following da Costa and Soares (2005) and da Costa (2015). This measure weighs the distance between two ranks using a linear function of those ranks:

$$r_w(\hat{R}, R) = 1 - \frac{6 \sum_{i=1}^n (\hat{R}_i - R_i)^2 ((n - \hat{R}_i + 1) + (n - R_i + 1))}{n^4 + n^3 - n^2 - n}. \quad (3.8)$$

The authors provide a table of critical values permitting to test whether a given value of the coefficient is significantly different from zero.

The weighted measure of correlation is useful in many practical applications including, besides algorithm recommendation, information retrieval, stock trading, and recommender systems. In all these systems the output is in the form of a ranking.

## 3.6 Evaluating the Effects of Recommendations

A disadvantage of using correlation to evaluate rankings is that it does not show directly what the user is gaining or losing when following the ranked list of recommendations. As such, many researchers have adopted an approach which simulates the sequential evaluation of algorithms on the new dataset, as the ranked list of recommended algorithms is followed.

### 3.6.1 Performance loss and loss curves

The measure that is used is the *performance loss*, defined as the difference in accuracy between  $\hat{a}^*$  and  $a^*$ , where  $\hat{a}^*$  represents the best algorithm identified by the system at a particular time and  $a^*$  the truly best algorithm that is known to us (Leite et al., 2012). Note that in many cases the truly best algorithm is not known to us. However, it is common to use some proxy, such as the algorithm found after some rather long search.

<sup>2</sup>The recommended ranking is the same as the one presented in Table 2.3 in Chapter 2.

<sup>3</sup>The significance level is 1 – confidence level.

Many loss curves used in the literature show how the loss depends on the number of tests carried out. An example of such a curve is shown in Figure 3.2.

As tests may take different times, it is useful to show how the loss depends on time. Let us use the term *loss-time curves* to refer to such curves. Figure 3.3 shows the result.

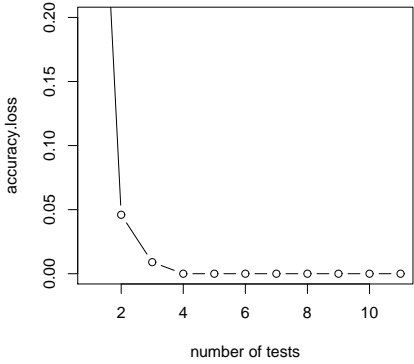


Fig. 3.2: Loss curves depending on the number of tests

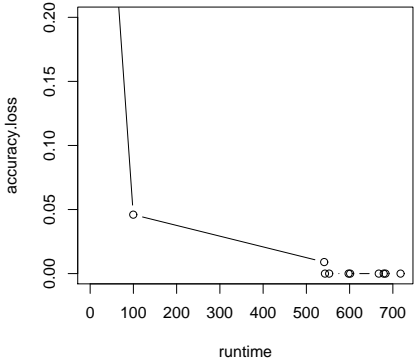


Fig. 3.3: Loss curves depending on runtime



### 3.6.2 Characterizing loss curves by AUC

Each loss–time curve can be characterized by a value representing the *mean loss* in a given interval, corresponding to the area under the loss curve. This characteristic is similar to the *area under the curve (AUC)*, but there is one important difference. When talking about AUCs, the  $x$ -axis values span between 0 and 1, while the loss curves span between some  $T_{min}$  and  $T_{max}$  defined by the user. Typically, the user searching for a suitable algorithm would not worry about very short times where the loss could still be rather high. In the experiments carried out by Abdulrahman et al. (2018),  $T_{min}$  was set to 10 seconds. In an on-line setting, however, we might need a much smaller value. The value of  $T_{max}$  was set to  $10^4$  seconds, corresponding to about 2.78 hours. We assume that most users would be willing to wait a few hours, but not days, for the answer. Also, many loss curves reach 0, or values very near 0, at this time. Note that this is an arbitrary setting representing a kind of default. Suitable values should be sought in accordance with the requirements of a particular domain.

### 3.6.3 Aggregating loss curves from multiple passes of CV

In Subsection 3.4.2 we discussed multiple-pass evaluation of metalearning/AutoML systems with recourse to either a CV or LOO-CV strategy. In each pass of the evaluation the system generates one loss/loss–time curve. It is thus useful to aggregate the individual loss into a *mean loss curve* in order to obtain an overall picture. An alternative to this would be to construct a *median loss curve*. Figure 3.4 shows the loss/loss–time curves of five different metalearning systems obtained on tests on 105 datasets. It is often useful

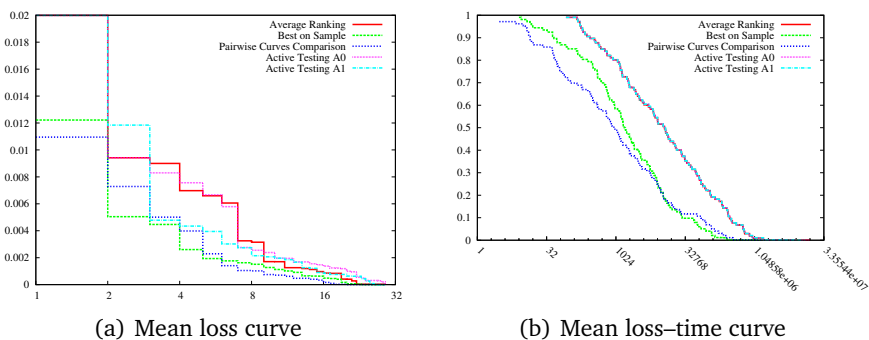


Fig. 3.4: Mean loss and loss–time curves of five metalearning systems obtained on 105 datasets. Image taken from van Rijn (2016)

to elaborate percentile bands (e.g., 25% and 75%), showing where the most frequent values lie.

### 3.6.4 Statistical tests at a given time budget

The need for statistical comparisons is motivated by the fact that showing that a metalearning/AutoML method generates more accurate predictions than another one on average does not provide a sufficiently convincing argument. The values used in the comparison are *estimates* of the corresponding true values obtained on a sample of datasets. These estimates, like the estimates of accuracy of algorithms in the learning tasks, have a certain variance, which may imply that the differences between two methods may not be statistically significant. Therefore, we need a methodology to assess the statistical significance of the differences between metalearning/AutoML methods.

Statistical tests can be applied whenever we have two (or more) series of numeric values capturing some aspect of the performance of two (or more) metalearning/AutoML methods. The numbers can be of two types. The first one involves some kind of correlation coefficients between the recommended ranking and true ranking (see Section 3.5). The second involves values that characterize two loss curves. This can be performance at a given time budget, or alternatively AUC values that characterize performance in a given interval of time. In situations when two or more methods have been used on multiple datasets, it is usual to follow the multiple comparison procedure described by Demšar (2006). It involves Friedman’s test, and if this test indicates that there are significant differences among the methods, it is possible to proceed with a post hoc test, which can be either Nemenyi test or Dunn’s test.

Systems can be ranked according to their performance at a given time budget. If we do not know beforehand the value of the budget, but know the time interval where the budget could lie, it is possible to carry out aggregation across all the values in this interval. One way to do this is to estimate the *area under the loss–time curve* in this interval. Figure 3.5 shows an example. All systems are ranked according to their performance, measured by the area under the loss–time curve in the given interval. It would of course be possible to create an alternative ranking relative to a specific time budget.

These ranks are represented in 1D linear space. If the distance between two ranks is greater than the so-called *critical distance*, the performance difference between the two systems is statistically significant. This critical distance depends on the number of algorithms and the number of datasets. Systems for which no statistical difference was found are connected by a thick black line.

In our example the test did not find a statistical difference between “Pairwise Curves Comparisons” and “Best on Sample”. Indeed, as Figure 3.4(b) already showed, the two loss–time curves are near one another, so one might be inclined to come to a similar conclusion by just looking at the earlier figure. On the other hand, the figure shows that there is statistical difference between “Pairwise Curves Comparisons” and the “Average Rank”. Note that these conclusions depend on the selected time budget, selected datasets, and, of course, the metalearning systems under scrutiny.

## 3.7 Some Useful Measures

### 3.7.1 Loose accuracy

The loose accuracy measure ( $LA@X$ ) uses the ranking accuracy of the top  $X$  elements in the ranked list (Kalousis, 2002; Sun and Pfahringer, 2013).  $LA@X$  returns the value of 1 if one of the  $X$  top predicted elements includes the top element in the correct ranking, and otherwise returns 0.  $LA@1$  is a special case and is sometimes referred to as *restricted accuracy*. It returns the value of 1 if the top prediction is correct, and 0 otherwise.

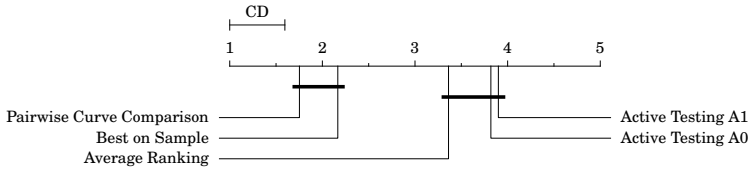


Fig. 3.5: A typical result obtained with the Nemenyi test. Image taken from van Rijn (2016)

### 3.7.2 Normalized discounted cumulative gain (DCG)

Discounted cumulative gain (DCG) has often been used to evaluate the effectiveness of search engines (Järvelin and Kekäläinen, 2002) and has been adopted in some work on algorithm selection (e.g., Sun and Pfahringer (2013)). This function uses a graded relevance scale in accordance with the corresponding position in the ranked list:

$$NDCG@X = DCG@X \times (IDCG@X)^{-1}, \quad (3.9)$$

where  $IDCG@X$  represents the ideal  $DCG$  at  $X$ . The term  $DCG@X$  is defined as

$$DCG@X = \sum_{i=1}^X \left( \frac{2^{g_i - 1}}{\log_2(i + 1)} \right), \quad (3.10)$$

where  $g_i$  is the value of the position in the ranked list.

## References

- Abdulrahman, S., Brazdil, P., van Rijn, J. N., and Vanschoren, J. (2018). Speeding up algorithm selection using average ranking and active testing by introducing runtime. *Machine Learning*, 107(1):79–108.
- Brazdil, P. and Soares, C. (2000). A comparison of ranking methods for classification algorithm selection. In de Mántaras, R. L. and Plaza, E., editors, *Machine Learning: Proceedings of the 11th European Conference on Machine Learning ECML 2000*, pages 63–74. Springer.
- Brazdil, P., Soares, C., and da Costa, J. P. (2003). Ranking learning algorithms: Using IBL and meta-learning on accuracy and time results. *Machine Learning*, 50(3):251–277.
- da Costa, J. P. (2015). *Rankings and Preferences: New Results in Weighted Correlation and Weighted Principal Component Analysis with Applications*. Springer.
- da Costa, J. P. and Soares, C. (2005). A weighted rank measure of correlation. *Aust. N.Z. J. Stat.*, 47(4):515–529.
- Demšar, J. (2006). Statistical comparisons of classifiers over multiple data sets. *The Journal of Machine Learning Research*, 7:1–30.
- Gama, J. and Brazdil, P. (1995). Characterization of classification algorithms. In Pinto-Ferreira, C. and Mamede, N. J., editors, *Progress in Artificial Intelligence, Proceedings of the Seventh Portuguese Conference on Artificial Intelligence*, pages 189–200. Springer-Verlag.

- Järvelin, K. and Kekäläinen, J. (2002). Cumulative gain-based evaluation of IR techniques. *IEEE Transactions on Information Systems*, 20(4).
- Kalousis, A. (2002). *Algorithm Selection via Meta-Learning*. PhD thesis, University of Geneva, Department of Computer Science.
- Kohavi, R. (1995). A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of Int. Joint Conference on Artificial Intelligence (IJCAI)*, volume 2, pages 1137–1145. Montreal, Canada.
- Leite, R., Brazdil, P., and Vanschoren, J. (2012). Selecting classification algorithms with active testing. In *Machine Learning and Data Mining in Pattern Recognition*, pages 117–131. Springer.
- Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill.
- Neave, H. R. and Worthington, P. L. (1992). *Distribution-Free Tests*. Routledge.
- Schaffer, C. (1993). Selecting a classification method by cross-validation. *Machine Learning*, 13(1):135–143.
- Sohn, S. Y. (1999). Meta analysis of classification algorithms for pattern recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21(11):1137–1144.
- Spearman, C. (1904). The proof and measurement of association between two things. *American Journal of Psychology*, 15:72–101.
- Sun, Q. and Pfahringer, B. (2013). Pairwise meta-rules for better meta-learning-based algorithm ranking. *Machine Learning*, 93(1):141–161.
- Thornton, C., Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2013). Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 847–855. ACM.
- Torgo, L. (1999). *Inductive Learning of Tree-Based Regression Models*. PhD thesis, Faculty of Sciences, Univ. of Porto.
- van Rijn, J. N. (2016). *Massively collaborative machine learning*. PhD thesis, Leiden University.
- Varma, S. and Simon, R. (2006). Bias in error estimation when using cross-validation for model selection. *BMC Bioinformatics*, 7(1):91.

## Dataset Characteristics (Metafeatures)

**Summary.** This chapter discusses dataset characteristics that play a crucial role in many metalearning systems. Typically, they help to restrict the search in a given configuration space. The basic characteristic of the target variable, for instance, determines the choice of the right approach. If it is numeric, it suggests that a suitable regression algorithm should be used, while if it is categorical, a classification algorithm should be used instead. This chapter provides an overview of different types of dataset characteristics, which are sometimes also referred to as metafeatures. These are of different types, and include so-called simple, statistical, information-theoretic, model-based, complexity-based, and performance-based metafeatures. The last group of characteristics has the advantage that it can be easily defined in any domain. These characteristics include, for instance, *sampling landmarks* representing the performance of particular algorithms on samples of data, *relative landmarks* capturing differences or ratios of performance values and providing *estimates of performance gains*. The final part of this chapter discusses the specific dataset characteristics used in different machine learning tasks, including classification, regression, time series, and clustering.

### 4.1 Introduction

One of the goals of metalearning is to relate the performance of learning algorithms to data characteristics, i.e., *metafeatures*. Therefore, it is necessary to identify which data characteristics are good predictors of the relative performance of algorithms and compute their values from the data. Using the framework of Rice (1976), these metafeatures can then be used to predict the performance of algorithms across datasets. This can be seen as a regression, classification, or ranking task (see Chapter 5, Sections 5.2 and 5.3).

#### What are good dataset features?

The development of metafeatures for metalearning should take the following issues into account:

**Discriminative power.** The set of metafeatures should contain information that distinguishes between the base-algorithms in terms of their performance. Therefore they should be carefully selected and represented in an adequate way.

Computational complexity. The metafeatures should not be too computationally complex. If this is not the case, the savings obtained by not executing all the candidate algorithms may not compensate for the cost of computing the measures used to characterize datasets. Pfahringer et al. (2000) argued that the computational complexity of metafeatures should be at most  $O(n \log n)$ .

Dimensionality. The number of metafeatures should not be too large compared with the amount of available metadata; otherwise overfitting may occur.

## Task- and data-specific characterization

The set of metafeatures suitable for different metalearning problems may vary substantially. The best set of metafeatures for a given metalearning problem depends essentially on the task, the datasets, and the algorithms. Although this book focuses on metalearning for the recommendation of algorithms in the machine learning domain, it can be applied to various other domains. Smith-Miles (2008) discusses how metalearning can be applied to sorting, forecasting, constraint satisfaction, and optimization. Cunha et al. (2018b) discuss how metalearning can be applied to recommender systems, and Costa et al. (2020) to imbalanced domains.

Within machine learning, the most common domains are classification, regression, time series forecasting, clustering and optimization, among others. In the following sections we provide more details on the data characteristics used in some of these domains.

## Characterization of algorithms

Most metalearning approaches focus on characterizing datasets. However, information about the algorithms may also be useful. For example, Hilario and Kalousis (2001) use information concerning *type of representation* (e.g., type of data they are able to deal with), *approach* (e.g., learning strategy, such as lazy or eager), *resilience* (e.g., sensitivity to irrelevant attributes, based on experimental studies), and *practicality* (e.g., easy parameter handling).

## Metafeature development

Developing useful metafeatures is an essential challenge for successful metalearning systems. In metalearning, similarly to any machine learning task, this challenge is mostly addressed using (meta)feature engineering approaches, which were not very systematic initially. Recently, there is a growing interest in more systematic approaches to metafeature development.<sup>1</sup> We discuss this issue in more detail in Section 4.6.

## 4.2 Data Characterization Used in Classification Tasks

### Types of Metafeatures

In this section we review the main types of features used in classification tasks. Usually, they are organized into different groups, depending on the type. Here we consider the following types:

<sup>1</sup>See e.g. <https://ieeexplore.ieee.org/abstract/document/8215494>  
<https://ieeexplore.ieee.org/document/7344858>  
<https://www.ijcai.org/Proceedings/2017/0352.pdf> .

1. Simple, statistical, and information-theoretic metafeatures;
2. Model-based metafeatures;
3. Performance-based metafeatures;
4. Concept and complexity metafeatures.

Each of these groups is discussed in more detail in the following subsections. Interested readers can also consult other sources which include an overview of the most common dataset features (see, e.g., Muñoz et al. (2018); Vanschoren (2019); Rivolli et al. (2019)).

### 4.2.1 Simple, statistical, and information-theoretic (SSI) metafeatures

Various features presented in this section represent data characteristics that are derived from dependent and/or independent variables of the given dataset.

#### Simple metafeatures

Typically, this set includes very simple descriptive measures, such as:

- Number of examples (instances),  $n$ ;
- Number of attributes (features),  $p$ ;
- Number of classes,  $c$ ;
- Proportion of discrete attributes;
- Proportion of missing values of feature  $x_i$ ;
- Proportion of outliers of feature  $x_i$ .

Some of these were used in the earliest metalearning approaches (e.g., Rendell et al. (1987); Aha (1992); Michie et al. (1994); Kalousis (2002)) and are still among the most commonly used metafeatures. The metafeature *number of classes* characterizes the complexity of the classification task. Some authors use different variants of some of the metafeatures shown above. For instance, instead of using number of examples,  $n$ , some researchers use  $\log(n)$ . Some ratios of two metafeatures seem rather useful:

- Number of examples per class  $n/c$ .
- Number of examples per dimension (feature)  $n/p$ .

Normally, we would want the value of the *number of examples per class* ( $n/c$ ) to be sufficiently high. It provides an estimate of data density. If the value is low, it indicates that the data is sparse, and consequently, the classification problem may be more difficult. Similarly, we would want the value of the *number of examples per dimension* ( $n/p$ ) to be high too. If it is low, it indicates that we have rather too many base-level features to choose from. Michie et al. (1994) referred to this situation as the *curse of dimensionality*.

Some metafeatures refer to a particular dataset feature (e.g., proportion of outliers of feature  $x_i$ ). Aggregation operations across different features are discussed in Section 4.6.

#### Statistical metafeatures

The most common approach to data characterization consists of the use of descriptive statistics, typically associated with numeric features.<sup>2</sup> Some metafeatures, such as the ones shown below, focus on a single independent feature ( $x_i$ ) or a class ( $y$ ).

<sup>2</sup>These features were used extensively in early works on metalearning (Michie et al., 1994; Brazdil et al., 1994; Brazdil and Henery, 1994; Gama and Brazdil, 1995; Todorovski and Džeroski, 1999; Lindner and Studer, 1999; Bensusan and Kalousis, 2001; Kalousis and Theoharis, 1999; Sohn, 1999; Vilalta, 1999; Köpf et al., 2000; Kalousis, 2002).

- Skewness of  $x_i$ ;
- Kurtosis of  $x_i$ ;
- Probability of class  $y$ .

Skewness and kurtosis characterize the shape of the underlying distribution (e.g., normality). Other metafeatures characterize the relationship between two or more independent features. These include, for instance:

- Correlation of  $x_i$  and  $x_j$ ,  $\rho(x_i, x_j)$ ;
- Covariance of  $x_i$  and  $x_j$ ;
- Concentration of  $x_i$  and  $x_j$ .

The first two were discussed by Michie et al. (1994), and concentration was discussed by Kalousis and Hilario (2001b). These measures provide an estimate of feature interdependence.

The metafeatures shown in this subsection (and other subsections too) can give rise to different derived metafeatures. For instance, it is possible to apply *aggregation operations* (e.g., mean, max) to derive new metafeatures, such as *mean correlation*, from individual values. Section 4.6 discusses the details of different operations that can be used to derive new features.

## Information-theoretic metafeatures

These metafeatures originated in information theory and are typically associated with nominal attributes. Some metafeatures apply to just one attribute or the class:

- Feature entropy of  $x_i$ ,  $H(x_i)$ ;
- Class entropy of  $y$ ,  $H(y)$ .

Class entropy provides an estimate of the difficulty of the classification task (Michie et al., 1994). It can also provide an estimate of class imbalance. Other metafeatures characterize the relationship between two or more independent features:

- Mutual information between  $x_i$  and  $y$ ,  $MI(x_i, y)$ .

Other metafeatures can be derived from the basic ones above (Michie et al., 1994):

- Intrinsic task dimensionality,  $\frac{H(y)}{MI(x_i, y)}$ ;
- Noise-signal ratio,  $\frac{H(y) - MI(x_i, y)}{MI(x_i, y)}$ .

### 4.2.2 Model-based metafeatures

In this approach a model is induced from the data and the metafeatures are based on their properties (Bensusan, 1998; Peng et al., 2002). The model used here depends on the type of task. When dealing with classification tasks, it is possible to use, for instance, a decision tree. This type of model would obviously not be inappropriate, if we were dealing with some other ML task (e.g., regression). The model must be related in some way to the candidate algorithms to provide metafeatures that are useful. Metafeatures obtained using this approach are only useful for algorithm recommendation if the induction of the model is sufficiently fast. Some examples of some basic tree-based metafeatures reflecting concept complexity are:

- Number of nodes;



- Number of leaves;
- Branch length.

Other metafeatures can be derived from the basic ones:

- Number of nodes per feature;
- Number of leaves per class;
- Leaves agreement.

Note that, while the SSI metafeatures discussed earlier are computed directly on the dataset, model-based metafeatures are obtained indirectly through a model.

### 4.2.3 Performance-based metafeatures

#### Landmarkers

Yet another approach to data characterization is the use of *landmarkers* (Bensusan and Giraud-Carrier, 2000; Pfahringer et al., 2000).<sup>3</sup> Landmarkers are quick estimates of algorithm performance on a given dataset. They can be obtained by running simplified versions of the algorithms.<sup>4</sup> For instance, a decision stump, i.e., the root node of a decision tree, can be the landmarker for decision trees. The following landmarkers were suggested by Pfahringer et al. (2000):

- 1NN, characterizing data sparsity;
- Decision tree (or decision stump), characterizing data separability;
- Linear discriminant, characterizing linear separability;
- Naive Bayes, characterizing feature independence.

Like model-based metafeatures, landmarkers characterize the dataset indirectly. But they go one step further, by representing the performance of a model on some dataset rather than representing properties of the model.

Several studies report on comparisons of some of the approaches for data characterization discussed here (e.g., Bensusan and Kalousis (2001); Köpf and Iglezakis (2002); Todorovski et al. (2002)).

#### Relative landmarkers

Relative landmarkers can also be used to characterize datasets. As in the previous case, the characterization is indirect. Relative landmarkers are based on a difference (or a ratio) of the performance of two algorithms. Relative landmarkers were used for *probing the performance* of a particular algorithm  $a$ , as its performance can be compared with the performance of other algorithms (Fürnkranz and Petrak, 2001; Soares et al., 2001). Furthermore, Leite et al. (2012) used relative landmarkers in the so-called *active testing* method, discussed in Chapter 5. Finally, Post et al. (2016) used relative landmarkers to determine whether feature selection should be applied for a given algorithm and dataset combination.

---

<sup>3</sup>The concept of landmarkers can be related to earlier work on yardsticks (Brazdil et al., 1994).

<sup>4</sup>Chapter 3 explains how such estimates of performance can be obtained.

## Subsampling landmarks and partial learning curves

An alternative way of obtaining quick performance estimates is to run the algorithms whose performance we wish to estimate on a sample of the data, obtaining the so-called *subsampling landmarks* (Fürnkranz and Petrak, 2001; Soares et al., 2001; Leite and Brazdil, 2004).

A more informative characteristic is obtained by considering an ordered sequence of subsampling landmarks for a single algorithm, representing, in effect, a part of its learning curve (Leite and Brazdil, 2005). In this case, metalearning can take into account not only the values of the estimates, but also the shape of the curve.

As with the previous two cases, subsampling landmarks also characterize the dataset indirectly. If the performance of the subsampling landmarks were, in fact, related to the performance of the base-algorithms, one can expect this approach to be more successful than the previous ones. Experimental results exist to support this (Leite and Brazdil, 2007; van Rijn et al., 2015).

## Multiple performance landmarks

As we have pointed out in one of the previous subsections, a *landmarker* represents the performance of a particular algorithm on a particular dataset. There is no reason why we could not associate more than one landmarker with a particular dataset and represent them in the form of a vector.

### 4.2.4 Concept and complexity-based metafeatures

In this section we discuss a group of measures that characterize the complexity of the supervised classification task (Rendell and Seshu, 1990; Ho and Basu, 2002). Some of these measures can serve as useful metafeatures. Here we consider the following types of measures:

- Concept variation/roughness in output space;
- Overlap of individual features;
- Separability of classes.

More details about each type are given in the following subsections. Most of the metafeatures were discussed by Ho and Basu (2002), unless otherwise stated. Smith et al. (2014) use similar features, but characterize the complexity of specific instances, rather than the full dataset.

#### Concept variation/roughness in output space

Concept variation (Rendell and Seshu, 1990; Perez and Rendell, 1996) captures the roughness of the target concept in instance space. Irregularity in the output space occurs when neighboring examples in the input space have different labels. The measure  $\delta(e_i, e_j)$  is 0 if two neighboring examples  $e_i$  and  $e_j$  belong to the same class, and 1 otherwise. The pairs of examples used differed only in one feature. The values across different pairs were then averaged to obtain the final value.

**Nonlinearity of linear classifier:** This measure is sensitive to the smoothness of the classifier's decision boundary, and so the aim is similar to the *concept variation* discussed earlier. The aim is to slightly alter the input points (examples), use these points as test points, and investigate the effects on the error rate of the linear classifier. The new test points are generated by repeatedly picking two points (examples) of the same class and performing a *linear interpolation* (with random coefficients) on the corresponding feature values. The classifier trained on the original training set is applied to this new test set, and its error represents this measure.

**Nonlinearity of 1NN classifier:** This measure is obtained in a similar way to the *non-linearity of linear classifier*. The new test generated in the way described above is applied to the 1NN classifier trained on the original training set. The error of this classifier represents this measure.

### Overlap of individual features

**Fisher's discriminant ratio:** This is calculated as  $\frac{\mu_1 - \mu_2}{\sigma_1 - \sigma_2}$ , where  $\mu_1$  and  $\sigma_1$  represent the mean and standard deviation of feature values associated with class 1. Similarly,  $\mu_2$  and  $\sigma_2$  are associated with class 2.

**Volume overlap region:** It is possible to determine a region delimited by the maximum and minimum values of some feature associated with class 1. This can be repeated also for class 2. Finally, it is possible to calculate the overlap region.

**Feature efficiency:** The aim is to characterize how much each feature contributes towards the separation of the two classes. If some feature values can lead to both classes, the classes are *ambiguous* in that region of values. It is possible to eliminate ambiguity progressively. In each pass, the features can be ordered by how many points are in the nonoverlapping region. The *efficiency* of each feature is defined as the fraction of all remaining points separable by that feature.

More details on the above features can be found in the article by Ho and Basu (2002).

### Separability of classes

Ho and Basu (2002) proposed two groups of measures. The first one characterizes linear separability and the second whether the two sets of points (examples) come from two different distributions. Below we present just one feature from each group. Both metafeatures provide an estimate about how hard a given classification problem is.

**Linear separability:** This approach presupposes the application of a linear classifier. One metafeature is defined as the error rate of the linear classifier.

**Fraction of points on the class boundary:** The aim is to determine whether two samples (of class 1 and 2) come from the same distribution. The method uses the concept of *minimum spanning tree (MST)* to do this. The MST connects points (data examples) regardless of the class. Then the number of points connected to the opposite class represent the *points on the class boundary*. Figure 4.1 shows an illustrative example. The fraction of such points is used as one of the measures.

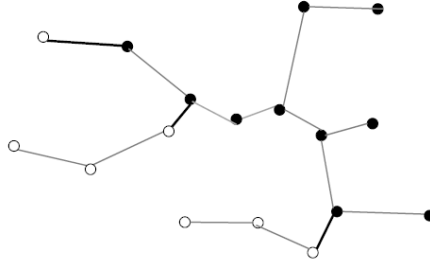


Fig. 4.1: A minimum spanning tree connecting points of two classes. The thicker edges connect two classes. Reproduced from Ho and Basu (2002)

### Relationship of some complexity measures to other types

It is interesting to note that some measures discussed in this subsection presuppose a usage of a certain model type (linear classifier, NN classifier) and the measures are derived from their application. One can compare this to the model-based features discussed in Subsection 4.2.2. Also, as some features are represented by the error rates of a particular classifier (linear classifier, NN classifier), this approach could be compared to the landmarker approach discussed in Subsection 4.2.3.

## 4.3 Data Characterization Used in Regression Tasks

Various researchers have studied the application of metalearning approaches to regression tasks and consequently also discussed the metafeatures used (Soares et al., 2004; Lorena et al., 2018). The metafeatures can be divided into the following major groups:

- Simple and statistical metafeatures;
- Complexity-based metafeatures;
- Smoothness metafeatures.

Here we follow rather closely the exposition presented by Lorena et al. (2018), unless stated otherwise.

### 4.3.1 Simple and statistical metafeatures

These metafeatures are not too different from the ones discussed in Subsection 4.2.1. Many of these features can be reused, and for this reason they will not be included here. However, as the target variable is numeric, all features that involve the target variable need to be altered. Some features that characterize just one variable are:

- Coefficient of variation of the target variable,  $\frac{\sigma(y)}{\mu(y)}$ ;
- Number of outliers of the target variable  $y$ .

The coefficient of variation of the target variable is calculated as the ratio of the standard deviation  $\sigma(y)$  and the mean  $\mu(y)$  of the target variable (Soares and Brazdil, 2006; Soares, 2004). Some metafeatures capture the relationship between two variables:

- Data density,  $n/p$ .

The concept of data density is similar to the concept used in classification tasks (see Subsection 4.2.1). It is calculated as the ratio of the number of examples and features.

### Correlation-based metafeatures

- Correlation between feature  $x_i$  and target  $y$ ,  $\rho(x_i, y)$ ;
- Correlation between feature  $x_i$  and feature  $x_j$ ,  $\rho(x_i, x_j)$ .

Other features can be derived from the basic set using various operations, including, e.g., aggregation operations (average, etc.), described in Section 4.6. Two metafeatures seem particularly important (Lorena et al., 2018):

- Maximum correlation between features and the target,  $\rho_{max}$ ;
- Average correlation between features and the target variable,  $\bar{\rho}$ .

A high value of  $\rho_{max}$  indicates that it may be possible to obtain good predictions of the target using this feature alone.

### 4.3.2 Complexity-based measures

**Maximum individual feature efficiency:** This measure can be seen as an adaptation of feature efficiency defined for classification tasks to regression. The concept of *high effect of feature on separability of classes* is substituted by *high effect of feature on correlation to target*. For each feature  $x_i$  the method identifies the smallest number of examples that must be removed until a high correlation ( $< 0.9$ ) between feature  $x_i$  and the target variable is obtained. The numbers of examples removed are then converted to proportions. Finally, this measure is equal to the minimum proportion identified across all features. Small values indicate relatively easy problems.

**Collective feature efficiency:** This involves an iterative process of identifying the feature with the highest correlation, carrying out a linear fit, and eliminating examples with small residual value. This measure corresponds to the proportion of examples that remain after all features have been examined. Small values indicate relatively easy problems and higher values more complex ones.

### 4.3.3 Complexity/model-based measures

Lorena et al. (2018) include the following two features among the complexity-based measures. However, as they are derived from a particular model (linear regressor), we could regard them also as model-based features.

**Mean absolute value of linear regressor:** This measure averages the residuals of a multiple linear regression. Small values indicate simpler problems.

**Variance of residuals of linear regressor:** This measure averages the squares of residuals of a multiple linear regression. Small values indicate simpler problems.

#### 4.3.4 Smoothness measures

Similarity of target values for similar examples:<sup>5</sup> This metafeature has a similar aim to *concept variation* in that it tries to estimate the smoothness/roughness of similar examples in the output space (see Subsection 4.2.4). The method borrows the idea of *minimum spanning tree* (MST) used in the definition of *fraction of points on the boundary*. The MST joins the most similar examples in the input (feature) space, while the edges are weighted by the Euclidean distance. This measure then captures the mean distance between the target values. Lower values indicate simpler problems.

Similarity of features for examples with similar targets:<sup>6</sup> This measure complements the measure above. It measures how similar the inputs (features) are for pairs of examples with similar target values.

Errors of 1NN regressor: This metafeature is an adaptation of a similar metafeature, namely the 1NN landmarker, defined for classification tasks. A suitable error measure, such as *mean squared error* (MSE), needs to be used here.

#### 4.3.5 Nonlinearity measures

Nonlinearity of linear regressor: This metafeature is an adaptation of a similar metafeature defined for classification tasks. First, two examples with similar outputs are selected and both input (feature) and output values are interpolated to generate a new test data item. This step is repeated. The linear regressor is trained on the original data and applied to the new test set. The *mean squared error* (MSE) obtained is used as the metafeature. Lower values indicated simpler problems.

Nonlinearity of 1NN regressor: This metafeature is an adaptation of a similar metafeature defined for classification tasks. Instead of a linear regressor, the 1NN regressor is used.

### 4.4 Data Characterization Used in Time Series Tasks

The issue of how to apply metalearning to time series tasks was investigated by various researchers in the past (see, e.g., Adya et al. (2001); Prudêncio and Ludermir (2004); dos Santos et al. (2004); Lemke and Gabrys (2010), etc.). Characterization of time series data needs to take into account the fact that a time series is an ordered set of values.

Lemke and Gabrys (2010) divided the features into four groups: general statistics, frequency domain characteristics, autocorrelation characteristics and diversity measures. Further details about the first three groups are given in the following subsections. The last group that involves diversity measures is useful in the construction of ensembles. More details about this issue can be found in Chapter 10.

#### General statistics (descriptive statistics)

To calculate the descriptive statistics of the time series, Lemke and Gabrys (2010) first detrended it using polynomial regression. Some of the characteristics used are shown below:

<sup>5</sup>Lorena et al. (2018) call this measure the *output distribution*.

<sup>6</sup>Lorena et al. (2018) call this measure the *input distribution*.

- Length of time series;
- Standard deviation (std) of detrended series;
- Skewness and kurtosis;
- Trend, calculated as  $\text{std}(\text{series})/\text{std}(\text{detrended series})$ ;
- Number of turning points;
- Number of step changes;
- Estimate of nonlinearity;

The estimate of nonlinearity is obtained by generating a surrogate linear time series and comparing it with the original one.

### Frequency-domain characteristics

Frequency-based features can be derived from the power spectrum which, in turn, is obtained by applying fast Fourier transform to the time series data. Lemke and Gabrys (2010), for instance, used the following features:

- Frequencies of three largest values;
- Maximal value indicating the strongest seasonal or cyclic component;
- Number of peaks that have at least 60% of the maximal component.

### Autocorrelation-based characteristics

These features provide information about the stationarity and seasonability of time series. Autocorrelation and partial autocorrelation (Box and Jenkins, 2008) provide important information about the properties of a time series (Chatfield, 2003). These values are calculated with respect to data points that include a *lag* by  $d$  positions. Lemke and Gabrys (2010) used:

- Autocorrelation at lags 1 and 2;
- Partial autocorrelation at lags 1 and 2;
- Partial autocorrelation at lag 7 (or 12) capturing weakly (or monthly) seasonality.

Other metafeatures can be derived from the basic characteristics in a similar way as discussed earlier. Some examples include the *mean absolute value of the first five autocorrelations* (i.e., with  $d \in \{1, \dots, 5\}$ ) or the *statistical significance of the first autocorrelation coefficients* (Prudêncio and Ludermit, 2004; dos Santos et al., 2004).

## 4.5 Data Characterization Used in Clustering Tasks

In this section we analyze various metafeatures that can be used in clustering. This problem was addressed by various researchers before (de Souto et al., 2008; Soares et al., 2009; Ferrari and de Castro, 2015; Pimentel and de Carvalho, 2019).

This area represents a challenge, as it belongs to a group of algorithms referred to as unsupervised learning. As these tasks do not contain a target variable, fewer descriptive characteristics are available to describe the data. In the following we describe different techniques that can be used to respond to this challenge.

In Section 4.2 we presented the main types of metafeatures that tend to be used in classification tasks. They were divided into four major groups. So a question arises regarding whether each group can be adapted to clustering tasks and, if so, how. The following subsections provide details on this topic.

## Simple, statistical, and information-theoretic metafeatures

As the data does not include the target variable, it is possible to use only the features that involve independent variables. Ferrari and de Castro (2015), for instance, used an appropriate subset of metafeatures, similar to those shown in Subsection 4.2.1. Pimentel and de Carvalho (2019) proposed metafeatures describing the distribution of rank correlation between examples (not features).

### Model-based metafeatures

Interestingly, Ferrari and de Castro (2015) adapted this idea to the task of clustering. The authors defined a vector  $\mathbf{d}$  containing pairwise Euclidean distances  $d_{i,j}$  between all pairs of objects (data instances)  $i$  and  $j$  of a given dataset. The vector is then normalized into an interval  $[0, 1]$  and characterized using statistical measures. These can be divided into three groups discussed next.

The first subgroup includes some simple measures, such as mean, variance, standard deviation, skewness, and kurtosis. All these characterize the distribution of values in  $\mathbf{d}$ .

The second subgroup of metafeatures characterizes a histogram constructed on the basis of distribution of the values in  $\mathbf{d}$ , following the approach of Kalousis (2002). The authors used 10 bins (intervals) of equal size. The feature corresponding to bin  $j$  includes the percentage of values contained in this bin.

The third subgroup of metafeatures provides an alternative way of characterizing the distribution. The authors have first generated  $z$ -scores defined by  $z = \frac{x-\mu}{\sigma}$ , where  $\mu$  represents the mean and  $\sigma$  standard deviation. The absolute values of  $z$  scores were discretized into four bins:  $[0, 1)$ ,  $[1, 2)$ ,  $[2, 3)$ , and  $[3, \infty)$ . The corresponding metafeatures captured the proportion of cases in each bin.

### Performance-based metafeatures

Some authors have used *internal validation measures* in the meta-learning framework for clustering (Vukicevic et al., 2016; Tomp et al., 2019).

It would seem that various measures that were used in classification tasks, such as landmarks and subsampling landmarks, could be adapted to this domain.

### Metalearning vs. optimization on target dataset

This domain provides, however, a challenge to metalearning approaches. It may be difficult to provide a good recommendation of clustering algorithms (or their configurations) just by looking at the data. This is because many approaches do not cluster the points in their original space, but rather use dimensionality reduction first. So, the alternative is to carry out a search for the best solution on the target concept.

## 4.6 Deriving New Features from the Basic Set

### Generating new features by aggregation

We note that some features, such as *skewness*, can be calculated for each numeric attribute. Given that the number of attributes varies for different datasets, this implies



that the number of values describing *skewness* for different datasets varies. This creates a problem for metalearning systems that use propositional representation.

The most common approach to solve this problem is to do some form of aggregation, for instance by calculating *mean skewness*. However, it should be expected that important information may be lost by this aggregation. Alternatively, Kalousis and Theoharis (1999) used a finer-grained aggregation, where histograms with a fixed number of bins were used to construct new metafeatures. For instance, the distribution of skewness values could be represented with three metafeatures corresponding to the number of attributes with skewness smaller than 0.2, between 0.2 and 0.4, and larger than 0.4.

### Generating a complete set of metafeatures

Some researchers (Pinto et al., 2016; Pinto, 2018) have observed that many systems use a set of dataset features that can be considered incomplete. For instance, *entropy* is commonly applied to the target variable, but not to dataset features. Aggregation operations often involve calculating, for instance, the mean value of all numeric features. Different aggregation operations (see, e.g., Tukey (1977)) are listed below:

- mean value ( $\mu$ )
- standard deviation ( $\sigma$ )
- minimum value (min)
- maximum value (max)
- first quartile (q1)
- median value (q2)
- third quartile (q3)

These are often not used. So the authors have proposed to generate a complete set of features, thus enriching the initial set that was considered. Pinto (2018) has shown that a metalearning system that uses the complete set achieves better performance than the initial set. Although feature selection can be used to reduce this set, it was shown that it could degrade performance.

### Generating new features by PCA

Principal component analysis can be used to project features into a low-dimensional space that includes the principal components. This method was used, for instance, by Smith-Miles et al. (2014). However, the PCA model was somewhat unsatisfactory to predict performance, since PCA is only concerned with maximizing the variance explained by the features.

### Transforming features by feature selection and projection

The method used by Smith-Miles et al. (2014), which included 235 dataset instances, used two steps. In the first one, feature selection was used to reduce a relatively large set of features (509) to ten features. In the second step, the ten-dimensional space was projected onto a two-dimensional space. The projection was defined as an optimization problem, where the aim was to minimize the *approximation error*, defined in terms of both true and predicted values of the feature data matrix and performance vector. The projection revealed regions in the 2D space, referred to as a *footprint*, where a particular algorithm is expected to do well. More details about this study can be found in Subsection 4.7.1.

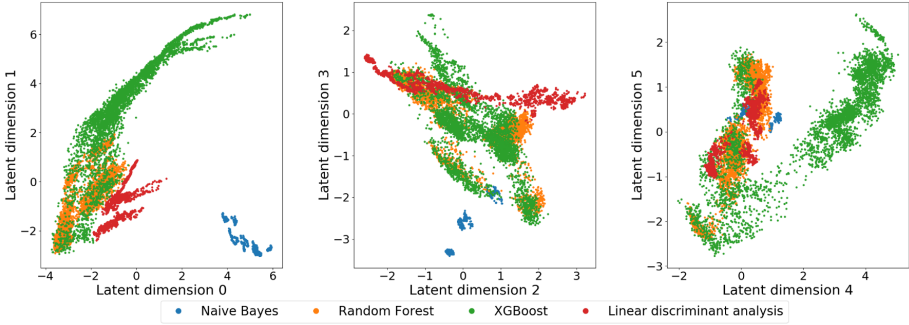


Fig. 4.2: Latent embedding based on probabilistic matrix decomposition of 42,000 configurations, color coded by the algorithm. Image taken from Fusi et al. (2018)

### Constructing new latent features by matrix factorization

Fusi et al. (2018) proposed to apply matrix factorization to the performance matrix  $Y \in \mathbb{R}^{N \times D}$ , where  $N$  is the number of algorithms (workflows) and  $D$  is the number of datasets. Each cell contains the performance of a particular algorithm on a particular dataset. The authors proposed to use a probabilistic matrix factorization algorithm that decomposes  $Y \approx XW$ , where  $X \in \mathbb{R}^{N \times Q}$  and  $W \in \mathbb{R}^{Q \times D}$ .

The authors note that, in many cases,  $Y$  is a sparse matrix and this method helps to provide a solution. Being able to deal with missing values is important in situations when a repeated algorithm selection method is carried out for the new (target) dataset. An advantage of the probabilistic matrix factorization method is that it maps each dataset into a latent feature vector of size  $Q$ .

This opens up a way to new lines of research, where the resulting matrix can be mined for patterns, such as shown in Figure 4.2. The plot shows various results on several datasets relative to four different algorithms (more precisely OpenML flows) and their differently configured variants. The algorithms involved in this study were naive Bayes, random forest, XGBoost, and linear discriminant analysis (LDA).

Yang et al. (2019) describe an AutoML system that is integrated with an algorithm selection method with similar latent features.

### Generating new features in the form of embeddings

Some researchers have suggested to use a so-called Siamese neural network (SNN) to generate a feature vector from a given dataset (Baldi and Chauvin, 1993; Bromley et al., 1994).<sup>7</sup> This network consists of two similar sub-networks. During training each sub-network is applied to similar examples, such as, examples from one class. This permits to extract a feature vector consisting of neural weights, representing effectively an embedding. Classification consists of comparing an extracted feature vector of each example

<sup>7</sup>Chapter 13 discusses Siamese neural networks.

with a stored feature vector for each class. Items closer to this stored representation for the positive class than a chosen threshold are accepted as belonging to the positive class.

This approach was originally used to distinguish original signatures from forgeries. In subsequent works it was reused and adapted to various other domains, including, e.g., speaker recognition (Chen and Salman, 2011) and sentence similarity (Mueller and Thyagarajan, 2016).

In another work on recommender systems which involves a metalearning approach, Cunha et al. (2018a) used graph embeddings to create *dataset embeddings*.

## 4.7 Selection of Metafeatures

### 4.7.1 Static selection of metafeatures

It is often important to select a suitable subset of data characteristics and the corresponding metafeatures from all the possible alternatives. The number of metafeatures should not be too large compared with the amount of available metadata. An excessively large number of measures may cause overfitting and, thus, poor predictions on unseen data. This is particularly true because the number of examples in metalearning (i.e., datasets) tends to be small.

Selection of metafeatures may be done during the development of the metalearning system by including only measures that are expected to be relevant (Brazdil et al., 2003). This can be done by taking into account the characteristics of the metalearning problem, as discussed above.

Alternatively, it is possible to include as many metafeatures as possible. A feature selection method can then be applied to obtain a smaller subset of suitable metafeatures. Obviously, if it were possible to use a relatively small subset, this would have advantages. The whole process of metalearning would be simpler, as one would not have to calculate so many characteristics.

It has been shown that the use of wrapper-based feature selection methods at the meta level can improve the quality of the results (Todorovski et al., 2000; Kalousis and Hilario, 2001a). The improvement can be attributed to the fact that “noisy” attributes have been dropped. The wrapper-based approach normally uses *backward elimination*, which is normally used for feature selection (Kuhn and Johnson, 2013).

Recently, Muñoz et al. (2018) carried out a comprehensive study in the area of classification with 509 features. The aim was to select a small subset that would characterize well the hardness of the classification task. The level of hardness was established by measures, such as *nonlinear separability*, among others. The whole process of identifying the relevant features is quite complex, so the authors have identified the following ten features:

- Maximum normalized entropy of the attributes (information-theoretic)
- Normalized entropy of class attribute (information-theoretic)
- Mean mutual information of attributes and class (information-theoretic)
- Error rate of the decision node (landmarker)
- Training error of linear classifier (landmarker)
- Standard deviation of the weighted distance (concept characterization)
- Maximum feature efficiency (complexity)
- Collective feature efficiency (complexity)
- Fraction of points on the class boundary (complexity)

- Nonlinearity of nearest-neighbor classifier (complexity)

The type of metafeature is mentioned as well. We note that this set includes representatives of several metafeature types.

## 4.7.2 Dynamic (iterative) data characterization

In the previous subsection we described the process of selecting metafeatures prior to their use by a metalearning system. An alternative approach consists of gathering the metafeatures in an iterative fashion (Leite and Brazdil, 2005, 2007). This approach is useful in situations when gathering the metafeatures incurs costs. We may not want to use the most informative set from the start simply to save effort.

Suppose the aim is to determine whether algorithm A or algorithm B should be used with the target dataset. A test of both algorithms on a small sample (i.e., on the basis of a given *subsampling landmarker*) provides some information that can be used for this decision. Obviously, if we use more samples, we obtain more information. But if we can make a decision on the basis of the existing metafeatures, there is no need to extend it further.

In each phase of the algorithm described by Leite and Brazdil (2005, 2007), the system tries to determine whether the currently available set of metafeatures is adequate or whether it should be extended, and if so, how. This is done with the help of existing metadata. The aim is to determine what happened in similar circumstances in the past. If there is evidence that some extensions lead to a marked improvement of performance, the system tries to identify the best one. This is the one which is expected to provide maximum information while requiring the least computational effort.

We note that characterization of datasets is built up gradually. In each step, the system determines the next sample sizes that should be tried out. The plan of these experiments is built up gradually, by taking into account the results of all previous experiments, both on other datasets (past metadata) and partly also on the target dataset (new metadata).

## 4.8 Algorithm-Specific Characterization and Representation Issues

### 4.8.1 Algorithm-specific data characterization

The set of base-algorithms should also be taken into account in the development of metafeatures. In the case where diverse algorithms are included, different sets of metafeatures could be useful for discriminating the performance of different pairs of algorithms (Aha, 1992; Kalousis and Hilario, 2001a, 2000; Sun and Pfahringer, 2013).

For instance, the *proportion of continuous features* can be useful to discriminate between naive Bayes and  $k$ -NN, but not between naive Bayes and a rule-based learner (Kalousis and Hilario, 2000). This is consistent with the knowledge that  $k$ -NN is better suited for continuous features than naive Bayes, but both the naive Bayes and rule-based systems have problems to deal with this kind of attributes. Therefore, a set of metafeatures that is able to discriminate among all of the algorithms should be used.

## Data characterization useful for ranking pairs of algorithms

Another approach is to transform the problem into several pairwise metalearning problems (i.e., predict whether to use algorithm A or B, or whether they are equivalent) and use different sets of metafeatures for each of them. This strategy has been used, for instance, by Sun and Pfahringer (2013). The existing meta-data is used to train a rule-based classifier whose aim is to predict whether algorithm A (or B) is better for a particular dataset. The rules include base-level features which may include landmarks (e.g., AUC associated with a particular type of tree (REPTree.depth2)).

When the base-algorithms are similar, specific metafeatures that represent the differences between them should be designed. A particular case is when the base-algorithms represent the same algorithm with different parameter settings. In the case of selecting parameters for the kernel of SVM, it has been shown that better results are obtained with algorithm-specific metafeatures than with general ones (Soares and Brazdil, 2006). The metafeatures used in this work were based on the kernel matrices for the different kernel parameters considered. In a different approach to this problem, metafeatures characterizing the kernel matrix were combined with other metafeatures describing the data in terms of its relation to the margin (Tsuda et al., 2001).

### 4.8.2 Representation Issues

Most researchers represent the metafeatures using a vector with a fixed number of positions. However, some approaches have exploited a *relational representation* of metafeatures (Todorovski and Džeroski, 1999; Hilario and Kalousis, 2001; Kalousis and Hilario, 2003), commonly used in inductive logic programming (ILP). For instance, in a dataset with  $k_c$  continuous attributes, skewness is described by  $k_c$  metafeatures, with the skewness value of each attribute. An ILP approach has also been proposed to take full advantage of the model-based approach to data characterization, which is also non-propositional (Bensusan et al., 2000). The authors illustrate their proposal by characterizing the dataset using a decision tree induced from that dataset.

## 4.9 Establishing Similarity Between Datasets

### 4.9.1 Similarity based on metafeatures

Let us represent the metafeatures of some meta-instance (dataset)  $d_i$  using a vector  $\mathbf{f}_{d_i} = (f_{d_i,1}, f_{d_i,2}, \dots, f_{d_i,m})$ , where  $m$  is the number of metafeatures. Similarly,  $\mathbf{f}_{d_{new}}$  represents the vector of metafeatures of the target dataset  $d_{new}$ . The vectors of metafeatures are used to identify the datasets that are most similar to the target dataset. This is done using an approach similar to  $k$ -NN. The similarity between examples is usually based on some simple distance measure (e.g., Manhattan, Euclidean, etc.).<sup>8</sup>

The following formula shows how we can calculate the distance between dataset  $d_{new}$  and dataset  $d_i$ , assuming that all features are numeric and are attributed equal weight:

$$Dist_{mf}(d_{new}, d_i) = \sum_{p=1}^m \frac{|f_{d_{new},p} - f_{d_i,p}|}{\max(f_{*,p}) - \min(f_{*,p})}. \quad (4.1)$$

<sup>8</sup>Distance measures are discussed by Atkeson et al. (1997).

This formula uses the *L1 norm* in the calculation of the distance. The distance value for each metafeature is normalized by dividing it by the corresponding range of values across all datasets. The value of similarity based on metafeatures can be obtained from the distance using:  $Sim_{mf} = 1 - Dist_{mf}$ .

#### 4.9.2 Similarity based on performance results of algorithms

The following two similarity measures are based on recent work of Leite and Brazdil (2021).

##### Cosine-based similarity of performance results

This version of similarity calculates the similarity between two datasets by considering the performance results of different algorithms on these datasets. The actual measure used here is *cosine similarity*. This measure permits to calculate the similarity between two vectors  $\mathbf{v}(d_{new})$  and  $\mathbf{v}(d_i)$  representing dataset  $d_{new}$  and  $d_i$  as follows (Manning et al., 2009):

$$Sim_{cos}(d_{new}, d_i) = \frac{\mathbf{v}(d_{new}) \cdot \mathbf{v}(d_i)}{|\mathbf{v}(d_{new})|_2 * |\mathbf{v}(d_i)|_2}, \quad (4.2)$$

where the numerator represents the *dot product* (inner product) of the two vectors, while the denominator is the product of their Euclidean lengths. The denominator is used to normalize the resulting values so that they would be in the range between 0 and 1. Here the vectors  $\mathbf{v}(d_{new})$  and  $\mathbf{v}(d_i)$  represent performance values obtained by evaluating algorithms (workflows)  $\mathbf{a}$  on dataset  $d_{new}$  or  $d_i$  respectively. This can be represented by function  $p(\mathbf{a}, d_{new})$  and  $p(\mathbf{a}, d_i)$ . Consequently, the equation above can be rewritten as

$$Sim_{cos}(d_{new}, d_i) = \frac{p(\mathbf{a}, d_{new}) \cdot p(\mathbf{a}, d_i)}{|p(\mathbf{a}, d_{new})|_2 * |p(\mathbf{a}, d_i)|_2}. \quad (4.3)$$

After substituting the dot product by the sum of products we get

$$Sim_{cos}(d_{new}, d_i) = \frac{\sum_{a_k \in \mathbf{a}} p(a_k, d_i) * p(a_k, d_{new})}{|p(\mathbf{a}, d_{new})|_2 * |p(\mathbf{a}, d_i)|_2}. \quad (4.4)$$

The Euclidean length of the terms in the denominator of the form  $|\mathbf{x}|_2$  is calculated as  $\sum \sqrt{x_k^2}$ . The algorithm set  $\mathbf{a}$  is a subset of all possible elements which were already evaluated on  $d_{new}$ .

##### Correlation-based similarity of performance results

This version of similarity is similar to the previous one. It also calculates similarity between two datasets by considering the performance results of different algorithms on these datasets. Instead of using cosine similarity it uses similarity based on Spearman's correlation

$$Sim_{rs}(d_{new}, d_i) = r_s(p(\mathbf{a}, d_{new}), p(\mathbf{a}, d_i)), \quad (4.5)$$

where  $p(\mathbf{a}, d_{new})$  is a function applied to a vector of algorithms (workflows)  $\mathbf{a}$  that returns the corresponding performance values (e.g., accuracies) on dataset  $d_{new}$ , and  $p(\mathbf{a}, d_i)$  is defined in a similar way. The term  $r_s$  represents Spearman's correlation function.

Another similar variant uses a weighted rank measure of correlation (da Costa and Soares, 2005; da Costa, 2015) discussed in Chapter 3 (Section 3.2)

$$Sim_{rw}(d_{new}, d_i) = r_w(p(\mathbf{a}, d_{new}), p(\mathbf{a}, d_i)). \quad (4.6)$$

Similarly,  $r_w$  represents the weighted rank correlation function.

## References

- Adya, M., Collopy, F., Armstrong, J., and Kennedy, M. (2001). Automatic identification of time series features for rule-based forecasting. *International Journal of Forecasting*, 17(2):143–157.
- Aha, D. W. (1992). Generalizing from case studies: A case study. In Sleeman, D. and Edwards, P., editors, *Proceedings of the Ninth International Workshop on Machine Learning (ML92)*, pages 1–10. Morgan Kaufmann.
- Atkeson, C. G., Moore, A. W., and Schaal, S. (1997). Locally weighted learning. *Artificial Intelligence Review*, 11(1-5):11–73.
- Baldi, P. and Chauvin, Y. (1993). Neural networks for fingerprint recognition. *Neural Computation*, 5.
- Bensusan, H. (1998). God doesn't always shave with Occam's razor - learning when and how to prune. In *ECML '98: Proceedings of the 10th European Conference on Machine Learning*, pages 119–124, London, UK. Springer-Verlag.
- Bensusan, H. and Giraud-Carrier, C. (2000). Discovering task neighbourhoods through landmark learning performances. In Zighed, D. A., Komorowski, J., and Zytkow, J., editors, *Proceedings of the Fourth European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD 2000)*, pages 325–330. Springer.
- Bensusan, H., Giraud-Carrier, C., and Kennedy, C. (2000). A higher-order approach to meta-learning. In *Proceedings of the ECML 2000 Workshop on Meta-Learning: Building Automatic Advice Strategies for Model Selection and Method Combination*, pages 109–117. ECML 2000.
- Bensusan, H. and Kalousis, A. (2001). Estimating the predictive accuracy of a classifier. In Flach, P. and De Raedt, L., editors, *Proceedings of the 12th European Conference on Machine Learning*, pages 25–36. Springer.
- Box, G. and Jenkins, G. (2008). *Time Series Analysis, Forecasting and Control*. John Wiley & Sons.
- Brazdil, P., Gama, J., and Henery, B. (1994). Characterizing the applicability of classification algorithms using meta-level learning. In Bergadano, F. and De Raedt, L., editors, *Proceedings of the European Conference on Machine Learning (ECML94)*, pages 83–102. Springer-Verlag.
- Brazdil, P. and Henery, R. J. (1994). Analysis of results. In Michie, D., Spiegelhalter, D. J., and Taylor, C. C., editors, *Machine Learning, Neural and Statistical Classification*, chapter 10, pages 175–212. Ellis Horwood.
- Brazdil, P., Soares, C., and da Costa, J. P. (2003). Ranking learning algorithms: Using IBL and meta-learning on accuracy and time results. *Machine Learning*, 50(3):251–277.
- Bromley, J., Guyon, I., LeCun, Y., Säckinger, E., and Shah, R. (1994). Signature verification using a “siamese” time delay neural network. In *Advances in Neural Information Processing Systems 7, NIPS'94*, pages 737–744.
- Chatfield, C. (2003). *The Analysis of Time Series: An Introduction*. Chapman & Hall/CRC, 6th edition.

- Chen, K. and Salman, A. (2011). Extracting speaker-specific information with a regularized Siamese deep network. In *Advances in Neural Information Processing Systems 24*, NIPS'11, pages 298–306.
- Costa, A. J., Santos, M. S., Soares, C., and Abreu, P. H. (2020). Analysis of imbalance strategies recommendation using a meta-learning approach. In *7th ICML Workshop on Automated Machine Learning (AutoML)*.
- Cunha, T., Soares, C., and de Carvalho, A. C. (2018a). cf2vec: Collaborative filtering algorithm selection using graph distributed representations. *arXiv preprint arXiv:1809.06120*.
- Cunha, T., Soares, C., and de Carvalho, A. C. (2018b). Metalearning and recommender systems: A literature review and empirical study on the algorithm selection problem for collaborative filtering. *Information Sciences*, 423:128 – 144.
- da Costa, J. P. (2015). *Rankings and Preferences: New Results in Weighted Correlation and Weighted Principal Component Analysis with Applications*. Springer.
- da Costa, J. P. and Soares, C. (2005). A weighted rank measure of correlation. *Aust. N.Z. J. Stat.*, 47(4):515–529.
- de Souto, M. C. P., Prudencio, R. B. C., Soares, R. G. F., de Araujo, D. S. A., Costa, I. G., Ludermir, T. B., and Schliep, A. (2008). Ranking and selecting clustering algorithms using a meta-learning approach. In *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, pages 3729–3735.
- dos Santos, P. M., Ludermir, T. B., and Prudêncio, R. B. C. (2004). Selection of time series forecasting models based on performance information. In *Proceedings of the Fourth International Conference on Hybrid Intelligent Systems (HIS'04)*, pages 366–371.
- Ferrari, D. and de Castro, L. (2015). Clustering algorithm selection by meta-learning systems: A new distance-based problem characterization and ranking combination methods. *Information Sciences*, 301:181–194.
- Fürnkranz, J. and Petrak, J. (2001). An evaluation of landmarking variants. In Giraud-Carrier, C., Lavrač, N., and Moyle, S., editors, *Working Notes of the ECML/PKDD 2000 Workshop on Integrating Aspects of Data Mining, Decision Support and Meta-Learning*, pages 57–68.
- Fusi, N., Sheth, R., and Elibol, M. (2018). Probabilistic matrix factorization for automated machine learning. In *Advances in Neural Information Processing Systems 32*, NIPS'18, pages 3348–3357.
- Gama, J. and Brazdil, P. (1995). Characterization of classification algorithms. In Pinto-Ferreira, C. and Mamede, N. J., editors, *Progress in Artificial Intelligence, Proceedings of the Seventh Portuguese Conference on Artificial Intelligence*, pages 189–200. Springer-Verlag.
- Hilario, M. and Kalousis, A. (2001). Fusion of meta-knowledge and meta-data for case-based model selection. In Siebes, A. and De Raedt, L., editors, *Proceedings of the Fifth European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD01)*. Springer.
- Ho, T. and Basu, M. (2002). Complexity measures of supervised classification problems. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(3):289–300.
- Kalousis, A. (2002). *Algorithm Selection via Meta-Learning*. PhD thesis, University of Geneva, Department of Computer Science.
- Kalousis, A. and Hilario, M. (2000). Model selection via meta-learning: A comparative study. In *Proceedings of the 12th International IEEE Conference on Tools with AI*. IEEE Press.



- Kalousis, A. and Hilario, M. (2001a). Feature selection for meta-learning. In Cheung, D. W., Williams, G., and Li, Q., editors, *Proc. of the Fifth Pacific-Asia Conf. on Knowledge Discovery and Data Mining*. Springer.
- Kalousis, A. and Hilario, M. (2001b). Model selection via meta-learning: a comparative study. *Int. Journal on Artificial Intelligence Tools*, 10(4):525–554.
- Kalousis, A. and Hilario, M. (2003). Representational issues in meta-learning. In *Proceedings of the 20th International Conference on Machine Learning, ICML'03*, pages 313–320.
- Kalousis, A. and Theoharis, T. (1999). NOEMON: Design, implementation and performance results of an intelligent assistant for classifier selection. *Intelligent Data Analysis*, 3(5):319–337.
- Köpf, C. and Iglezakis, I. (2002). Combination of task description strategies and case base properties for meta-learning. In Bohanec, M., Kavšek, B., Lavrač, N., and Mladenić, D., editors, *Proceedings of the Second International Workshop on Integration and Collaboration Aspects of Data Mining, Decision Support and Meta-Learning (IDDM-2002)*, pages 65–76. Helsinki University Printing House.
- Köpf, C., Taylor, C., and Keller, J. (2000). Meta-analysis: From data characterization for meta-learning to meta-regression. In Brazdil, P. and Jorge, A., editors, *Proceedings of the PKDD 2000 Workshop on Data Mining, Decision Support, Meta-Learning and ILP: Forum for Practical Problem Presentation and Prospective Solutions*, pages 15–26.
- Kuhn, M. and Johnson, K. (2013). *Applied Predictive Modeling*. Springer.
- Leite, R. and Brazdil, P. (2004). Improving progressive sampling via meta-learning on learning curves. In Boulicaut, J.-F., Esposito, F., Giannotti, F., and Pedreschi, D., editors, *Proc. of the 15th European Conf. on Machine Learning (ECML2004)*, LNAI 3201, pages 250–261. Springer-Verlag.
- Leite, R. and Brazdil, P. (2005). Predicting relative performance of classifiers from samples. In *Proceedings of the 22nd International Conference on Machine Learning, ICML'05*, pages 497–503, NY, USA. ACM Press.
- Leite, R. and Brazdil, P. (2007). An iterative process for building learning curves and predicting relative performance of classifiers. In *Proceedings of the 13th Portuguese Conference on Artificial Intelligence (EPIA 2007)*, pages 87–98.
- Leite, R. and Brazdil, P. (2021). Exploiting performance-based similarity between datasets in metalearning. In Guyon, I., van Rijn, J. N., Treguer, S., and Vanschoren, J., editors, *AAAI Workshop on Meta-Learning and MetaDL Challenge*, volume 140, pages 90–99. PMLR.
- Leite, R., Brazdil, P., and Vanschoren, J. (2012). Selecting classification algorithms with active testing. In *Machine Learning and Data Mining in Pattern Recognition*, pages 117–131. Springer.
- Lemke, C. and Gabrys, B. (2010). Meta-learning for time series forecasting and forecast combination. *Neurocomputing*, 74:2006–2016.
- Lindner, G. and Studer, R. (1999). AST: Support for algorithm selection with a CBR approach. In Giraud-Carrier, C. and Pfahringer, B., editors, *Recent Advances in Meta-Learning and Future Work*, pages 38–47. J. Stefan Institute.
- Lorena, A., Maciel, A., de Miranda, P., Costa, I., and Prudêncio, R. (2018). Data complexity meta-features for regression tasks. *Machine Learning*, 107(1):209–246.
- Manning, C., Raghavan, P., and Schütze, H. (2009). *An Introduction to Information Retrieval*. Cambridge University Press.
- Michie, D., Spiegelhalter, D. J., and Taylor, C. C. (1994). *Machine Learning, Neural and Statistical Classification*. Ellis Horwood.

- Muñoz, M., Villanova, L., Baatar, D., and Smith-Miles, K. (2018). Instance Spaces for Machine Learning Classification. *Machine Learning*, 107(1).
- Mueller, J. and Thyagarajan, A. (2016). Siamese recurrent architectures for learning sentence similarity. In *Thirtieth AAAI Conference on Artificial Intelligence*.
- Peng, Y., Flach, P., Brazdil, P., and Soares, C. (2002). Improved dataset characterisation for meta-learning. In *Discovery Science*, pages 141–152.
- Perez, E. and Rendell, L. (1996). Learning despite concept variation by finding structure in attribute-based data. In *Proceedings of the 13th International Conference on Machine Learning*, ICML'96.
- Pfahring, B., Bensusan, H., and Giraud-Carrier, C. (2000). Meta-learning by landmarking various learning algorithms. In Langley, P., editor, *Proceedings of the 17th International Conference on Machine Learning*, ICML'00, pages 743–750.
- Pimentel, B. A. and de Carvalho, A. C. (2019). A new data characterization for selecting clustering algorithms using meta-learning. *Information Sciences*, 477:203 – 219.
- Pinto, F. (2018). *Leveraging Bagging for Bagging Classifiers*. PhD thesis, University of Porto, FEUP.
- Pinto, F., Soares, C., and Mendes-Moreira, J. (2016). Towards automatic generation of metafeatures. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 215–226. Springer International Publishing.
- Post, M. J., van der Putten, P., and van Rijn, J. N. (2016). Does feature selection improve classification? a large scale experiment in OpenML. In *Advances in Intelligent Data Analysis XV*, pages 158–170. Springer.
- Prudêncio, R. and Ludermir, T. (2004). Meta-learning approaches to selecting time series models. *Neurocomputing*, 61:121–137.
- Rendell, L. and Seshu, R. (1990). Learning hard concepts through constructive induction: Framework and rationale. *Computational Intelligence*, 6:247–270.
- Rendell, L., Seshu, R., and Tchong, D. (1987). More robust concept learning using dynamically-variable bias. In *Proceedings of the Fourth International Workshop on Machine Learning*, pages 66–78. Morgan Kaufmann Publishers, Inc.
- Rice, J. R. (1976). The algorithm selection problem. *Advances in Computers*, 15:65–118.
- Rivolli, A., Garcia, L. P. F., Soares, C., Vanschoren, J., and de Carvalho, A. C. P. L. F. (2019). Characterizing classification datasets: a study of meta-features for meta-learning. In *arXiv*. <https://arxiv.org/abs/1808.10406>.
- Smith, M. R., Martinez, T., and Giraud-Carrier, C. (2014). An instance level analysis of data complexity. *Machine Learning*, 95(2):225–256.
- Smith-Miles, K., Baatar, D., Wreford, B., and Lewis, R. (2014). Towards objective measures of algorithm performance across instance space. *Computers & Operations Research*, 45:12–24.
- Smith-Miles, K. A. (2008). Cross-disciplinary perspectives on meta-learning for algorithm selection. *ACM Computing Surveys (CSUR)*, 41(1):6:1–6:25.
- Soares, C. (2004). *Learning Rankings of Learning Algorithms*. PhD thesis, Department of Computer Science, Faculty of Sciences, University of Porto.
- Soares, C. and Brazdil, P. (2006). Selecting parameters of SVM using meta-learning and kernel matrix-based meta-features. In *Proceedings of the ACM SAC*.
- Soares, C., Brazdil, P., and Kuba, P. (2004). A meta-learning method to select the kernel width in support vector regression. *Machine Learning*, 54:195–209.
- Soares, C., Petrak, J., and Brazdil, P. (2001). Sampling-based relative landmarks: Systematically test-driving algorithms before choosing. In Brazdil, P. and Jorge, A., editors, *Proceedings of the 10th Portuguese Conference on Artificial Intelligence (EPIA2001)*, pages 88–94. Springer.

- Soares, R. G. F., Ludermir, T. B., and De Carvalho, F. A. T. (2009). An analysis of meta-learning techniques for ranking clustering algorithms applied to artificial data. In Alippi, C., Polycarpou, M., Panayiotou, C., and Ellinas, G., editors, *Artificial Neural Networks – ICANN 2009*. Springer, Berlin, Heidelberg.
- Sohn, S. Y. (1999). Meta analysis of classification algorithms for pattern recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21(11):1137–1144.
- Sun, Q. and Pfahringer, B. (2013). Pairwise meta-rules for better meta-learning-based algorithm ranking. *Machine Learning*, 93(1):141–161.
- Todorovski, L., Blockeel, H., and Džeroski, S. (2002). Ranking with predictive clustering trees. In Elomaa, T., Mannila, H., and Toivonen, H., editors, *Proc. of the 13th European Conf. on Machine Learning*, number 2430 in LNAI, pages 444–455. Springer-Verlag.
- Todorovski, L., Brazdil, P., and Soares, C. (2000). Report on the experiments with feature selection in meta-level learning. In Brazdil, P. and Jorge, A., editors, *Proceedings of the Data Mining, Decision Support, Meta-Learning and ILP Workshop at PKDD 2000*, pages 27–39.
- Todorovski, L. and Džeroski, S. (1999). Experiments in meta-level learning with ILP. In Rauch, J. and Zytkow, J., editors, *Proceedings of the Third European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD99)*, pages 98–106. Springer.
- Tomp, D., Muravyov, S., Filchenkov, A., and Parfenov, V. (2019). Meta-learning based evolutionary clustering algorithm. In *Lecture Notes in Computer Science, Vol. 11871*, pages 502–513.
- Tsuda, K., Rätsch, G., Mika, S., and Müller, K. (2001). Learning to predict the leave-one-out error of kernel based classifiers. In *ICANN*, pages 331–338. Springer-Verlag.
- Tukey, J. (1977). *Exploratory Data Analysis*. Addison-Wesley Publishing Company.
- van Rijn, J. N., Abdulrahman, S., Brazdil, P., and Vanschoren, J. (2015). Fast algorithm selection using learning curves. In *International Symposium on Intelligent Data Analysis XIV*, pages 298–309.
- Vanschoren, J. (2019). Meta-learning. In Hutter, F., Kotthoff, L., and Vanschoren, J., editors, *Automated Machine Learning: Methods, Systems, Challenges*, chapter 2, pages 39–68. Springer.
- Vilalta, R. (1999). Understanding accuracy performance through concept characterization and algorithm analysis. In Giraud-Carrier, C. and Pfahringer, B., editors, *Recent Advances in Meta-Learning and Future Work*, pages 3–9. J. Stefan Institute.
- Vukicevic, M., Radovanovic, S., Delibasic, B., and Suknovic, M. (2016). Extending meta-learning framework for clustering gene expression data with component-based algorithm design and internal evaluation measures. *International Journal of Data Mining and Bioinformatics (IJDMB)*, 14(2).
- Yang, C., Akimoto, Y., Kim, D. W., and Udell, M. (2019). Oboe: Collaborative filtering for AutoML model selection. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1173–1183. ACM.



---

## Metalearning Approaches for Algorithm Selection II

**Summary.** This chapter discusses different types of metalearning models, including regression, classification and relative performance models. Regression models use a suitable regression algorithm, which is trained on the metadata and used to predict the performance of given base-level algorithms. The predictions can in turn be used to order the base-level algorithms and hence identify the best one. These models also play an important role in the search for the potentially best hyperparameter configuration discussed in the next chapter. Classification models identify which base-level algorithms are *applicable* or *non-applicable* to the target classification task. Probabilistic classifiers can be used to construct a ranking of potentially useful alternatives. Relative performance models exploit information regarding the relative performance of base-level models, which can be either in the form of rankings or pairwise comparisons. This chapter discusses various methods that use this information in the search for the potentially best algorithm for the target task.

### 5.1 Introduction

In this chapter we discuss different approaches to algorithm selection. Some of them were used in early studies involving metalearning, but many methods have been upgraded and represent quite competitive variants.<sup>1</sup>

In Section 5.2 we discuss the use of suitable regression models within the algorithm recommendation system. If we can predict the performance of a given set of base-level algorithm (e.g., classifiers), we can order them and generate, in effect, a ranking. Then we can use just the topmost element, or else use the top- $N$  strategy discussed in Chapter 2, to search for the potentially best algorithm.

An alternative approach is discussed in Section 5.3 which uses a classification algorithm at a meta-level. Here the aim is to identify which of the base-level algorithms are simply applicable to the target machine learning (e.g., classification) task.

If we use probabilistic classifiers at the meta-level, which provide not only the class (e.g., applicable or non-applicable) but also numeric values related to the probability of classification, then the probabilities can again be used to elaborate a ranking. As in the previous case, the potentially best possible base-level algorithm can be identified.

---

<sup>1</sup>This part is based on pages 36–42 in the first edition of this book. The text has been extended and upgraded.

One important class of approaches is based on pairwise comparisons of the performance of the base-level algorithms. In these approaches the algorithm recommendation system converts the actual base-level performance into information that captures relative performance. Pairwise models are discussed in Sections 5.4, 5.5, and 5.6.

Section 5.7 describes a two-stage approach. First, pairwise models are used to generate features, which are subsequently used to generate predictions.

Some of the methods discussed above conduct the pairwise tests in a kind of preprocessing stage. However, this has a disadvantage, as not all of these tests may be needed in the process of identifying the potentially best algorithm. Some methods use a more intelligent way to identify the potentially useful tests. This method, referred to as *active testing*, is discussed in Section 5.8.

Section 5.9 discusses the possibility of using non-propositional (i.e., relational or ILP-based) representation for the description of datasets. In consequence, the corresponding methods need to be upgraded to deal with such representations.

## 5.2 Using Regression Models in Metalearning Systems

In Chapter 2 we discussed a metalearning system that converted base-level performance data into ranks, and the aim was to predict a ranking of base-level algorithms together with their configurations. Let us for simplicity represent each combination  $a_\theta$  simply by  $\theta \in \Theta$  and refer to it simply as a *configuration*. Symbol  $\Theta$  is used to represent all possible configurations, or the distribution of all possible configurations.

In some approaches, the basic idea is to estimate the performance of each base-level configuration and select the one that leads to the best results. Some authors refer to these models as *empirical performance models* (EPMs) (Leyton-Brown et al., 2009; Hutter et al., 2014; Eggenberger et al., 2018).

### 5.2.1 Empirical Performance Models

The existing approaches can be divided into two groups, according to whether the regressor uses metadata obtained from only the current dataset, or from other datasets as well.

#### Using metadata from the current dataset

Let us analyze first the approaches that do not use metadata obtained on other datasets. This approach forms the basis of *Auto-WEKA* system (Thornton et al., 2013).

When no metadata from other datasets is available, the only source of metadata can be from earlier runs on the dataset itself. At first, several preselected configurations are tested and the outcome constitutes the resulting metadata. These configurations can either be selected at random or by initial design (see, e.g., Pfisterer et al. (2018); Feurer et al. (2018)). The metadata consists of  $n$  cases (meta-examples) of the form

$$MetaD \equiv (\theta_i, P_i)^n, \quad (5.1)$$

where  $i = \{1, 2, \dots, n\}$ ,  $\theta_i$  represents the  $i^{th}$  algorithm configuration, and  $P_i$  is the performance of this configuration. A suitable regressor can be trained on this metadata, and this way we obtain a meta-level model *MetaM*:

$$MetaM \leftarrow Train(MetaD). \quad (5.2)$$

As in all learning tasks, the performance depends on how many cases are available. So we need a sufficient number of cases in  $MetaD$  to obtain a reasonable performance. The empirical performance model  $MetaM$  can be used to predict (more precisely, estimate) the performance for some new configuration that is not even a part of the existing metadata  $MetaD$ :

$$\hat{P}_k \leftarrow Predict(MetaM, \theta_k). \quad (5.3)$$

So different configurations can be queried, and the one with the estimated best performance selected and evaluated on the current base-level dataset. More details regarding this are given in Section 6.3.

### Approaches that use metadata from other datasets

Some researchers used information on other datasets in the search for the best possible configuration (Gama and Brazdil, 1995; Sohn, 1999; Köpf et al., 2000; Bensusan and Kalousis, 2001; Feurer et al., 2014). Let us see how this works. First, let us extend the notation in Eq. 5.1 to include also different datasets,

$$MetaD \equiv (\theta_i, d_j, P_{i,j})_{i=1, j=1}^{n, m}, \quad (5.4)$$

where  $\theta_i$  represents a configuration and  $d_j$  the dataset on which the performance  $P_{i,j}$  is measured. Each dataset is characterized by metafeatures  $mf(d_j)$ .<sup>2</sup> An exhaustive overview of possible metafeatures is presented in Chapter 4.

The meta-level model,  $MetaM$ , can be trained as shown in Eq. 5.2. Obviously, constructing such a model is more complex, as in the previous case, as it involves various datasets. Predictions can be done by invoking

$$\hat{P}_{k,t} \leftarrow Predict(MetaM, \theta_k, d_t). \quad (5.5)$$

The prediction task has two parameters, if we disregard  $MetaM$  itself. One is the configuration  $\theta_k$ . The aim is to find a configuration  $\theta^*$  that achieves the highest possible performance on the target dataset  $d_t$ . So one naive way to obtain this is to consider various configurations generated by the meta-model and select the best one.

Various researchers have devised other, more effective approaches. Typically, the search would be done in an iterative manner. The best configuration found in one iteration is used to update the meta-level model or else to condition the future search in some way. This way the method requires less time to identify the potentially best solution. Some metalearning methods that follow this line are discussed further on in this chapter (Sections 5.6 and 5.8) and in Chapter 6 (Sections 6.3 and 6.4).

Let us come back to Eq. 5.5 describing how the prediction is obtained. Another parameter in this process is the target dataset  $d_k$  characterized by metafeatures  $mf(d_k)$ . Although this parameter is fixed, it affects the performance of the meta-level model. So one question is — how can it be exploited to boost up the performance?

This can be achieved by *adapting* the meta-level model  $MetaM$ , constructed on the basis of the test results on different datasets in  $D \equiv (d_i)^m$ . So one important part of

---

<sup>2</sup>Note that the above notation implies that the configurations are represented in the form of a finite list and hence can be enumerated. It is not required that a performance result exists for every configuration and dataset pair.

the adaptation process is to consider the similarity between the target dataset  $d_t$  and the individual datasets in  $D$ . Let us represent the similarity by  $Sim(d_t, d_j)$ . Chapter 4 (Section 4.10) shows how the similarity can be calculated using the dataset measures.

Chapter 2 (Section 2.2) shows how the similarity can be exploited to adapt the ranking approach to provide a specific ranking. As Brazdil et al. (2003) have shown, this leads to enhanced performance. A similar adaptation process is described in Subsection 5.8.2 in relation to the *active testing* (AT) method.

## 5.2.2 Normalizing performance

Earlier we have shown the format of metadata  $MetaD$  (see Eq. 5.1). We see that it involves performance results obtained on different datasets. However, as has already been noted in Chapter 3, the range of performance values (in absolute terms) may vary substantially for different datasets. An accuracy of 90% may be quite high on a classification problem but low on another one. If the aim is to exploit metadata from different datasets in a metalearning system, it is useful to normalize the performance values in a kind of preprocessing step. Chapter 3 (Section 3.1) describes some common approaches that can be used to do this.

## 5.2.3 Performance models

Performance models can be represented in an extensional form or in an intensional one (e.g. using a function). The former corresponds to enumeration of different cases, as in instance-based learning (IBL) approaches to machine learning. This representation has been used in many ranking approaches, where the cases are reordered according to the chosen performance value.

Our aim in this chapter is to discuss models that use an intensional form. In principle, it is possible to use any regression algorithms for the task of predicting performance. However, some have proved more useful than others, when judged by the quality of predictions.

In one rather early work by Gama and Brazdil (1995), the authors used linear regression, regression trees, model trees, and IBL to estimate the error of a large number of base-level algorithms.

In another study, a comparison between *Cubist*<sup>3</sup> and a *kernel method* was carried out on a problem of estimating the error of ten classification algorithms (Bensusan and Kalousis, 2001). Results showed a slight advantage for the kernel method.

Some researchers have adopted *Gaussian processes* (e.g., Rasmussen and Williams (2006)), others *random forests* (RFs) (e.g. Eggenberger et al. (2018); Hutter et al. (2014); Leyton-Brown et al. (2009)). As both types of meta-level models are discussed in more detail in Chapter 6, we do not provide more details here.

Some proposals combined several models at the meta-level. One early metalearning approach has been discussed by Gama and Brazdil (1995). They showed that a linear combination of the meta-level models yields better results than any of these models considered individually.

---

<sup>3</sup>Cubist combines rules with regression trees and can be seen as an extension of the M5 model tree (Quinlan, 1992).



Table 5.1: Examples of different forms of recommendation

	Algorithms					
	$a_1$	$a_2$	$a_3$	$a_4$	$a_5$	$a_6$
(1) Estimates of performance	0.89	0.68	0.90	0.74	0.81	0.75
	Rank					
	1	2	3	4	5	6
(2) Ranking (linear and complete)	$a_3$	$a_1$	$a_5$	$a_6$	$a_4$	$a_2$

### 5.2.4 Clustering trees

Clustering trees can be used to induce a single model for several target variables and hence provide multitarget prediction (Blockeel et al., 1998). In our case, the performance of multiple base-algorithms represented in the form of the multiple targets.

Clustering trees are obtained with a common algorithm for top-down induction of decision trees (TDIDT) that tries to minimize the variance of the target variables for the cases in every leaf (and maximize the variance across different leaves). They have been applied to the problem of estimating the performance of several algorithms (Todorovski et al., 2002). The decision nodes represent tests on the values of metafeatures, and the leaf nodes represent sets of performance estimates, one for each algorithm.

The results obtained with clustering trees are comparable to those obtained with the approach using separate models, with the advantage of improved readability, because the former approach generates a single model rather than several models (Todorovski et al., 2002).

### 5.2.5 Transforming performance predictions into rankings

A set of performance estimates of different base-level algorithms can be transformed into rankings by ordering the algorithms according to their expected performance (Sohn, 1999; Bensusan and Kalousis, 2001). This is illustrated in Table 5.1. The performance values (line 1) can be used to reorder the corresponding algorithms to generate a ranking (line 2). This ranking can be followed in a top- $n$  fashion (as described in Chapter 2, Section 2.2).

It could be argued that, in many cases, the user simply requires to identify the potentially best algorithm and hence detailed information on performance has simply an auxiliary role.

Not much work has been dedicated to empirical comparisons of different forms of recommendation discussed here and in Chapter 2 (Section 2.1). One early study was done by Bensusan and Kalousis (2001). However, it would be useful to conduct a new study which would incorporate up-to-date approaches and methods.

### 5.2.6 Predicting performance for each example

A different approach to estimate the performance of an algorithm is based on the use of metalearning model to predict the performance of a base-level algorithm on each individual base-level example (Tsuda et al., 2001). For instance, in a classification problem,

the metalearning model predicts whether the base-level algorithm correctly predicts the class for each test example. A prediction of the performance of the algorithm on the dataset is obtained by aggregating the set of individual predictions that are obtained with the metalearning model. In the classification setting, the predicted performance of the algorithm could be given by the predicted accuracy.

## 5.2.7 Advantages and disadvantages of performance predictions

### Advantages

By providing performance estimates for each algorithm rather than, for instance, a rank, more information is provided. As we have argued earlier (Subsection 5.2.5), the performance estimates can be converted into rankings quite easily.

Additionally, each regression problem can be solved independently, generating one model for each base-algorithm. With this approach, it is easier to change the portfolio comprising the set of base-algorithms. Removing an algorithm simply means eliminating the corresponding meta-level model, while inserting a new one can be done by generating the corresponding meta-level model. In both cases, the meta-level models of the remaining algorithms are not affected.

### Disadvantages

Generating as many meta-level models as there are algorithms has a disadvantage. It is not trivial to understand when an algorithm performs better than another one and vice versa. Take, for instance, the rules presented in Table 5.2 that were selected from models that predict the error of C4.5 and CN2 (Gama and Brazdil, 1995). The metafeatures are: *fract1*, the first normalized eigenvalues of canonical discriminant matrix; *cost*, a Boolean value indicating if errors have different costs; and *Ha*, the entropy of attributes. These models do not describe directly the conditions when C4.5 is better than CN2 and vice versa.

Table 5.2: Four sample rules that predict the error of C4.5 and CN2 (Gama and Brazdil, 1995).

Estimated		
Algorithm	Error	Conditions
C4.5	22.5	$\leftarrow fract1 > 0.2 \wedge cost > 0$
c4.5	58.2	$\leftarrow fract1 < 0.2$
CN2	8.5	$\leftarrow Ha \leq 5.6$
CN2	60.4	$\leftarrow Ha > 5.6 \wedge cost > 0$

It can be expected that predicting individual performance values is much harder than discriminating between a finite number of classes or predicting rankings.

Additionally, the fact that several regression problems are solved independently can be regarded as a disadvantage. In fact, the error in itself is not so important as the question of whether it affects the relative order of the algorithms. This issue was addressed by other researchers who have provided models for pairs of algorithms. One work on this line is discussed in Section 5.4.

Another disadvantage of this approach was that it provided a pointwise prediction. It is well known that the performance of a given algorithm normally varies substantially across different datasets. Some examples of this are shown in Chapter 16. Consequently, interval prediction provides information about the variability of the performance, which can be exploited in algorithm selection and configuration. The methods discussed in Chapter 6 do just that.

### 5.3 Using Classification at Meta-level for the Prediction of Applicability

In one early work of Brazdil et al. (1994), the aim was to predict whether a given base-level classification algorithm was applicable (with relatively low error) or non-applicable (with relatively high error) to a given target dataset. Following the notation introduced in Section 5.2.1, the metadata  $MetaD$  of the form  $(\theta_i, d_j, P_{i,j})_{i=1, j=1}^{n, m}$  was transformed into metadata suitable for a classification algorithm. That is, each  $P_{i,j}$  was discretized into two possible categorical values, *applicable* and *non-applicable*.

A suitable classification algorithm was then used at the meta-level. Each dataset was described using dataset characteristics (see Chapter 4 for details), and then dataset similarity was also explored to generate a specific recommendation for the target dataset.

The fact that the prediction is categorical class value (e.g. applicable/non-applicable) may seem as a limiting factor, as it is not possible to determine the order in which the base-algorithms should be tried out. However, if we included  $P_{i,j}$  as a feature and opted for a probabilistic classifier at the meta-level, this limitation would be mitigated. The class value generated by the probabilistic classifier could be accompanied by the estimate of probability. Hence, it would be possible to order the base-level algorithms according to this probability. So the effects could be similar to those obtained with classical ranking approaches (see Chapter 2).

#### 5.3.1 Classification algorithms used at meta-level

Given their wide availability, many different classification algorithms have been used in meta-level learning. Early studies focused on the decision tree classifier (Brazdil et al., 1994) or  $k$ -NN classifier (Brazdil et al., 2003).

An extreme example is to use the same set of base-level algorithms also at the meta-level (Pfahring et al., 2000; Bensusan and Giraud-Carrier, 2000). In this work, the ten classification algorithms used were quite diverse, including decision trees, a linear discriminant, and neural networks, among others. The authors compared pairs of meta-level classifiers on several metalearning problems. The comparisons were not conclusive in one of the studies (Bensusan and Giraud-Carrier, 2000), while in another study (Pfahring et al., 2000) the results indicated that decision tree- and rule-based models lead to better metalearning results. In another study (Kalousis, 2002; Kalousis and Hilario, 2000), the authors compared four variants of decision tree classifiers, IBL, and boosted C5.0 at the meta-level. The best results were obtained by boosted C5.0.

### 5.4 Methods Based on Pairwise Comparisons

Various authors have studied the issue of using pairwise models for the task of predicting the potentially best algorithm, or alternatively, a ranking of algorithms. Some early work

focused on the problem of predicting which of the two algorithms is likely to perform better on a new dataset (Pfahring et al., 2000; Fürnkranz and Petrak, 2001; Soares et al., 2001; Leite and Brazdil, 2004, 2005).

In subsequent years, some researchers have addressed the issue of how the method of pairwise comparisons could be incorporated into a method that provides a ranking of all algorithms, or else identify the first element, i.e., the potentially best-performing algorithms (plus possibly all algorithms with equivalent performance).

Some of the methods generate various pairwise models, which are used afterwards to obtain a ranking (Leite and Brazdil, 2010; Sun and Pfahring, 2012, 2013).

Other methods use an iterative method which conducts pairwise tests as the method proceeds. When this process is terminated, the potentially best algorithm is returned (Leite and Brazdil, 2007; van Rijn et al., 2015). More details about these methods are given in the following subsections.

#### 5.4.1 Pairwise tests that exploit landmarks

Pfahring et al. (2000) proposed the use of simplified and fast versions of algorithms referred to as *landmarks* in pairwise comparisons (see Chapter 4, Section 4.2 for more details on landmarks). The authors have shown that landmarks can be used for the problem of predicting which of the two algorithms is likely to perform better on a new dataset.

Other researchers have proposed that, apart from simplified versions of algorithms, one could also use simplified versions of datasets. This concept was referred to as *sub-sampling landmarks* (Fürnkranz and Petrak, 2001; Soares et al., 2001).

The experimental results carried out at the time did not seem to show a clear advantage of this approach. However, as was shown later (van Rijn et al., 2015; van Rijn, 2016), the method that uses the performance on a single subsample of the data to select the classifier that performs best is rather competitive and beats many more complex methods. This method is referred to as the *best-on-sample* method. In other communities it is referred to as *constant prediction* or *constant liar*. This simple baseline gave rise to complex methods, such as *successive halving* (Jamieson and Talwalkar, 2016) and *Hyperband* (Li et al., 2017).

One surprising result of the early work cited earlier was that using more information in the form of more samples did not lead to marked improvement. This led some researchers (Leite and Brazdil, 2004, 2005) to investigate the reasons for this. This study led to a new variant of the method which is discussed in the next subsection.

#### 5.4.2 Pairwise method oriented towards partial learning curves

Leite and Brazdil (2004, 2005) proposed a variant of the pairwise method that used a series of samples as data characteristics. The series of samples used represents, in effect, a *partial learning curve*. The method relies on a meta-dataset that includes full learning curves of all algorithms on historical datasets. On the new target dataset it is necessary to obtain partial learning curves. This process typically takes less time than obtaining the performance value on a full dataset.

Based on the partial learning curve on the target dataset, and full learning curves on historical datasets, the meta-algorithm can determine which algorithm to recommend.

Figure 5.1 shows an example of learning curves. The  $x$ -axis shows the number of data points; the  $y$ -axis shows the measured performance. We can see some typical behavior

of learning curves: (i) most classifiers typically perform better when larger data samples are available, (ii) this is not always the case, as sometimes performance may drop when more data is available, and (iii) learning curves can cross, as some classifiers that do not perform well on small samples can attain higher performance on larger data samples.

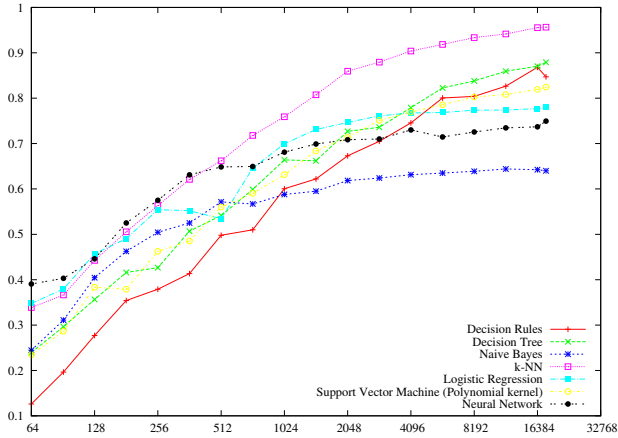


Fig. 5.1: Learning curves of various Weka classifiers on the “letter” dataset. Image taken from van Rijn (2016).

The method described can be used to solve the following problem: Given two algorithms  $a_p$  and  $a_q$ , which of them will most likely perform best on the current dataset? The method involves the following steps:

- Determine how to represent learning curves;
- Obtain a partial learning curve of both algorithms  $a_p$  and  $a_q$  on the target dataset;
- Identify datasets with most similar curves while examining the meta-database of past experiments;
- Use the retrieved curves to predict the performance  $a_p$  and  $a_q$  for the target dataset. Predict which algorithm will achieve a higher performance.

The main idea of this method is to save time. A more expensive CV test is substituted by a cheaper estimate, as is done when using surrogate models discussed in Chapter 6.

As this idea was reused in follow-up work (e.g., Leite and Brazdil (2010); van Rijn et al. (2015)), the method is explained in some detail in the following subsections.

## Representation of partial learning curves

Each partial learning curve is represented by a sequence of performance values of a given algorithm on a given dataset on samples of increasing size. Formally, partial learning curve  $P_{a_p, d_i}$  is represented by a sequence of elements of the form  $p_{a_p, d_i, k}$ , where  $a_p$  represents an algorithm,  $d_i$  a dataset, and  $k$  an index of a sample which varies from 1 to  $k_{max}$ .

Following previous work of Provost et al. (1999), the sizes follow a geometric progression. Leite and Brazdil (2005) have set the size of the  $k$ -th sample to the rounded

value of  $2^{6+0.5 \times k}$ . So, the size of the first sample is set to  $2^{6.5}$ , giving 91 after rounding, and the second sample is set to  $2^7$ , i.e., 128, etc. This scheme was reused in various subsequent papers.

## Conducting tests on the target dataset

Tests are conducted with the aim of obtaining a partial learning curve of both algorithm  $a_p$  and  $a_q$  on the target dataset.

## Identification of the most similar learning curve

In this step the aim is to identify datasets whose learning curves are most similar to the partial learning curve elaborated for the target dataset. Once such curves have been identified, the performance for any sample size can be retrieved and used for prediction.

The similarity between two learning curves is judged by an adaptation of the nearest neighbor algorithm ( $k$ -NN). The distance measure used between datasets  $d_i$  and  $d_j$  for a given pair of algorithms  $a_p$  and  $a_q$  is defined by the sum of two distances, one for algorithm  $a_p$  and the other for  $a_q$ :

$$d_{a_p, a_q}(d_i, d_j) = d_{a_p}(d_i, d_j) + d_{a_q}(d_i, d_j). \quad (5.6)$$

The first term determines the distance between two learning curves of algorithm  $a_p$ , one on dataset  $d_i$  and the other on dataset  $d_j$ . It is calculated as follows:

$$d_{a_p}(i, j) = \sum_{k=1}^{k_{max}} (p_{a_p, i, k} - p_{a_p, j, k}). \quad (5.7)$$

As the learning curve has  $k$  points, Eq. 5.7 sums the individual distances for all  $k$  points. The second term, i.e.  $d_{a_q}(d_i, d_j)$ , is calculated in a similar way.

## Adaptation of retrieved curves

Leite and Brazdil (2005) have observed that, although the retrieved learning curve may often have the same shape, it may be shifted up (or down) with respect to the curve on the target dataset. They have concluded that this may impair the quality of prediction. To avoid this shortcoming the authors have proposed an additional *adaptation* step, related to the notion of *adaptation* in case-based reasoning (Kolodner, 1993; Leake, 1996).

Here the adaptation procedure involves moving the retrieved performance curve  $P_r$  to the partial performance curve  $P_t$  generated for the target dataset and obtaining  $P'_r$ . Adaptation is done by multiplying each item in  $P_r$  by the scale coefficient  $f$ . So the  $k$ -th item in the curve relative to algorithm  $a$  is transformed as follows:  $p'_{a, r, k} = f \times p_{a, r, k}$ . The scale coefficient  $f$  is designed to minimize the Euclidean distance between the two curves on the initial segment. The authors have used different weights for the different items in the curve in accordance with the corresponding sample size. The following equation shows how  $f$  is calculated:

$$f = \frac{\sum_{k=1}^{k_{max}} (p_{a, t, k} \times p_{a, r, k} \times w_k^2)}{\sum_{k=1}^{k_{max}} (p_{a, r, k}^2 \times w_k^2)}. \quad (5.8)$$

The process of adaptation is illustrated in Figure 5.2, reproduced from earlier work (Leite and Brazdil, 2005).

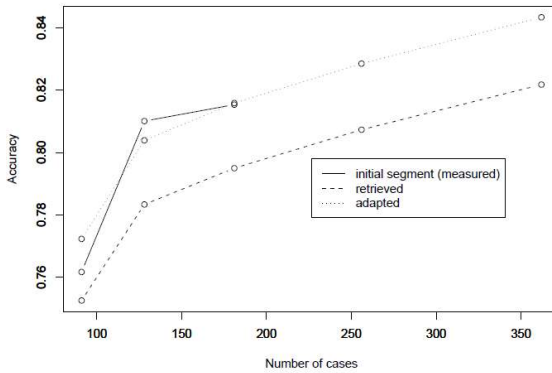


Fig. 5.2: Adaptation of the retrieved learning curve

### Carrying out predictions for $k$ nearest datasets

Assuming that  $k$  nearest datasets have been identified for algorithm  $a_p$  using the method described above, the next step is to obtain  $k$  predictions for the sample size  $S_t$  corresponding to the size of the target dataset. The authors then calculate a mean of the  $k$  values, which is then used as the predicted performance value of  $a_p$ . The predicted performance value of  $a_q$  is calculated in a similar way. The two values are compared to determine which of the two algorithms is better. In the experiments carried out the setting of  $k_{max} = 3$  was used, determining that the partial learning curve should include performance values for three data samples.

### Main results

The authors have studied a pair of algorithms, which included C5 (Quinlan, 1998) and SVM with radial basis kernel (the implementation in e1071 in R (Dimitriadou et al., 2004)). The authors have compared the classification accuracy of the proposed method (A-MDS) with the classical approach that used a set of dataset characteristics (MDC). The performance of MDS that used just one sample (91 cases) was better than the one obtained with MDC.

The authors have also shown that, if a learning curve with more samples was used, the performance would increase. This is not true for a variant that does not use adaptation in the process. This explains why Soares et al. (2001) obtained somewhat discouraging results earlier. As adaptation was not used, using a larger sample in the matching process did not lead to improved results.

Recent work suggests that learning curves can also be used model-free to speed up a cross-validation procedure (Domhan et al., 2015; Mohr and van Rijn, 2021).

## 5.5 Pairwise Approach for a Set of Algorithms

The approach described in the previous subsection was incorporated into various other methods that dealt with  $N$  algorithms. In this subsection we describe one of them, referred to as the SAM method (Leite and Brazdil, 2010). This method involves the following steps:

1. Select a pair of classification algorithms and determine which one of the two is better using a pairwise method.
2. Repeat this for all pairs of algorithms and generate a partial order (a graph).
3. Process the partial order to identify the best algorithm(s).

The details are given in the following subsections.

### Select a pair of algorithms and determine which one is better

This step can be achieved by a cross-validation test followed by a statistical significance test. This method is referred to as  $CV_{ST}$  by the authors. This method can be seen as a function which returns +1 (-1) if algorithm  $a_i$  achieves significantly better (worse) performance than  $a_j$ . The value 0 is returned if this cannot be established.

### Repeat for all pairs and generate a partial order

This step involves applying this method to all algorithm pairs and constructing a graph. When using the output of SAM, link  $a_i \leftarrow a_j$  is drawn if  $a_i$  is significantly better than  $a_j$ . An example of a partial ranking of algorithms generated this way is shown in Figure 5.3. In Chapter 2 this type of ranking was referred to as *quasi-linear ranking*. We note that the figure could include various other links that could be obtained by applying the rule of transitivity. They were omitted so as not to overload the figure with too many links.

### Identify the best algorithm(s)

This step concerns the analysis of the partially ordered set of items with the aim of identifying the topmost item(s). The authors have identified three different aggregation measures corresponding to three different strategies of aggregating the detailed information:

- $W$ , the sum of all wins (outgoing arrows)
- $L$ , the sum of all losses (incoming arrows), each as a negative sign
- $W+L$ , the sum of wins and losses

Let us examine what happens if we process the example shown in Figure 5.3. The algorithm  $MLP$ , for instance, is significantly better than three other algorithms, and so the measure  $W = 3$ . As it is not beaten by any algorithm,  $L = 0$ . Consequently,  $W+L$  is 3. The results for this algorithm and all the others used in our example are shown in Table 5.3.

In this example all three aggregation methods identify the algorithms  $MLP$  and  $LogD$  as the topmost two algorithms.



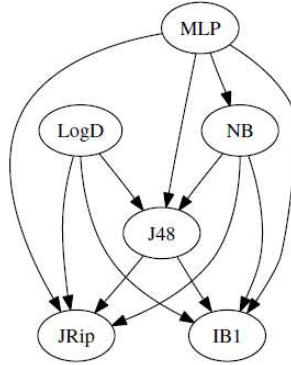


Fig. 5.3: Example of a partial order of six classification algorithms (reproduced from Leite and Brazdil (2010))

Table 5.3: Three different aggregation strategies applied to the example in Fig. 5.3

Algorithm	W	L	W+L
MLP	3	0	3
LogD	3	0	3
NB	3	-1	2
J48	2	-3	-1
JRip	0	-4	-4
IB1	0	-4	-4

## Evaluation

Leite and Brazdil (2010) also discussed the problem of evaluating the method described above. We note that the method can, in general, return more than one algorithm. Let us use  $\hat{\mathbf{A}}$  to refer to this set. In the example in Figure 5.3 the method returned two algorithms. This may not coincide with the set  $\mathbf{A}$  that is considered “correct”. So a question arises as to how we should proceed to calculate the appropriate measure of success,  $M_S$ .

The method of Leite and Brazdil (2010) is based on an assumption that the items (algorithms) in  $\hat{\mathbf{A}}$  could be considered equivalent. Consequently, as soon as  $\mathbf{A}$  includes at least one item of  $\hat{\mathbf{A}}$ ,  $M_S$  returns 1. This is captured in the following equation:

$$M_S = \frac{|\hat{\mathbf{A}} \cap \mathbf{A}|}{|\hat{\mathbf{A}}|}. \quad (5.9)$$

So, for instance, in a situation when  $\hat{\mathbf{A}} = \{a_1\}$  and  $\mathbf{A} = \{a_1, a_2\}$ ,  $M_S$  returns 1. Should  $\hat{\mathbf{A}}$  return several items, some of which are correct while others are not,  $M_S$  returns a

value corresponding to the proportion of correctly identified items in the set of correct items. So, for instance, in a situation when  $\hat{\mathbf{A}} = \{a_1, a_2\}$  and  $\mathbf{A} = \{a_1\}$ ,  $M_S$  returns 0.5. That is, if we chose one of the items from  $\hat{\mathbf{A}}$  at random, the probability of guessing right would be 0.5.

### Disadvantages of this approach

One disadvantage of this approach is that the number of pairwise models needed is  $(N \times (N - 1))/2$ , where  $N$  represents the number of algorithms. When  $N$  is 20, the total number of models is still manageable, i.e., 190. The method does not scale up well for larger numbers.

This problem can be mitigated by identifying a subset of algorithm,  $N_P$  as *pivot algorithms*. Then the  $N$  algorithms are compared with the pivot algorithms  $N_P$ . A study determining whether this could lead to improvements in terms of accuracy and time could be carried out in the future.

As is shown later in Section 5.6, this problem can be mitigated by adopting a search method that identifies the best possible pairwise test in each step. This way, many pairwise tests are actually skipped.

### Using a partial ranking for top- $n$ execution

Partial rankings can be used to schedule tests, leading to the top- $n$  execution discussed in Chapter 2. Conceptually, we can view this as an activity that can be divided into two phases. In the first phase, the algorithms are characterized by one of the possible measures, such as  $W+L$  discussed earlier, and reordering all algorithms according to this measure. The second phase involves following this reordered set using the top- $n$  execution (see Chapter 2).

### Extending the average ranking method to partial rankings

The average ranking method discussed in Chapter 2 exploited a set of rankings, one per dataset, and constructed average ranking. A question arises as to how this could be extended to accept a set of partial rankings.

One possibility involves first annotating each item in each partial ranking with the  $W+L$  measure, and then calculating an average value for each item (algorithm).

## 5.6 Iterative Approach of Conducting Pairwise Tests

In Section 5.5 we discussed an approach that exploits pairwise tests to identify the potentially best algorithm in a set. In this approach, the pairwise tests were conducted in a kind of preprocessing stage. One disadvantage of this approach is that not all pairwise tests may be needed.

In a follow-up work (van Rijn et al., 2015; van Rijn, 2016), the pairwise approach was improved in two different aspects. First, this method used a search method with the aim of identifying the potentially best algorithm. In consequence, the pairwise tests are conducted on demand, as the method proceeds. The method has, at any time, a suggestion regarding what is the potentially best algorithm found so far. So time is not

wasted by carrying out useless pairwise tests in a preparatory phase. This aspect is similar to the active testing method discussed further on in Section 5.8.

The second very important improvement is that this method takes into account the runtime of tests. So relatively fast tests which are likely to lead to good-performing algorithms are preferred to the slower ones. The method referred to as *pairwise curve comparison* (PCC) by the authors involves the following steps:

1. Initialize the current best algorithm and elaborate partial learning curves (one per algorithm) on the target dataset;
2. Search for the best candidate algorithm to perform a test on;  
Conduct the selected test and update the current best algorithm;
3. Repeat the above until the termination condition has been satisfied.

More details about each step are given below.

### Initialize the current best algorithm

This method used the concept of *current best* algorithm,  $a_{best}$ .<sup>4</sup> The authors have proposed to initialize it using a randomly chosen algorithm. However, it is not difficult to envisage an alternative approach that would choose some good candidate to initialize  $a_{best}$ . One such good candidate is the top element in the A3R-based average ranking discussed in Chapter 2.

After the algorithm  $a_{best}$  has been selected, it is necessary to elaborate partial learning curves on the target dataset. This involves conducting tests on different samples of the target dataset. Note that, so far, no tests have been executed on the target dataset.

### Search for the best pairwise test

This step considers all possible candidate algorithms, one by one. Each of these represents a potential candidate  $a_{comp}$  (competitor) that can replace the current best algorithm,  $a_{best}$ . In order to determine whether the replacement should go ahead, it is necessary to predict its performance in relation to  $a_{best}$ .

The method uses  $k$  datasets with the most similar curves to both  $a_{comp}$  and  $a_{best}$  to do this. To determine which  $k$  datasets are most similar, Eq. 5.7 is used. After the learning curves have been retrieved, they are subject to *curve adaptation*, discussed earlier in Section 5.4.2.

The decision regarding whether algorithm  $a_{comp}$  should replace  $a_{best}$  depends on the predictions on  $k$  datasets. If  $a_{comp}$  was predominantly better on the full learning curve on these datasets than  $a_{best}$ , the replacement goes ahead.

This method can output a ranking to be used later. Once the comparison against all other algorithms has been carried out, the algorithm that is at that moment  $a_{best}$  is added to the ranking. At that point, it can be cross-validated. Note that this PCC can result in an *unstable ranking*: the resulting ranking can differ somewhat based on which algorithm is initialized as  $a_{best}$ .

---

<sup>4</sup>In some papers this algorithm is called the *incumbent* algorithm (the holder of a position). Note that the name  $a_{best}$  does not mean that is the best candidate, but rather the current best option at some point in time.

## Upgrade to incorporate accuracy and time

As the authors have shown, the method can be upgraded quite easily to incorporate time into the method. The performance comparisons between  $a_{comp}$  and  $a_{best}$  can use a combined measure of accuracy and time,  $A3R'$ , which is a simplified version of  $A3R$  discussed in Chapter 2:

$$A3R'_{a_j}^{d_i} = \frac{P_{a_j}^{d_i}}{(T_{a_j}^{d_i})^Q}, \quad (5.10)$$

where  $P$  represents performance (e.g., accuracy),  $T$  runtime, and  $Q$  is a scaling factor that controls the importance of runtime.

## Main findings

This work confirmed that it is indeed useful to consider both accuracy and time as a measure of performance, when the aim is to obtain a ranking of classifiers. This strategy that gives preference to faster and relatively good methods earlier pays off. A similar conclusion was also reached in conjunction with two other similar approaches,  $AR^*$ , discussed in Chapter 2, and  $AT^*$ , discussed further on in Section 5.8.

The PCC method dominates two other approaches that were used in the comparative study. One of them was the *average ranking method* (AR), discussed in Chapter 2. The other method was the baseline method, referred to as *best on sample* (BoS). Quite surprisingly, method BoS is very competitive, if the evaluation is done with loss curves that portray how performance (accuracy) depends on the number of tests. The advantage of PCC becomes apparent when the evaluation is done with loss curves that portray how performance depends on runtime. This is, of course, what interests users. Their aim is to identify the potentially best algorithm as soon as possible, and they normally have an allocated time budget for a given task.

## Relationship to surrogate models

The area of automatic hyperparameter configuration discussed in Chapter 6 introduces various useful concepts. One of these is the concept of *surrogate models*, which represent simplified algorithms that provide estimates of the predictions of the actual algorithms but are much faster to execute. An empirical performance model (EPM) is an example of such a surrogate model: instead of running a configuration on a dataset, the empirical performance model can assess whether the configuration is worth trying out. The prediction of an empirical performance model is typically cheaper in terms of runtime than running the actual configuration on a dataset.

The method discussed in this section that predicts the performance of a particular algorithm on the basis of a pair of learning curves (partial learning curve on the target dataset and a full learning curve on a similar dataset) can be related to surrogate models, in the sense that these represent simplified performance models of the underlying algorithm.

Another useful concept used in the area of hyperparameter configuration is the concept of *acquisition function*. This function selects the potentially best parameter configuration to test. We note that the method described in this section incorporates a method for selecting the next pair of algorithms to test. This method can be related to the acquisition function used in the area of hyperparameter configuration.

## 5.7 Using ART Trees and Forests

The approach of Sun and Pfahringer (2013) involves two phases:

1. Construct a set of pairwise models and generate features;
2. Use *approximate ranking trees* (ART) forests in conjunction with new features to generate predictions.

More details about each phase are given in the following subsections.

### Construct a set of pairwise models

In this phase, a set of pairwise metalearning models is generated using a rule-based classifier (RIPPER) (Cohen, 1995). Each model is trained to predict whether to use algorithm  $a_i$  in preference to  $a_j$  or vice versa.

One interesting aspect of this work is that each pairwise model can use a specific set of base-level features. These may include statistical, information-theoretic, and landmarking features, such as AUC associated with a particular type of tree (REP-Tree.depth2).

The binary classifiers are then invoked to generate new metafeatures. In one of the variants of the method, each rule associated with a binary model gives rise to one new feature. These are appended to the more traditional features, including, besides some statistical and information-theoretic features, also landmarkers.

### Use ART forests to generate predictions

In this phase, an *ART forest* is used to generate a prediction. ART forest can be viewed as a random forest of ART trees (*approximate ranking trees*). ART trees are based on the notion of predictive *clustering trees* for ranking (Blockeel et al., 1998) discussed earlier (see Subsection 5.2.4).

The authors show that this approach achieves better predictive performance results than the classical  $k$ -NN approach. They also demonstrate that the addition of performance-based features associated with binary classifiers enhances the performance.

The disadvantage of this approach is as follows. As the method requires that all pairwise models be constructed beforehand, the method does not scale up well when the number of algorithms is large. One possible way to resolve this problem is by devising a method that determines the best possible pairwise test to carry out in each step. This method is discussed in the next section.

## 5.8 Active Testing

The algorithm selection methods, based on average ranking discussed in Chapter 2 suffer from one shortcoming. This is due to the fact that the schedule of tests is fixed. This has the disadvantage that, whenever a given set of algorithms includes many similar variants with similar performance, these could appear close to each other in the ranking. It does not make much sense to test them all in succession. It seems beneficial to try to use algorithms of different types that could hopefully yield better results. The average

ranking method, however, just follows the ranking, and hence is unable to skip very similar algorithms.

This problem is quite common, as similar variants can arise for instance when considering algorithms that include parameters, which may be set to different values. Quite often, many of the settings have a limited impact on performance. Even if we selected only some of all possible alternative parameter settings, we can end up with a large number of variants. Many of them exhibit rather similar performance. Clearly, it is desirable to have a more intelligent way of choosing algorithms from a given ranking. In other words, it is desirable to have a more intelligent schedule of tests. One method that does that is referred to as *active testing* (Leite et al., 2012; Abdulrahman et al., 2018) and is described in the next section. Two variants of the active testing method have been described in literature. The version AT of Leite et al. (2012) uses accuracy as the performance measure. The version AT\*, discussed by Abdulrahman et al. (2018), uses a combined measure of accuracy and runtime as the performance measure. As AT can be regarded as a special case of AT\*, where the effect of runtime is ignored, the focus here is on AT\*, which is more general.

### 5.8.1 Active testing that considers accuracy and runtime

This active testing method involves two important concepts, *current best algorithm*,  $a_{best}$ , and *best competitor*,  $a_c$ . The method starts by initializing  $a_{best}$  to the topmost algorithm in the average ranking AR\*. This algorithm can be run on the target dataset without problems.

It then selects new algorithms in an iterative fashion, searching in each step for the *best competitor*,  $a_c$ . This best competitor is identified by calculating the *estimated performance gain* of each untested algorithm with respect to  $a_{best}$ . In the work of Abdulrahman et al. (2018), the estimate of performance gain,  $\Delta P$ , was expressed in terms of A3R as follows:

$$\Delta P(a_j, a_{best}, d_i) = \max(A3R_{a_{best}, a_j}^{d_i} - 1, 0), \quad (5.11)$$

where  $A3R$  is defined as

$$A3R_{a_{ref}, a_j}^{d_i} = \frac{P_{a_j}^{d_i} / P_{a_{ref}}^{d_i}}{(T_{a_j}^{d_i} / T_{a_{ref}}^{d_i})Q}, \quad (5.12)$$

where  $P_{a_j}^{d_i}$  represents the performance (e.g., accuracy) of algorithm  $a_j$  on dataset  $d_i$ , and  $T_{a_j}^{d_i}$  the corresponding runtime. The term  $a_{ref}$  is a reference algorithm and  $Q$  is a scaling factor that controls the importance of runtime.<sup>5</sup>

In one recent work Leite and Brazdil (2021) have shown that the following definition of  $\Delta P$ , which is simpler than Eq. 5.11 shown before leads to better results:

$$\Delta P(a_j, a_{best}, d_i) = A3R_{a_{best}, a_j}^{d_i}. \quad (5.13)$$

An illustrative example is shown in Fig. 5.4, which shows different values of  $\Delta P$  for one competitor of  $a_{best}$  on all datasets. If another algorithm outperformed  $a_{best}$  on some datasets, it is regarded as a competitor. One important question here is to select the potentially best competitor. This is done by selecting the algorithm that maximizes the estimate of performance gain, as expressed by the following equation:

<sup>5</sup>The earlier variant of Leite et al. (2012) used just accuracy, without considering runtime. In this work,  $\Delta P$  was defined as a difference, rather than a ratio.

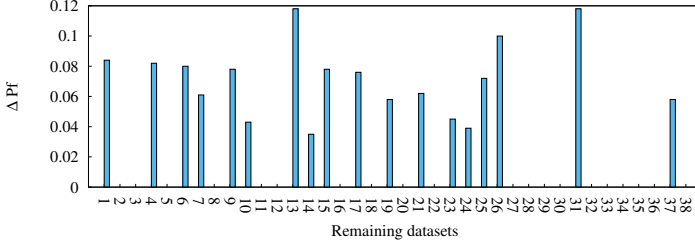


Fig. 5.4: Values of  $\Delta P$  of a potential competitor with respect to  $a_{best}$  on all datasets (reproduced from (Abdulrahman et al., 2018)).

**Require:** Target dataset  $d_{new}$ ; Top algorithm in  $AR^*$   $a_{best}$ ; Datasets  $D_s$ ;  
Set of algorithms  $A$ ; Parameter of importance of runtime  $Q$

1: Initialize the loss curve  $L_i \leftarrow ()$

2: Obtain the performance of  $a_{best}$  on dataset  $d_{new}$  using a CV test:

$$(P_{a_{best}}^{d_{new}}, T_{a_{best}}^{d_{new}}) \leftarrow CV(a_{best}, d_{new})$$

3: **while**  $|A| > 0$  **do**

4: Find the most promising competitor  $a_c$  of  $a_{best}$  using estimates of performance gain:

$$a_c = \operatorname{argmax}_{a_k} \sum_{d_i \in D_s} \Delta P(a_k, a_{best}, d_i)$$

5:  $A \leftarrow A - a_c$  (Remove  $a_c$  from  $A$ )

6: Obtain the performance of  $a_c$  on dataset  $d_{new}$  using a CV test:

$$(P_{a_c}^{d_{new}}, T_{a_c}^{d_{new}}) \leftarrow CV(a_c, d_{new})$$

$$L_i \leftarrow L_i + (P_{a_c}^{d_{new}}, T_{a_c}^{d_{new}})$$

7: Compare the performance of  $a_c$  with  $a_{best}$  and carry out updates:

8: **if**  $P_{a_c}^{d_{new}} > P_{a_{best}}^{d_{new}}$  **then**

9:  $a_{best} \leftarrow a_c, P_{a_{best}}^{d_{new}} \leftarrow P_{a_c}^{d_{new}}, T_{a_{best}}^{d_{new}} \leftarrow T_{a_c}^{d_{new}}$

10: **end if**

11: **end while**

12: **return** Loss-time curve  $L_i$  and  $a_{best}$

**Algorithm 5.1:** AT-A3R: active testing with A3R for dataset  $d_{new}$

$$a_c = \operatorname{argmax}_{a_k} \sum_{d_i \in D_s} \Delta P(a_k, a_{best}, d_i). \quad (5.14)$$

After the best competitor has been identified, the method proceeds with a test on the new dataset to obtain the actual performance of the best competitor. The aim is to determine whether the best competitor achieves a better performance than the current best algorithm,  $a_{best}$ . If it does, it becomes the new  $a_{best}$ . The reader can consult Algorithm 5.1 for a detailed description of the method (adapted from (Abdulrahman et al., 2018)).

In order to use AT-A3R in practice, it is necessary to determine a good setting of the parameter  $Q$  in A3R. The authors have experimented with different values and recommend the setting  $Q = 1/16$ . It appears that the setting in the range from  $P = 1$  to  $Q = 1/64$  does not affect the results much.

However, the setting of  $Q = 0$ , which is outside this range and represents a situation when only accuracy matters, makes a difference. This is illustrated in Figure 5.5 (reproduced from the work referred to above). The optimized version of AT is referred to as AT\*. The version AT0 uses the  $P = 0$  setting and represents the version that does not take runtime into account. We note that AT0 has far worse performance than AT\*. The similar loss is achieved at times much later. We note that the speed-up is huge: approximately two orders of magnitude!

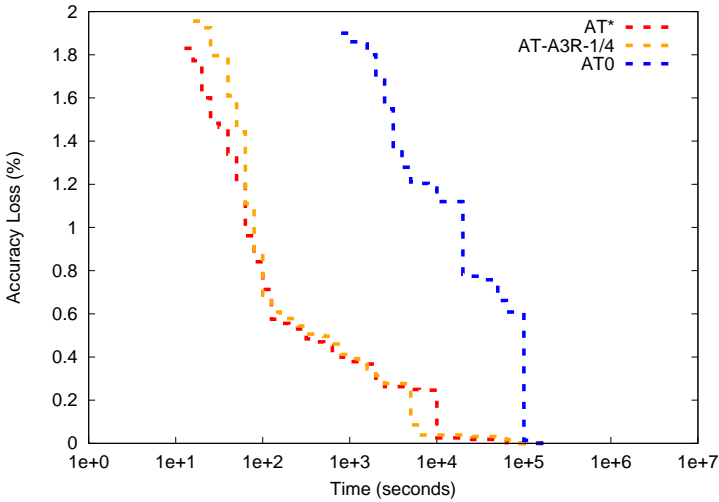


Fig. 5.5: Mean loss–time curves for different variants of AT-A3R

## Expected performance gains and relative landmarks

In previous work by Leite et al. (2012) the estimate of performance gain,  $\Delta P$  was referred to as the *relative landmark*,  $RL$ . The former term seems clearer than the second one, and this is why it has been adopted in this chapter. However, the relationship between the two concepts is interesting. All different dataset characteristics, including relative landmarks, are discussed in Chapter 4.

### 5.8.2 Active testing with focus on similar datasets

In the work of Leite et al. (2012), the performance gain was estimated by finding the *most similar datasets* and calculating the performance gain only for those datasets. For



the sake of simplicity of exposition, Algorithm 5.1 used all datasets without focusing on the most similar ones. However, it is not difficult to alter the method to focus only on relevant datasets. Let us examine some possibilities.

One possibility involves identifying similar datasets prior to invoking the AT algorithm (Algorithm 5.1). So the identification of similar datasets could be seen as a kind of preprocessing step of datasets. The similarity could be determined using a subset of dataset measures discussed in Chapter 4. These would necessarily have to exclude performance-based measures. This is because, if this was done prior to running any experiments on the target dataset, no performance-based data would be available. So, this approach has the disadvantage that it excludes the use of performance-based measures.

If we want to use performance-based measures, we need to incorporate the selection mechanism inside the AT algorithm. The best alternative seems to do this while searching for the algorithm (workflow) with the largest performance gain. This is captured by Equation 5.15.

$$a_c = \operatorname{argmax}_{a_k \in A_s} \sum_{d_i \in D_s} \Delta P(a_k, a_{best}, d_i) * \operatorname{Sim}(d_{new}, d_i), \quad (5.15)$$

where  $A_s$  is a given portfolio of algorithms (workflows) and  $\operatorname{Sim}(d_{new}, d_i)$  is an appropriate measure of similarity between the dataset  $d_{new}$  and dataset  $d_i$ . Various alternatives arise regarding how the similarity is established:

1. Similarity based on polarity of performance difference
2. Cosine-based similarity of performance results
3. Correlation-based similarity of performance results

More details about the first measure in the list above (similarity based on polarity of performance difference) are given in the next subsection. Further details about the other two measures are given in Chapter 4 (Section 4.10).

### Similarity based on polarity of performance difference

Leite et al. (2012) have described a variant of active testing referred to as *ATI*, where the similarity was based on performance (accuracy) difference. Suppose that, at some stage, a particular algorithm was identified as the current best candidate,  $a_{best-}$ . Suppose that later on other algorithms were examined and algorithm  $a_{best}$  was identified as the best one. This implies that  $\Delta P(a_{best}, a_{best-}, d_{new}) > 0$ , where  $\Delta P$  represents the difference of accuracy. Then, in the subsequent iteration, all prior datasets  $d_k$  satisfying a similar condition are considered similar to  $d_{new}$ . So, the similarity can be defined by

$$\operatorname{Sim}_{pol}(d_{new}, d_k) = \Delta P(a_{best}, a_{best-}, d_{new}) > 0 \ \& \ \Delta P(a_{best}, a_{best-}, d_k) > 0. \quad (5.16)$$

We note that this similarity takes into account the performance of two algorithms only. Despite this limitation, the authors reported a speed-up of 2 with respect to the basic version of AT (Leite et al., 2012).

## 5.8.3 Discussion

### Identifying best choice to test with acquisition functions

The process of identifying the best candidate algorithm to test in accordance with Eq. 5.11 can be compared to what *acquisition functions* do. In Bayesian optimization, the

acquisition function evaluates hyperparameter configurations based on their *expected utility* and selects the one with the highest utility. This issue is discussed in depth in Chapter 6.

## Using the AT method for workflow selection and configuration

In many practical applications it is not sufficient to focus on a selection of a single algorithm, but rather construct a workflow (pipeline) of operation. Although this issue is discussed in Chapter 7 (Section 7.4), in this section we give a brief overview of the work of Ferreira and Brazdil (2018), who have examined whether the AT method discussed here could be extended to recommendation of workflows for text classification. Their meta-database included nearly 20,000 workflows. They have shown that the active testing approach somewhat surpassed the average ranking. They also used meta-level analysis, whose aim was to determine importance of different elements of the workflows.

## Relationship of AT to reduction of configuration spaces

The AT method searches for the most competitive algorithms (workflows) in a given configuration space. Many non-competitive algorithms may not even be tried out. These algorithms are eliminated at run-time, as a side-effect of the AT search method.

This approach can be contrasted to another one, whose aim is to reduce the configuration space by eliminating non-competitive algorithms in a kind of preprocessing stage. That is, this operation can be carried out before invoking the AT method, or any other search method whose aim is to identify the best algorithm for the target dataset. This issue is addressed and discussed in detail in Chapter 8.

## 5.9 Non-propositional Approaches

The algorithms discussed so far are only able to deal with propositional representations of the metalearning problem. That is, they assume each meta-example is described by a fixed set of metafeatures,  $\mathbf{x} = (x_1, x_2, \dots, x_k)$ . However, the problem is non-propositional. On the one hand, the size of the set of dataset characteristics varies for different datasets (e.g., depending on the number of features). On the other hand, information about the algorithms can also be useful for metalearning (e.g., the interpretability of the generated models). In spite of this, there are very few approaches that use relational learning approaches.

One approach that exploits the non-propositional dataset description is FOIL, a well-known inductive logic programming (ILP) algorithm (Quinlan and Cameron-Jones, 1993). With FOIL, models can be induced that contain existentially quantified rules, such as “CN2 is applicable to datasets that contain a discrete feature with more than 2.3% missing values” (Todorovski and Džeroski, 1999).

A different approach uses a case-based reasoning tool, *CBR-Works Professional* (Lindner and Studer, 1999; Hilario and Kalousis, 2001). This can be viewed as a  $k$ -NN algorithm that not only allows a non-propositional description of datasets but also enables the use of information about the algorithms, independently of datasets. This work was extended later by the analysis of different distance measures for non-propositional representation (Kalousis and Hilario, 2003). Some of these measures enable the distance

between two datasets to be defined by a pair of individual features, e.g., the two features which are most similar in terms of one property such as *skewness*.

These papers usually compare their approaches against propositional methods. However, we have no knowledge of a comparison between different non-propositional methods.

## References

- Abdulrahman, S., Brazdil, P., van Rijn, J. N., and Vanschoren, J. (2018). Speeding up algorithm selection using average ranking and active testing by introducing runtime. *Machine Learning*, 107(1):79–108.
- Bensusan, H. and Giraud-Carrier, C. (2000). Discovering task neighbourhoods through landmark learning performances. In Zighed, D. A., Komorowski, J., and Zytgow, J., editors, *Proceedings of the Fourth European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD 2000)*, pages 325–330. Springer.
- Bensusan, H. and Kalousis, A. (2001). Estimating the predictive accuracy of a classifier. In Flach, P. and De Raedt, L., editors, *Proceedings of the 12th European Conference on Machine Learning*, pages 25–36. Springer.
- Blockeel, H., De Raedt, L., and Ramon, J. (1998). Top-down induction of clustering trees. In *Proceedings of the 15th International Conference on Machine Learning, ICML'98*, pages 55–63, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.
- Brazdil, P., Gama, J., and Henery, B. (1994). Characterizing the applicability of classification algorithms using meta-level learning. In Bergadano, F. and De Raedt, L., editors, *Proceedings of the European Conference on Machine Learning (ECML94)*, pages 83–102. Springer-Verlag.
- Brazdil, P., Soares, C., and da Costa, J. P. (2003). Ranking learning algorithms: Using IBL and meta-learning on accuracy and time results. *Machine Learning*, 50(3):251–277.
- Cohen, W. W. (1995). Fast effective rule induction. In Prieditis, A. and Russell, S., editors, *Proceedings of the 12th International Conference on Machine Learning, ICML'95*, pages 115–123. Morgan Kaufmann.
- Dimitriadou, E., Hornik, K., Leisch, F., Meyer, D., and Weingessel, A. (2004). e1071: Misc functions of the Department of Statistics (e1071), R package version 1.5-1. Technical report, TU Wien.
- Domhan, T., Springenberg, J. T., and Hutter, F. (2015). Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*.
- Eggensperger, K., Lindauer, M., Hoos, H., Hutter, F., and Leyton-Brown, K. (2018). Efficient benchmarking of algorithm configuration procedures via model-based surrogates. *Special Issue on Metalearning and Algorithm Selection, Machine Learning*, 107(1).
- Ferreira, M. and Brazdil, P. (2018). Workflow recommendation for text classification with active testing method. In *Workshop AutoML 2018 @ ICML/IJCAI-ECAI*. Available at site <https://sites.google.com/site/automl2018icml/accepted-papers>.
- Feurer, M., Eggensperger, K., Falkner, S., Lindauer, M., and Hutter, F. (2018). Practical automated machine learning for the AutoML challenge 2018. In *International Workshop on Automatic Machine Learning at ICML2018*, pages 1189–1232.
- Feurer, M., Springenberg, J. T., and Hutter, F. (2014). Using meta-learning to initialize Bayesian optimization of hyperparameters. In *ECAI Workshop on Metalearning and Algorithm Selection (MetaSel)*, pages 3–10.

- Fürnkranz, J. and Petrak, J. (2001). An evaluation of landmarking variants. In Giraud-Carrier, C., Lavrač, N., and Moyle, S., editors, *Working Notes of the ECML/PKDD 2000 Workshop on Integrating Aspects of Data Mining, Decision Support and Meta-Learning*, pages 57–68.
- Gama, J. and Brazdil, P. (1995). Characterization of classification algorithms. In Pinto-Ferreira, C. and Mamede, N. J., editors, *Progress in Artificial Intelligence, Proceedings of the Seventh Portuguese Conference on Artificial Intelligence*, pages 189–200. Springer-Verlag.
- Hilario, M. and Kalousis, A. (2001). Fusion of meta-knowledge and meta-data for case-based model selection. In Siebes, A. and De Raedt, L., editors, *Proceedings of the Fifth European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD01)*. Springer.
- Hutter, F., Xu, L., Hoos, H., and Leyton-Brown, K. (2014). Algorithm runtime prediction: Methods and evaluation. *Artificial Intelligence*, 206:79–111.
- Jamieson, K. and Talwalkar, A. (2016). Non-stochastic best arm identification and hyperparameter optimization. In *Artificial Intelligence and Statistics*, pages 240–248.
- Kalousis, A. (2002). *Algorithm Selection via Meta-Learning*. PhD thesis, University of Geneva, Department of Computer Science.
- Kalousis, A. and Hilario, M. (2000). Model selection via meta-learning: A comparative study. In *Proceedings of the 12th International IEEE Conference on Tools with AI*. IEEE Press.
- Kalousis, A. and Hilario, M. (2003). Representational issues in meta-learning. In *Proceedings of the 20th International Conference on Machine Learning, ICML'03*, pages 313–320.
- Kolodner, J. (1993). *Case-Based Reasoning*. Morgan Kaufmann Publishers.
- Köpf, C., Taylor, C., and Keller, J. (2000). Meta-analysis: From data characterization for meta-learning to meta-regression. In Brazdil, P. and Jorge, A., editors, *Proceedings of the PKDD 2000 Workshop on Data Mining, Decision Support, Meta-Learning and ILP: Forum for Practical Problem Presentation and Prospective Solutions*, pages 15–26.
- Leake, D. B. (1996). *Case-Based Reasoning: Experiences, Lessons & Future Directions*. AAAI Press.
- Leite, R. and Brazdil, P. (2004). Improving progressive sampling via meta-learning on learning curves. In Boulicaut, J.-F., Esposito, F., Giannotti, F., and Pedreschi, D., editors, *Proc. of the 15th European Conf. on Machine Learning (ECML2004)*, LNAI 3201, pages 250–261. Springer-Verlag.
- Leite, R. and Brazdil, P. (2005). Predicting relative performance of classifiers from samples. In *Proceedings of the 22nd International Conference on Machine Learning, ICML'05*, pages 497–503, NY, USA. ACM Press.
- Leite, R. and Brazdil, P. (2007). An iterative process for building learning curves and predicting relative performance of classifiers. In *Proceedings of the 13th Portuguese Conference on Artificial Intelligence (EPIA 2007)*, pages 87–98.
- Leite, R. and Brazdil, P. (2010). Active testing strategy to predict the best classification algorithm via sampling and metalearning. In *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI)*, pages 309–314.
- Leite, R. and Brazdil, P. (2021). Exploiting performance-based similarity between datasets in metalearning. In Guyon, I., van Rijn, J. N., Treguer, S., and Vanschoren, J., editors, *AAAI Workshop on Meta-Learning and MetaDL Challenge*, volume 140, pages 90–99. PMLR.

- Leite, R., Brazdil, P., and Vanschoren, J. (2012). Selecting classification algorithms with active testing. In *Machine Learning and Data Mining in Pattern Recognition*, pages 117–131. Springer.
- Leyton-Brown, K., Nudelman, E., and Shoham, Y. (2009). Empirical hardness models: Methodology and a case study on combinatorial auctions. *Journal of the ACM*, 56(4).
- Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., and Talwalkar, A. (2017). Hyperband: Bandit-Based Configuration Evaluation for Hyperparameter Optimization. In *Proc. of ICLR 2017*.
- Lindner, G. and Studer, R. (1999). AST: Support for algorithm selection with a CBR approach. In Giraud-Carrier, C. and Pfahringer, B., editors, *Recent Advances in Meta-Learning and Future Work*, pages 38–47. J. Stefan Institute.
- Mohr, F. and van Rijn, J. N. (2021). Towards model selection using learning curve cross-validation. In *8th ICML Workshop on Automated Machine Learning (AutoML)*.
- Pfahringer, B., Bensusan, H., and Giraud-Carrier, C. (2000). Meta-learning by landmarking various learning algorithms. In Langley, P., editor, *Proceedings of the 17th International Conference on Machine Learning*, ICML'00, pages 743–750.
- Pfisterer, F., van Rijn, J. N., Probst, P., Müller, A., and Bischl, B. (2018). Learning multiple defaults for machine learning algorithms. *arXiv preprint arXiv:1811.09409*.
- Provost, F., Jensen, D., and Oates, T. (1999). Efficient progressive sampling. In Chaudhuri, S. and Madigan, D., editors, *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM.
- Quinlan, J. (1992). Learning with continuous classes. In Adams and Sterling, editors, *AI'92*, pages 343–348. Singapore: World Scientific.
- Quinlan, R. (1998). *C5.0: An Informal Tutorial*. RuleQuest. <http://www.rulequest.com/see5-unix.html>.
- Quinlan, R. and Cameron-Jones, R. (1993). FOIL: A midterm report. In Brazdil, P., editor, *Proc. of the Sixth European Conf. on Machine Learning*, volume 667 of LNAI, pages 3–20. Springer-Verlag.
- Rasmussen, C. and Williams, C. (2006). *Gaussian Processes for Machine Learning*. The MIT Press.
- Soares, C., Petrak, J., and Brazdil, P. (2001). Sampling-based relative landmarks: Systematically test-driving algorithms before choosing. In Brazdil, P. and Jorge, A., editors, *Proceedings of the 10th Portuguese Conference on Artificial Intelligence (EPIA2001)*, pages 88–94. Springer.
- Sohn, S. Y. (1999). Meta analysis of classification algorithms for pattern recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21(11):1137–1144.
- Sun, Q. and Pfahringer, B. (2012). Bagging ensemble selection for regression. In *Proceedings of the 25th Australasian Joint Conference on Artificial Intelligence*, pages 695–706.
- Sun, Q. and Pfahringer, B. (2013). Pairwise meta-rules for better meta-learning-based algorithm ranking. *Machine Learning*, 93(1):141–161.
- Thornton, C., Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2013). Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 847–855. ACM.
- Todorovski, L., Blockeel, H., and Džeroski, S. (2002). Ranking with predictive clustering trees. In Elomaa, T., Mannila, H., and Toivonen, H., editors, *Proc. of the 13th European Conf. on Machine Learning*, number 2430 in LNAI, pages 444–455. Springer-Verlag.
- Todorovski, L. and Džeroski, S. (1999). Experiments in meta-level learning with ILP. In Rauch, J. and Zytkow, J., editors, *Proceedings of the Third European Conference on*

- Principles and Practice of Knowledge Discovery in Databases (PKDD99)*, pages 98–106. Springer.
- Tsuda, K., Rätsch, G., Mika, S., and Müller, K. (2001). Learning to predict the leave-one-out error of kernel based classifiers. In *ICANN*, pages 331–338. Springer-Verlag.
- van Rijn, J. N. (2016). *Massively collaborative machine learning*. PhD thesis, Leiden University.
- van Rijn, J. N., Abdulrahman, S., Brazdil, P., and Vanschoren, J. (2015). Fast algorithm selection using learning curves. In *International Symposium on Intelligent Data Analysis XIV*, pages 298–309.

---

## Metalearning for Hyperparameter Optimization

**Summary.** This chapter describes various approaches for the hyperparameter optimization (HPO) and combined algorithm selection and hyperparameter optimization problems (CASH). It starts by presenting some basic hyperparameter optimization methods, including grid search, random search, racing strategies, successive halving and hyperband. Next, it discusses Bayesian optimization, a technique that learns from the observed performance of previously tried hyperparameter settings on the current task. This knowledge is used to build a meta-model (surrogate model) that can be used to predict which unseen configurations may work better on that task. This part includes the description *sequential model-based optimization* (SMBO). This chapter also covers metalearning techniques that extend the previously discussed optimization techniques with the ability to transfer knowledge across tasks. This includes techniques such as *warm-starting* the search, or *transferring previously learned meta-models* that were trained on prior (similar) tasks. A key question here is how to establish how similar prior tasks are to the new task. This can be done on the basis of past experiments, but can also exploit the information gained from recent experiments on the target task. This chapter presents an overview of some recent methods proposed in this area.

### 6.1 Introduction

Many machine learning algorithms include various hyperparameters that greatly affect their performance (Lavesson and Davidsson, 2006). These hyperparameters can be *numeric*, e.g., the gradient descent learning rate in a neural network, but also *categorical*, e.g., the choice of kernel in an SVM, and some hyperparameters are also *conditional* on the value of other hyperparameters, e.g., when choosing the Gaussian kernel for an SVM, one also needs to choose the kernel width (i.e.,  $\gamma$ ).

The effect of hyperparameter configurations on performance can be very complex and greatly dependent on the properties of the dataset at hand. Hence, we want to *learn* – based on prior experimentation – which configurations are likely to work better than others on a particular dataset, or across a group of datasets. Such experience is partly encoded in the *default hyperparameter settings* provided by algorithm designers,

---

<sup>1</sup>In the first edition, this topic was covered only briefly in Section 2.4, focusing mainly on metalearning approaches used to determine the potentially best parameter settings.

but these will seldom be optimal for a newly given task. Some illustrative examples of this are shown in Chapter 17.

The task of optimizing these hyperparameter settings for a particular task is known as *hyperparameter optimization* (HPO) or *algorithm configuration* (AC).

Algorithm selection (discussed in the previous chapter) can be seen as a special (discrete) form of HPO, simply by encoding the choice of algorithm as an additional hyperparameter (Thornton et al., 2013). That also means that one can optimize the choice of algorithms and their hyperparameters at the same time, also known as *combined algorithm selection and hyperparameter optimization* (CASH). Even more generally, one could define a hyperparameter search space that includes all possible design decisions involved in building a learning model, including the architecture of neural networks or the structure of machine learning pipelines (covered in the next chapter). Since the goal here is to completely automate the process of designing and training machine learning models, this is called *automatic machine learning* (AutoML).

In practice, turning every design decision into a new hyperparameter leads to an explosion of the search space. The larger and more complex the search space becomes, the harder it becomes to optimize it effectively, and the longer it may take until a satisfactory model is found.

In Chapter 8 we discuss general principles that can be followed in the design of the search spaces. This chapter also discusses some methods that can be used in the process of redesigning these spaces on the basis of experience. This is done on the basis of experience with different tasks.

In this chapter, we explore how *metalearning* can allow us to learn from past experimentation and leverage this prior experience to design algorithms and optimize hyperparameters more effectively. Much like a machine learning expert learns through trial and error how to design and optimize models for new tasks, the aim is to learn across tasks to make informed decisions about how to design and tune the best machine learning models.

### 6.1.1 Overview of this chapter

In Section 6.2, we start by presenting some basic concepts and then cover the basic hyperparameter optimization methods, which do not use metalearning, but form the basis for subsequent methods. These include grid search, random search, racing strategies, evolutionary methods, best-first search, and search with an elimination strategy, which is followed, for instance, in Hyperband.

Next, Section 6.3 focuses on Bayesian optimization, a technique that learns from the observed performance of previously tried hyperparameter settings on the current task to build a meta-model (surrogate model) that can be used to predict which unseen configurations may work better on that task. This section includes the description of the approach known under the name *sequential model-based optimization* (SMBO).

Section 6.4 covers metalearning techniques that extend the previously discussed optimization techniques with the ability to transfer knowledge across tasks. This includes techniques such as *warm-starting* the search for the best hyperparameter with configurations that worked well before, *learning a probability distribution* (a prior) of the best hyperparameter configurations based on previous tasks, or *transferring previously learned meta-models* that were trained on prior (similar) tasks. A key question here is to establish how similar prior tasks are to the new task, since the metaknowledge obtained from very similar tasks is likely much more useful. This can be done on the basis of past



experiments and the accompanying metadata, which include measurable data characteristics (see Chapter 4). Alternatively, this can be based on novel knowledge gained by new experimentation on the new task itself and by observing that the hyperparameter configurations tried on the new task behave similarly as on some previous tasks. Finally, Section 6.5 closes with concluding remarks.

## 6.2 Basic Hyperparameter Optimization Methods

### 6.2.1 Basic concepts

Let us first describe the task of hyperparameter optimization formally. Let  $M(a, \theta, d_{train})$  represent a trained model  $M$  generated by a particular algorithm  $a$  with hyperparameter configuration  $\theta$  on the training portion of the target dataset  $d$ ,  $d_{train}$ . Let  $A(M(a, \theta, d_{train}), X_{val})$  represent the application of the trained model to the validation data  $X_{val}$ , which returns a set of predictions. The output of  $A(\cdot)$  varies as  $\theta$  is varied. Then, the loss  $L$  can be determined using a given loss function  $\mathcal{L}$ :

$$L = \mathcal{L}(A(M(a, \theta, d_{train}), X_{val}), y_{val}). \quad (6.1)$$

Sometimes it is convenient to use the following short form of the loss function, which only includes the input arguments, namely  $\mathcal{L}(a, \theta, d_{train}, d_{val})$ . Whenever the algorithm  $a$  and the dataset  $d$  and train–test splits are fixed, we can simply use  $\mathcal{L}(a)$ . The aim of CASH is then to determine the values of  $a$  and  $\theta$  from the respective sets of all algorithms  $A$  and all possible configurations  $\Theta$  that minimize the loss.

$$(a^*, \theta^*) = \underset{\theta \in \Theta, a \in A}{\operatorname{argmin}} \mathcal{L}(a, \theta, d_{train}, d_{val}). \quad (6.2)$$

Since algorithm selection can be formulated as hyperparameter selection through a categorical variable representing the choice of algorithm, the pair  $(a^*, \theta^*)$  in the equation above can be substituted by a single hyperparameter  $(\theta^*)$ . The evaluation of various values of  $\theta$  gives rise to a *observation history*  $H$  of losses across hyperparameter settings. It is of the form

$$H_{\theta, L} \equiv (\theta_i, L_i)^n. \quad (6.3)$$

The observation history  $H$  can be part of the *metadata*  $MetaD$  discussed in Chapter 5. It can be stored for both prior tasks and the current task and leveraged in different ways, as we will discuss in this chapter.

### 6.2.2 Basic optimization methods

#### Grid search

One simple method to find  $\theta^*$  is *grid search*, which searches exhaustively through a predefined set of hyperparameter values of a given algorithm. It requires that the space of alternatives,  $\Theta$ , is identified and discretized beforehand. This involves identifying the hyperparameters that should be considered. Some have categorical values (e.g., the type of the SVM kernel), while others are real-valued. The latter need to be discretized and the resulting values provided to the system. Figure 6.1 (left) illustrates this. We note that

the choice of hyperparameter values may be conditional on other choices. For instance, if the SVM kernel is *Gaussian*, it makes sense to also tune the *kernel width*.

After the different hyperparameter configurations have been defined, the performance of the given algorithm is evaluated for each configuration. Finally, the configuration with the best performance,  $\theta^*$ , is returned.

Many machine learning libraries use a grid search (or other simple search method) internally and return a configuration that is often better than the default configuration (i.e., the one with default settings). The grid is normally predefined by the designer and includes a relatively small number of values. This is done to limit the search and the corresponding time spent in the search. For instance, the *caret* package (Kuhn, 2008, 2018) runs a predefined grid search with various machine learning algorithms. So we can say that these systems perform a rudimentary form of hyperparameter optimization in an autonomous manner.

### Random search

Random search explores the space of configuration randomly. As in the previous case, the space of alternatives  $\Theta$  needs to be determined beforehand. However, it is not necessary to discretize real-valued hyperparameters. All that is needed is to provide the interval from which the values should be sampled and the type of distribution that should be followed by the sampling process (e.g., uniform).

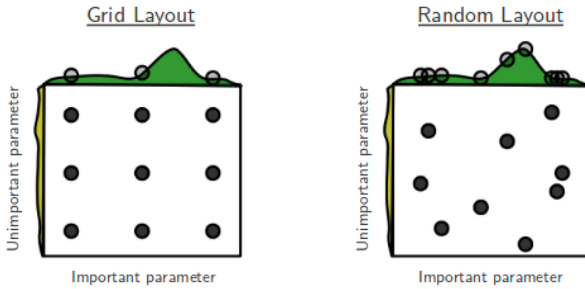


Fig. 6.1: Conceptual difference between random search and grid search. Image taken from Bergstra and Bengio (2012)

Bergstra and Bengio (2012) argue that random search has several advantages over grid search. First, grid search does not scale well with the number of hyperparameters. Adding one hyperparameter can have an exponential influence on the number of points that need to be evaluated. Second, grid search might very well miss the global optimum, as the discretization can remove it from the search space. Finally, when some hyperparameters happen to be irrelevant, i.e. they have little effect on performance, random search often converges to a better configuration. In order to illustrate this, consider Figure 6.1. In this example, we try to optimize two hyperparameters: an important hyperparameter on the horizontal axis, and an unimportant hyperparameter on the vertical axis

axis. The important hyperparameter has an effect on the performance of the algorithm, while the unimportant one has no such effect. In this example, random search still explores nine different values in the range of the important hyperparameter, whereas grid search only explores three. For this reason, random search has a higher chance of finding a better configuration.

## Enhancing search with racing methods

Random search, as well as more sophisticated search methods, can be sped up by using stochastic optimization techniques such as *racing* (Hutter et al., 2011; López-Ibáñez et al., 2011, 2016).

To compute the loss  $L$ , one often uses a cross-validation mechanism (see Chapter 3) that evaluates each hyperparameter configuration on multiple train–test splits (folds) to determine the one with the highest average performance across all folds. After evaluating multiple configurations  $\theta_i$  in parallel on several of these folds, some configurations may show suboptimal performance and hence have a low probability of beating the best configuration identified so far. Consequently, these configurations can be eliminated from further consideration without evaluating them on all folds. In the case of large datasets, one can also use racing of multiple configurations on a single train–test split by letting them predict individual test instances on increasingly more training data until they are statistically unlikely to be optimal.

Various algorithms implement such racing strategies. Originally, it was used to speed up the search for informative feature subsets in classification (Moore and Lee, 1994; John et al., 1994). It was later incorporated into various approaches whose aim is identify the best algorithm configurations. *ROAR* is an extension of random search that uses a rather aggressive version of *racing* to drop candidates (Hutter et al., 2011). *Trace* embodies a more conservative racing strategy; it performs a statistical test to determine whether a particular configuration can be dropped. This happens when the test indicates that this configuration has a very low chance of beating the more promising candidates (López-Ibáñez et al., 2011, 2016).

### 6.2.3 Evolutionary methods

Evolutionary algorithms and population-based methods are also often employed to optimize hyperparameters, because they can optimize many hyperparameters simultaneously, while providing more direction than basic random search. Popular approaches include genetic algorithms (Reif et al., 2012), evolution strategies (Hansen, 2006), tabu search (Gomes et al., 2012), and particle swarm optimization (de Miranda et al., 2012).

One of the most successful techniques is CMA-ES (Hansen, 2006), a population-based method that evaluates a set of randomly sampled configurations, selects the best one, and then iteratively samples new configurations *around* the current best one until it converges. Loshchilov and Hutter (2016) used CMA-ES to optimize the hyperparameters of neural networks.

### 6.2.4 Heuristic search methods

*Heuristic search methods* (Russell and Norvig, 2016), such as hill climbing and best-first methods, involve a *heuristic function* that attributes a heuristic value to a given state.

They can be used to optimize hyperparameters by viewing the hyperparameter space as a multi-dimensional space, where each point in this space is associated with a specific heuristic score: the performance of that configuration.

These methods traverse this space by moving to a *neighboring configuration* that has the highest score. That raises the question of which configurations are the neighbors of the current configuration. This could, for example, be all configurations that have one hyperparameter value changed to a different value.

As hill climbing methods can get stuck on local optima, some researchers have used so-called *diversification methods*, such as *restarts* and *random steps*, to avoid getting stuck at local minima. Some examples of works that use similar techniques are *iterated local search* (Lourenço et al., 2003) and ParamILS (Hutter et al., 2009).

Gradient descent methods can be used for adjusting numeric hyperparameters. It assumes that the best hyperparameter setting can be identified by following the gradient of the loss function. Many hyperparameters of machine learning algorithms do not satisfy this assumption, and hence this method can also get stuck at local minima. Maclaurin et al. (2015) compute the gradients of cross-validation performance with respect to all hyperparameters by chaining derivatives.

### 6.2.5 Hypergradients

When the learning algorithm uses stochastic gradient descent to optimize its model parameters (weights), as in neural networks, it is possible to also optimize certain numeric hyperparameters through gradient descent. Indeed, one can take the derivative of the used loss function  $\mathcal{L}$  with respect to certain hyperparameters (e.g., the learning rate), which also appear in the loss function. This is also called a *hypergradient*. Hence, we can use (hyper)gradient descent steps to optimize these hyperparameters bit by bit (Maclaurin et al., 2015; Baydin et al., 2018). Since computing the validation loss requires an inner loop of optimizing the model parameters (weights), this method is quite expensive, but it may still be useful when optimizing many hyperparameters simultaneously with parallel computational resources.

### 6.2.6 Multi-fidelity techniques

To speed up the search for the best hyperparameter configuration, it is possible to not evaluate every configuration on the entire dataset, which is expensive on larger datasets, but rather to evaluate many configurations on small samples of the training set, and only evaluate the best ones on more training data. Since the performance evaluated on small samples only gives a rough indication of the performance on the full training set, we require an optimization method that can handle noisy probabilistic rewards, such as *multi-armed bandit methods*, which are discussed in detail in Chapter 8 (Section 8.9). The aim of these methods is to solve the following problem: suppose there are various alternatives to be explored, each with an associated cost and the probability of a certain reward, then which alternatives should be explored to maximize the reward? When dealing with HPO, the cost is processing time and the reward is the performance measured (e.g., accuracy).

### Successive halving (SH)

Successive halving (SH) (Jamieson and Talwalkar, 2016; Li et al., 2017) is a multi-armed bandit (MAB) method. It conducts basically best-first search of a given initial number

of alternative configurations. Typically, this number would be quite high. The method differs from ordinary best first in that it includes a *budget* (e.g., computation time) that limits the exploration of each alternative.

It initiates a relatively large pool of candidate configurations that are allocated a low budget. After the budget has been exhausted, the method interrupts and all the nodes (configurations) are ordered by their respective performance (e.g., accuracy). The configurations in the bottom half of this list are eliminated. The search then continues only with the remaining nodes with their budget duplicated. This process continues until only one configuration has been obtained. By running a configuration with increasing budget, implicitly a learning curve is created. Algorithm 6.1 shows the details. While

```

input :  $\Theta$  space of hyperparameters of algorithm  $a$ 
          $n_{init}$  - initial number of alternatives
          $b_{init}$  - initial budget
output:  $\theta_{best}$  - algorithm configuration with the best performance
begin
   $\Theta' \leftarrow \text{Sample.uniformly}(\Theta, n_{init})$ 
   $b_c \leftarrow b_{init}$ 
   $n_c \leftarrow n_{init}$ 
  while  $n_c \geq 2$  do
    Run all configurations in  $\Theta'$  with budget  $b_c$ 
     $b_c \leftarrow b_c \times 2$ 
     $n_c \leftarrow n_c \div 2$ 
     $\Theta' \leftarrow \text{Select.best}(\Theta', n_c)$ 
  end
end

```

**Algorithm 6.1:** Successive halving (based on Jamieson and Talwalkar (2016))

many multi-armed bandit methods work by finding a trade-off between exploration and exploitation, SH is a full exploration method.

Regarding the nature of the budget, various alternatives have been suggested (Li et al., 2017): Apart from *runtime*, which was mentioned already, one can consider also the *number of steps*, which obviously affects runtime. It is possible to also consider different settings of some hyperparameter, such as the *number of epochs* in NNs or the *number of trees* in a random forest, that are also correlated with runtime.

## Hyperband and extensions to *successive halving* (SH)

One drawback of SH is that the outcome depends on the selection of an initial budget. Also, if the optimal configuration only performs well after a certain budget has been explored, e.g., a certain number of training examples are given, it can happen that it is prematurely eliminated.

Hyperband (Li et al., 2017) is a method that aims to address these issues, by running SH multiple times, each time with a higher initial budget but fewer initial alternatives. The final run of successive halving is in fact an emulation of random search, where a small number of configurations (arms) is run on a single budget, i.e., the maximum

budget. Hyperband is accompanied with several theoretical guarantees: one will never spend more than a log-factor of time more than random search to obtain the same result.

Both SH and Hyperband are very simple yet powerful methods. However, they do not exploit any meta-knowledge obtained on either the current, or other datasets. Some works have attempted to do this. Baker et al. (2017) train a learning curve model on the initial performance data to predict whether the performance of a new alternative would exceed the performance of the incumbent. If the model gives a negative prediction, the alternative is eliminated, resulting in faster execution.

van Rijn and Hutter (2018) propose to sample hyperparameter values that performed well on other datasets more frequently. Falkner et al. (2018) suggested a similar method, but only using the configurations that were obtained from the current dataset. This method is called *Bayesian Optimization with HyperBand (BOHB)* and combines good properties of both paradigms.

## 6.3 Bayesian Optimization

The term *Bayesian optimization*, first proposed by Mockus et al. (1978), refers to a black-box optimization method that places a prior over the function. The prior models capture a certain belief about the behavior of the function (Brochu et al., 2010). The methods described in the following subsection follow this basic strategy.

### 6.3.1 Sequential model-based optimization

Model-based search employs functional meta-level models in the search for the best configuration of hyperparameters of a given algorithm. Unlike both grid and random approaches, they take into account the results of previous evaluations. This approach is known under the name *sequential model-based optimization (SMBO)*. Algorithm 6.2, which is based on the work of Hutter et al. (2011), summarizes this method.

The aim is to find the optimal configuration  $\theta_{best}$  that minimizes the loss, as defined in Eq. 6.2. In order to model the prior with our beliefs about the behavior of the function, SMBO employs a model  $M_L$  that captures the dependence of loss on hyperparameter settings. Loosely speaking, this surrogate model expresses the probability function  $p(y|\theta)$  (Bergstra et al., 2011), where  $y$  is the expected performance of configuration  $\theta$ . Therefore, the surrogate model requires both good predictive power as well as reliable uncertainty estimates. This model is sometimes referred to as a *surrogate model*. The model  $M_L$  is used to determine a promising candidate configuration  $\theta_{new}$  which is used to conduct a test to determine the loss. The value  $\theta_{new}$  together with the corresponding loss are used to update the model  $M_L$ . These two steps are carried out in an iterative fashion.

### Acquisition function

In order to generate the next hyperparameter configuration, the method employs a so-called *acquisition function*,  $a_{M_L}$ . Several different acquisition functions were proposed in the past (Wistuba, 2018). These include, for instance:

- Probability improvement (PI) (Mockus et al., 1978);
- Expected improvement (EI) (Kushner, 1964; Jones et al., 1998);

```

input :  $a$  - algorithm whose hyperparameters are to be optimized
          $\Theta$  - space of hyperparameters
          $d$  - target dataset
output:  $\theta_{best}$  - algorithm configuration with the best performance
begin
   $H_{\theta,L} \leftarrow \text{InitialRandomTests}(a, \Theta, d)$ 
  Initialize model  $M_L$ 
   $(\theta_{best}, L_{best}) \leftarrow \text{SelectBest}(H_{\theta,L})$ 
  while Not converged and Time budget not exhausted do
     $\theta_{new} \leftarrow \text{GenConfig}(M_L)$ 
     $L_{new} \leftarrow \mathcal{L}(a, \theta, d_{train}, d_{val})$ 
    if  $L_{new} < L_{best}$  then
      |  $(\theta_{best}, L_{best}) \leftarrow (\theta_{new}, L_{new})$ 
    end
     $H_{\theta,L} \leftarrow (\theta_{new}, L_{new}) \cup H_{\theta,L}$  (update history)
     $M_L \leftarrow \text{update}(M_L, H_{\theta,L})$ 
  end
end

```

**Algorithm 6.2:** Sequential model-based optimization

- Entropy search (MacKay, 1992);
- Lower/upper confidence bounds, UCB (Cox and John, 1997; Srinivas et al., 2010).

Here we focus on EI, which is used in various systems, e.g., SMAC (Hutter et al., 2011), Auto-WEKA (Thornton et al., 2013), and Auto-sklearn (Feurer et al., 2015a). The aim is to identify the hyperparameter configuration  $\theta$  that will most likely have the lowest loss. Good candidate combinations are the ones with both high predicted value (low loss) and high uncertainty. This can be captured by the following equation:

$$I_{L_{min}}(\theta) = \max\{L_{min} - L(\theta), 0\}. \quad (6.4)$$

As the value of  $L(\theta)$  is not known, Thornton et al. (2013) suggest to calculate the expectation:

$$\mathbb{E}_{L_{min}}[I_{L_{min}}(\theta)] = \int_{-\infty}^{L_{min}} \max\{L_{min} - L(\theta), 0\} \cdot p_{M_L}(L|\theta) d\theta. \quad (6.5)$$

The form of the acquisition function depends on the underlying models of the loss function. The most common ones are based on *Gaussian processes* (GPs) and *random forests*. More details about both types are given in the following subsections.

## Gaussian processes as surrogate models of loss

*Gaussian processes* (GPs) (Rasmussen and Williams, 2006) have been commonly used by various researchers as surrogate models to model the loss function (e.g., Mockus et al. (1978); Bergstra et al. (2011); Hutter et al. (2011); Snoek et al. (2012); Wistuba (2018)).

The model  $M_L$  in Eq. 6.5 is modeled as a posterior GP, given the observation history  $H$ . The distribution  $p(\mathbf{L}|\boldsymbol{\theta})$  is assumed to be distributed as a multivariate Gaussian

$$\mathcal{N}(\mathbf{L}|m(\boldsymbol{\theta}), k(\boldsymbol{\theta}, \boldsymbol{\theta}), ) \quad (6.6)$$

where  $m(\boldsymbol{\theta})$  represents its mean function and  $K = k(\boldsymbol{\theta}, \boldsymbol{\theta})$  is a kernel matrix that captures covariance. To simplify matters, the mean  $m(\boldsymbol{\theta})$  is often set to 0.

Gaussian process assumption states that  $\mathbf{L}$  and  $\mathbf{L}^*$  are jointly Gaussian. In other words, Gaussian processes are *closed* under sampling (Bergstra et al., 2011). The predictive posterior distribution  $p(\mathbf{L}^*|\boldsymbol{\theta}, \mathbf{L}, \boldsymbol{\theta}^*)$  can be obtained from the joint distribution.

Bayesian optimization requires that the Gaussian process is frequently updated. Updating it every time from scratch is computationally expensive and dominated by the inversion of the kernel matrix. This operation has a cubic complexity with the number of training instances, i.e.,  $|H|$ . As Wistuba (2018) has shown, the update can be reduced to square complexity.

## Random forests as surrogate models of loss

Some researchers preferred *random forest* (RF) models (Breiman, 2001), as they perform well with discrete and high-dimensional data (Thornton et al., 2013). They provide quite accurate predictions and are also fast to train. They have been employed in various systems, including, e.g., SMAC (Thornton et al., 2013) and some predecessor systems (Hutter et al., 2011).

These systems employ a random forest to calculate a predictive mean  $\mu_\theta$  and variance  $\sigma_\theta$  on the basis of frequentist estimates for  $p(L|\theta)$ . So  $p_{ML}(L|\theta)$  is modeled as a Gaussian  $\mathcal{N}(\mu_\theta, \sigma_\theta)$ . The authors show that the expectation can be computed using a closed-form expression:

$$\mathbb{E}_{L_{min}}[I_{L_{min}}(\theta)] = \sigma_\theta * [u * \Phi(u) + \phi(u)], \quad (6.7)$$

where  $u = \frac{(L_{min} - \mu_\theta)}{\sigma_\theta}$ ,  $\phi$  represents the probability density function and  $\Phi$  the cumulative density function of a normal distribution.

## Note on previous approaches

Previous SMBO methods (see, e.g., Bartz-Beielstein et al. (2005); Hutter et al. (2009)) applied random sampling for the task of finding a new configuration. However, this is not very efficient, particularly in high-dimensional configuration spaces. This led Hutter et al. (2011) to adopt another approach. The method is referred to as *multi-start local search*. Certain locally maximal configurations are gathered and then used in a local search, in which the value of one parameter is varied.

### 6.3.2 Tree-structured Parzen estimator (TPE)

Instead of using a probabilistic regression model as a surrogate model, one can also use kernel density estimation, leading to a *tree-structured Parzen estimator* (Bergstra et al., 2011). The TPE method defines two probability distributions over the hyperparameter space:

$$p(\theta|y) = \begin{cases} \ell(\theta), & \text{if } y < y^* \\ g(\theta) & \text{if } y \geq y^* \end{cases} \quad (6.8)$$

where  $\ell(\theta)$  defines the density function over all points with a loss lower than the threshold  $y^*$  (the distribution of “good” configurations), and  $g(\theta)$  defines the density function



over all points with a loss higher than the given threshold (“bad” configurations). Assuming that we want to minimize a given measure, we would intuitively want to sample from distribution  $\ell(\theta)$ . However, there are better options.

The authors suggested to evaluate the configurations that maximize the ratio between  $\ell(\theta)$  and  $g(\theta)$ . Indeed, we want to sample configurations that have a high probability of leading to a low loss and a low probability of leading to a high loss. The configuration that satisfies this cannot be determined analytically. Therefore, this is typically done by sampling a large number of configurations and, for each, establishing the value of  $\ell(\theta)/g(\theta)$ . Additionally, Bergstra et al. (2011) show that sampling according to this criterion is similar to using the acquisition function *expected improvement*.

Thornton et al. (2013) identified that the tree-structured requirement of configuration spaces makes TPE a very suitable candidate for solving the CASH problem on an extensive set of Weka algorithms. Moreover, it is easy to parallelize, while most Bayesian optimization techniques are sequential in nature.

## 6.4 Metalearning for Hyperparameter Optimization

In this section, we cover metalearning techniques that extend the previously discussed optimization techniques with the ability to leverage knowledge from previous tasks. This often results in significantly faster anytime performance, since good configurations can be found faster.

### 6.4.1 Warm-starting: exploiting metaknowledge in initialization

Many optimization techniques start the search with randomly selected points. This can be improved by using metaknowledge to suggest the set of most suitable points to initialize the search.

#### Reusing best configuration

Bayesian methods suffer from a cold-start problem. That is, when relatively few test results are available, the surrogate model may not deliver good suggestions. This is why some researchers have proposed to reuse the meta-knowledge obtained on other datasets. This technique represents a kind of transfer from past datasets to the current one.

Reif et al. (2012) have done this in conjunction with the search method based on genetic algorithms. These methods usually achieve good results fast, but their performance may not reach the performance of simple grid search. The authors have shown that, by reusing the best configurations identified on *similar datasets*, the process can be speeded up substantially. The similarity of datasets was established with the help of metafeatures (see Chapter 4). A similar approach was used by Gomes et al. (2012).

Feurer et al. (2014, 2015b) have proposed a similar idea for Bayesian optimization. The best configurations identified on past problems were reused to initialize the search on the target datasets. This approach has led again to marked improvements, when compared, for instance, with random initialization. In particular, Feurer et al. (2015b) proposed two distance functions that estimate the distance between two datasets. One of these is based on a  $p$ -norm between meta-features of these datasets.

The other distance function is based on a *correlation of performance values* of different configurations that have been run on the dataset. The assumption is that configurations that have performed well on datasets similar to the current dataset (i.e., with a low distance between both datasets) will also perform well on the current dataset.<sup>1</sup> The authors have shown that initializing Bayesian optimization with such configurations yields superior performance.

Of course, these initialization procedures are limited to configurations that have already been examined in the past and the existing metadata captures this.

## Searching for a globally best configuration

The algorithm of Hutter et al. (2011) identifies several configurations, as the aim is to identify the best configuration for several dataset variants (called *instances*) at the same time. The algorithm includes an *intensify* step, which selects a subset which appears to be the best one for the given set of dataset variants.

A similar approach (Wistuba et al., 2015; Wistuba, 2018) uses an initialization method that generalizes the information found on different datasets. This way, the system can arrive at entirely new configurations that can be used for initialization.

The method uses the concept of *meta-loss*, representing effectively a *meta-level loss* across various datasets  $\mathcal{D}$ . The aim is to search for a configuration  $\theta_*$  that minimizes the difference between the global minimum and the best configuration for each dataset.

As this loss is not differentiable, the authors suggest to approximate the minimum function by a differentiable *softmin* function  $\sigma$ . So, the differentiable meta-loss can be expressed as

$$\mathcal{L}(\Theta_I, \mathcal{D}) = \frac{1}{|\mathcal{D}|} \sum_{D \in \mathcal{D}} \sum_{i=1}^I \sigma_{D,i} \hat{L}_D(\theta_i), \quad (6.9)$$

where  $\sigma_{D,i}$  represents the *softmin* function and  $\hat{L}_D(\theta_i)$  the estimate of the loss for the configuration  $\theta_i$  on dataset  $D$ .

The authors then derive an analytic form for the gradient which is then used in the gradient-descent approach. The procedure starts with the best set of configurations for each dataset  $\theta_1, \dots, \theta_I$  to initialize the process. In each cycle this solution is iteratively improved by selecting one of the configurations at a time. An enhanced method also uses dataset similarity, which captures the effect of configurations belonging to similar datasets.

Wistuba (2018) has conducted experiments which showed that this initialization method leads to a lower normalized loss than the method described earlier in Subsection 6.4.1 based on the work of Reif et al. (2012) and Feurer et al. (2014).

## Ranking configurations

We note that grid search, discussed in Section 6.2.2, does not really specify the order in which the alternatives should be tested. However, it is well known that some configurations may be better than others. Search methods that exploit metaknowledge from other datasets exploit this.

---

<sup>1</sup>Chapter 4 provides more details on various ways of establishing similarity between datasets.

All that is required is to define the configuration space beforehand, that is, all possible configurations of interest. This leads to a set of finite alternatives that can be used. Then, it is necessary to populate this space with test results to obtain metaknowledge *MetaD*. The ranking approach discussed in Chapter 2 can be used to construct a ranking of the pre-defined alternatives which can be used in the search.

One early work in this area was presented by Soares et al. (2004). The aim of the authors was to suggest a value of one parameter of SVM, the width of the Gaussian kernel ( $\sigma$ ), on the target dataset. The authors have shown that the methodology could be used to select a good configuration accompanied by a relatively low error. Although this approach could exploit metaknowledge acquired on prior datasets, it did not exploit the metaknowledge acquired on the current dataset. This shortcoming was corrected in the system AT\* (Abdulrahman et al., 2018), discussed in Chapter 5 (Section 5.8). The result of each new test carried out on the target dataset affects the selection of the subsequent tests.

Some experiments carried out with this approach are described in Chapter 7 (Section 7.4). In one of the experiments reported there, which was carried out by Cachada (2017), the given portfolio included various workflows with different hyperparameter configurations. This approach was able to identify a competitive workflow and compete well with Auto-WEKA.

## 6.4.2 Exploiting metaknowledge in Bayesian optimization

Recent work tries to reuse metaknowledge from past experiments in the search for the best algorithm configuration. This process can be seen as a kind of *transfer* from past datasets to the target dataset. The aim of this section is to describe some of the approaches that have been taken.

### Surrogate collaborative tuning (SCoT/MKL)

Bardenet et al. (2013) were the first to propose that a surrogate model be learned over observations from different datasets. Hence this method undertakes *multi-kernel learning*. They have used a *ranking model* instead of a regression model. This choice was motivated by the fact that, when a given algorithm is applied to different datasets, it tends to incur rather different losses. The relative model avoids this problem. A ranking of hyperparameter configurations per dataset was learned with *SVMRank* with an *RBF kernel*. As the ranking model does not provide the needed uncertainty estimations, the authors fit a Gaussian process to the output of the ranking model.

### Gaussian process with multi-kernel learning (MKL-GP)

Yogatama and Mann (2014) also proposed an algorithm for automatic hyperparameter tuning that can generalize across datasets. Their method is an instance of sequential model-based optimization (SMBO) that transfers information by constructing a common response surface for all datasets, similar to Bardenet et al. (2013). They did not use a *ranking model*, as Bardenet et al. (2013), but overcame the problem of losses of rather different magnitude by normalizing the data. After this, they could just use a regression model. The authors use a linear combination of two kernels:

- A squared exponential kernel with automatic relevance determination ( $k_{SE-ARD}$ ) for points belonging to the target dataset,

- A nearest-neighbor kernel ( $k_{NN}$ ) for modeling similarities between datasets.

The time complexity of reconstructing the response surface at every SMBO iteration is linear in the number of trials, allowing the method to scale up to many more datasets.

## Multi-task and multi-fidelity Bayesian optimization

Swersky et al. (2013) employed a *multi-task Gaussian process* (Bonilla et al., 2008) as a surrogate model. This model does not only model the performance of the algorithm across various configurations, but also the performance of the various configurations across other auxiliary tasks. The hope is that, if there are auxiliary tasks where similar configurations appear to have a similar performance as on task  $t$ , they can be used to speed up the process of learning. This is useful particularly when the experiments with the auxiliary task are faster to execute than the experiments on task  $t$ . This can happen, for example, if the auxiliary task is simpler, that is, if it includes fewer observations or attributes. This way it plays a similar role to *landmarkers* discussed in Chapter 4. The authors used the acquisition function *expected improvement per second* to emphasize the need for fast experimentation.

Klein et al. (2017) took this notion one step further and proposed a *multi-task Bayesian optimization* method, based on the notion of *multi-fidelity*. They argue that configurations perform similarly on subsets of the given dataset, and employ a GP to model the performance of the algorithm across various configurations and across various dataset sizes.

## Ensemble of individual surrogate models (SGPT)

As we have pointed out earlier, the aim of recent work on SMBO is to also exploit metadata concerning the effects of different hyperparameter configurations on different datasets in the search for the best configuration on the target dataset. However, Gaussian processes do not scale up well with growing metadata (Wistuba et al., 2018). This is due to the fact that the method involves an inversion of a kernel matrix, which represents a bottleneck.

To overcome this difficulty, Wistuba et al. (2016), Wistuba (2018), and Wistuba et al. (2018) have proposed to learn individual surrogate models on a set of different datasets. The target dataset is included in this set. Different surrogate models are then combined into a joint model using an ensembling technique.

The final surrogate is represented by a weighted sum of the individual surrogate models. In Wistuba et al. (2018) the authors call this approach the *scalable GP transfer surrogate framework* (SGPT).<sup>2</sup> Three different variants have been defined, both for *SGPT* and the corresponding *TST*. The variant SGTP-R, which uses pairwise hyperparameter performance descriptors, obtained the best experimental results.

## Transfer acquisition function (TAF)

The proposal discussed in the previous section suffers from some shortcomings. One major one is that the weights of different components do not change as tests proceed.

<sup>2</sup>In Wistuba (2018) (Chapter 7), this type of solution is referred to as the *two-stage transfer surrogate model*, *TST*.

This is counter-intuitive, because as tests proceed on the target dataset, the metadata on this dataset is more informative.

These observations led the authors (Wistuba et al., 2016; Wistuba, 2018; Wistuba et al., 2018) to propose another variant of the surrogate framework that exploits *transfer acquisition function* (TAF).<sup>3</sup>

The transfer acquisition function is defined as a weighted average of two components. The first one represents the expected improvement on the new target dataset. It tends to be rather unreliable in the early trials. The second component captures the predicted improvement on all other datasets used in previous experiments. This second component provided by the metadata is followed in the early trials. This favors hyperparameter configurations that have led to good performance on different datasets.

As time proceeds and as more information about the new target dataset has been gathered, the prediction obtained by the first component becomes more reliable, and consequently, the metadata starts to play a minor role.

Similarly, as with SGPT, the authors have defined three different variants. Here again, the variant TAF-R, which uses pairwise descriptors, has obtained better experimental results than the other two.

The authors have compared their system against others on two problems. One of them involved running 19 different Weka classifiers on 59 datasets with 21,871 hyperparameter configurations. TAF-R obtained competitive results when compared with the other approaches.

## Focusing on high-performance regions with QRF

Eggensperger et al. (2018) did not use Gaussian processes, but rather a regression algorithm as a surrogate model. The regression algorithm used was *quantile regression forest* (QRF) (Meinshausen, 2006) based on quantile regression (Koenker, 2005; Takeuchi et al., 2006).

Certain datasets were used as training data to generate the model. This involved results of various hyperparameter configurations obtained on the training datasets.

This method thus permits to focus on the high-performance regions of the parameter configuration space, in a way somewhat similar to *irace* (López-Ibáñez et al., 2011).

### 6.4.3 Adaptive dataset similarity

Chapter 5 (Section 5.8) describes various variants of active testing (AT\*), where the similarity is determined dynamically by combining the information gathered on both the new and past datasets. Some of the enhanced variants have led to significant improvements in performance on a combined algorithm selection and hyperparameter optimization (CASH) problem. It is foreseeable that this approach could compete with other existing approaches discussed in this section.

---

<sup>3</sup>In another publication (Wistuba, 2018) (Chapter 8), a similar system is discussed and referred to as *adaptive transfer hyperparameter learning* (AHT).

## 6.5 Concluding Remarks

### Relation to experiment design, exploration, and exploitation

The topics discussed in this chapter build on the work of many other areas, including, for instance, the area of *experiment design* (Robbins, 1952). Another area is the area of *reinforcement learning*, which has introduced the concepts of *exploration* and *exploitation*. The exploration phase can be equated to the process of conducting tests involving given algorithms and datasets, that is, the process of gathering metadata. As was shown in Chapters 2, 5 and 6, the metadata that was gathered is then used to construct a meta-model. So, the exploitation phase can be compared to the process of applying a given meta-level model to the target dataset so as to identify the best possible algorithm (or workflow).

The research in the area of multi-armed bandits (MABs) is in many ways related to the problems addressed in this chapter. The process of gathering test results can be compared to the process of gathering knowledge about different “arms” in multi-armed bandit (MAB) problems. The aim is to find a good compromise between exploration (i.e., examining different arms) and exploitation (using the best arm(s) for the target problem).

### Summary

In this chapter we have briefly reviewed various AutoML methods that successfully address the hyperparameter optimization (HPO) problem, as well as the combined algorithm selection and hyperparameter optimization (CASH) problem. We have started the exposition with simple uninformed search methods (grid search and random search) and then continued with more intelligent approaches, such as hill-climbing, best-first methods, successive halving, Hyperband and Bayesian optimization.

In general, the simple methods do not make use of metadata across tasks, but utilize knowledge that was acquired during the search process.

It has been shown that these techniques can be improved with metalearning. Section 6.4 provides an overview of the techniques that permit to incorporate metaknowledge obtained from previous tasks, often dramatically speeding up the search for the best models for new tasks.

### Discussion

One obvious question that arises is whether we have not replaced the original problem of algorithm selection for a specific dataset with a meta-level algorithm selection problem. This is due to the fact that various meta-level approaches exist. Several of those were discussed in this chapter.

However, we note that most of the hyperparameter search and optimization techniques enable users to automatically explore multiple algorithms and hyperparameter configurations. Even when these configurations are configured suboptimally, they enable data scientists to make better informed decisions regarding which configuration to use for their problem.

It is foreseeable that new comparative studies will provide a better insight in the future and enable us to identify fewer methods as generally useful and others that are useful in certain specific circumstances.

Chapter 7 continues with the topic of this chapter. The focus there is on how to construct solutions that include various algorithms, each with its own hyperparameters, usually referred to as *workflows* or *pipelines*.

## References

- Abdulrahman, S., Brazdil, P., van Rijn, J. N., and Vanschoren, J. (2018). Speeding up algorithm selection using average ranking and active testing by introducing runtime. *Machine Learning*, 107(1):79–108.
- Baker, B., Gupta, O., Raskar, R., and Naik, N. (2017). Accelerating neural architecture search using performance prediction. In *Proc. of ICLR 2017*.
- Bardenet, R., Brendel, M., Kégl, B., and Sebag, M. (2013). Collaborative hyperparameter tuning. In *Proceedings of the 30th International Conference on Machine Learning, ICML’13*, pages 199–207. JMLR.org.
- Bartz-Beielstein, T., Lasarczyk, C., and Preuss, M. (2005). Sequential parameter optimization. In *Proceedings of CEC-05*, page 773–780. IEEE Press.
- Baydin, A. G., Cornish, R., Rubio, D. M., Schmidt, M., and Wood, F. (2018). Online learning rate adaptation with hypergradient descent. In *Sixth International Conference on Learning Representations (ICLR), Vancouver, Canada, April 30 – May 3, 2018*.
- Bergstra, J. and Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305.
- Bergstra, J. S., Bardenet, R., Bengio, Y., and Kégl, B. (2011). Algorithms for hyperparameter optimization. In *Advances in Neural Information Processing Systems 24, NIPS’11*, pages 2546–2554.
- Bonilla, E. V., Chai, K. M., and Williams, C. (2008). Multi-task Gaussian process prediction. In *Advances in Neural Information Processing Systems 21, NIPS’08*, pages 153–160.
- Breiman, L. (2001). Random forests. *Machine learning*, 45(1):5–32.
- Brochu, E., Cora, V. M., and De Freitas, N. (2010). A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. arXiv preprint arXiv:1012.2599.
- Cachada, M. (2017). Ranking classification algorithms on past performance. Master’s thesis, Faculty of Economics, University of Porto.
- Cox, D. and John, S. (1997). SDO: A statistical method for global optimization. In *Multidisciplinary Design Optimization: State-of-the-Art*, page 315–329.
- de Miranda, P. B., Prudêncio, R. B., de Carvalho, A. C. P., and Soares, C. (2012). Combining a multi-objective optimization approach with meta-learning for SVM parameter selection. *Systems, Man, and Cybernetics (SMC)*, page 2909–2914.
- Eggenesperger, K., Lindauer, M., Hoos, H., Hutter, F., and Leyton-Brown, K. (2018). Efficient benchmarking of algorithm configuration procedures via model-based surrogates. *Special Issue on Metalearning and Algorithm Selection, Machine Learning*, 107(1).
- Falkner, S., Klein, A., and Hutter, F. (2018). BOHB: Robust and efficient hyperparameter optimization at scale. In Dy, J. and Krause, A., editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *ICML’18*, pages 1437–1446. JMLR.org.
- Feurer, M., Klein, A., Eggenesperger, K., Springenberg, J., Blum, M., and Hutter, F. (2015a). Efficient and robust automated machine learning. In Cortes, C., Lawrence, N., Lee, D., Sugiyama, M., and Garnett, R., editors, *Advances in Neural Information Processing Systems 28, NIPS’15*, pages 2962–2970. Curran Associates, Inc.

- Feurer, M., Springenberg, J., and Hutter, F. (2015b). Initializing Bayesian hyperparameter optimization via meta-learning. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, pages 1128–1135.
- Feurer, M., Springenberg, J. T., and Hutter, F. (2014). Using meta-learning to initialize Bayesian optimization of hyperparameters. In *ECAI Workshop on Metalearning and Algorithm Selection (MetaSel)*, pages 3–10.
- Gomes, T. A., Prudêncio, R. B., Soares, C., Rossi, A. L., and Carvalho, A. (2012). Meta-learning for evolutionary parameter optimization of classifiers. *Neurocomputing*, 75(1):3–13.
- Hansen, N. (2006). The CMA evolution strategy: a comparing review. In *Towards a New Evolutionary Computation*, pages 75–102. Springer.
- Hutter, F., Hoos, H., Leyton-Brown, K., and Stützle, T. (2009). ParamLLS: an automatic algorithm configuration framework. *JAIR*, 36:267–306.
- Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2011). Sequential model-based optimization for general algorithm configuration. *LION*, 5:507–523.
- Jamieson, K. and Talwalkar, A. (2016). Non-stochastic best arm identification and hyperparameter optimization. In *Artificial Intelligence and Statistics*, pages 240–248.
- John, G., Kohavi, R., and Pfleger, K. (1994). Irrelevant feature and the subset selection problem. In Cohen, W. and Hirsch, H., editors, *Machine Learning Proceedings 1994: Proceedings of the Eighth International Conference*, pages 121–129. Morgan Kaufmann.
- Jones, D., Schonlau, M., and Welch, W. (1998). Efficient global optimization of expensive black box functions. *Journal of Global Optimization*, 13:455–492.
- Klein, A., Falkner, S., Bartels, S., Hennig, P., and Hutter, F. (2017). Fast Bayesian optimization of machine learning hyperparameters on large datasets. In *Proc. of AISTATS 2017*.
- Koenker, R. (2005). *Quantile regression*. Cambridge University Press.
- Kuhn, M. (2008). Building predictive models in R using the caret package. *J. of Statistical Software*, 28(5).
- Kuhn, M. (2018). Package caret: Classification and regression training.
- Kushner, H. J. (1964). A new method of locating the maximum point of an arbitrary multipeak curve in the presence of noise. *Journal of Basic Engineering*, 86(1):97–106.
- Lavesson, N. and Davidsson, P. (2006). Quantifying the impact of learning algorithm parameter tuning. In *AAAI*, volume 6, pages 395–400.
- Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., and Talwalkar, A. (2017). Hyperband: Bandit-Based Configuration Evaluation for Hyperparameter Optimization. In *Proc. of ICLR 2017*.
- López-Ibáñez, M., Dubois-Lacoste, J., Cáceres, L. P., Birattari, M., and Stützle, T. (2016). The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58.
- López-Ibáñez, M., Dubois-Lacoste, J., Stützle, T., and Birattari, M. (2011). The irace package, iterated race for automatic algorithm configuration. Technical report, IRIDIA, Université libre de Bruxelles.
- Loshchilov, I. and Hutter, F. (2016). CMA-ES for hyperparameter optimization of deep neural networks. In *Proc. of ICLR 2016 Workshop*.
- Lourenço, H., Martin, O., and Stützle, T. (2003). Iterated local search. In Glover, F. and Kochenberger, G., editors, *Handbook of Metaheuristics*, pages 321–353. Kluwer Academic Publishers.
- MacKay, D. (1992). Information-based objective functions for active data selection. *Neural Computation*, 4(4):590–604.



- Maclaurin, D., Duvenaud, D., and Adams, R. P. (2015). Gradient-based hyperparameter optimization through reversible learning. In *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *ICML'15*, pages 2113–2122.
- Meinshausen, N. (2006). Quantile regression forests. *Journal of Machine Learning Research*, 7:983–999.
- Mockus, J., Tiesis, V., and Žilinskas, A. (1978). The application of Bayesian methods for seeking the extremum. *Towards Global Optimization*, 2:117–129.
- Moore, A. W. and Lee, M. S. (1994). Efficient algorithms for minimizing cross-validation error. In Cohen, W. and Hirsch, H., editors, *Machine Learning Proceedings 1994: Proceedings of the Eighth International Conference*, pages 190–198. Morgan Kaufmann.
- Rasmussen, C. and Williams, C. (2006). *Gaussian Processes for Machine Learning*. The MIT Press.
- Reif, M., Shafait, F., and Dengel, A. (2012). Meta-learning for evolutionary parameter optimization of classifiers. *Machine learning*, 87(3):357–380.
- Robbins, H. (1952). Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society*, 55:527–535.
- Russell, S. J. and Norvig, P. (2016). *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3rd edition.
- Snoek, J., Larochelle, H., and Adams, R. P. (2012). Practical Bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems 25*, NIPS'12, page 2951–2959.
- Soares, C., Brazdil, P., and Kuba, P. (2004). A meta-learning method to select the kernel width in support vector regression. *Machine Learning*, 54:195–209.
- Srinivas, N., Krause, A., Seeger, M., and Kakade, S. M. (2010). Gaussian process optimization in the bandit setting: No regret and experimental design. In *Proceedings of the 27th International Conference on Machine Learning*, ICML'10, page 1015–1022. Omnipress.
- Swersky, K., Snoek, J., and Adams, R. P. (2013). Multi-task Bayesian optimization. In Burges, C. J. C., Bottou, L., Welling, M., Ghahramani, Z., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 26*, NIPS'13, pages 2004–2012. Curran Associates, Inc.
- Takeuchi, I., Le, Q., Sears, T., and Smola, A. (2006). Nonparametric quantile estimation. *Journal of Machine Learning Research*, 7:1231–1264.
- Thornton, C., Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2013). Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 847–855. ACM.
- van Rijn, J. N. and Hutter, F. (2018). Hyperparameter importance across datasets. In *KDD '18: The 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM.
- Wistuba, M. (2018). *Automated Machine Learning: Bayesian Optimization, Meta-Learning & Applications*. PhD thesis, University of Hildesheim, Germany.
- Wistuba, M., N. Schilling, L., and Schmidt-Thieme (2018). Scalable Gaussian process-based transfer surrogates for hyperparameter optimization. *Machine Learning*, 107(1):43–78.
- Wistuba, M., Schilling, N., and Schmidt-Thieme, L. (2015). Learning hyperparameter optimization initializations. In *2015 IEEE International Conference on Data Science and Advanced Analytics, DSAA 2015*, pages 1–10.
- Wistuba, M., Schilling, N., and Schmidt-Thieme, L. (2016). Two-stage transfer surrogate model for automatic hyperparameter optimization. In *Machine Learning and Knowl-*

*edge Discovery in Databases - European Conference, ECML-PKDD 2016, Proceedings*, pages 199–214.

Yogatama, D. and Mann, G. (2014). Efficient transfer learning method for automatic hyperparameter tuning. In *Proceedings of the International Conference on Artificial Intelligence and Statistics*.

## Automating Workflow/Pipeline Design

**Summary.** This chapter discusses the design of *workflows* (or *pipelines*), which represent solutions that involve more than one algorithm. This is motivated by the fact that many tasks require such solutions. This problem is non-trivial, as the number of possible workflows (and their configurations) can be rather large. This chapter discusses various methods that can be used to restrict the design options and thus reduce the size of the configuration space. These include, for instance, ontologies and context-free grammars. Each of these formalisms has its merits and shortcomings. Many platforms have resorted to planning systems that use operators. These can be designed to be in accordance with the given ontologies or grammars. As the search space may be rather large, it is important to leverage prior experience. This topic is addressed in one of the sections, which discusses *rankings of plans* that have proved to be useful in the past. The workflows/pipelines that have proved successful in the past can be retrieved and used as plans in future tasks. Thus, it is possible to exploit both planning and metalearning.

### 7.1 Introduction

This chapter extends the discussion in Chapters 5 and 6, which discussed various approaches to selection of algorithms and their configurations. In many practical situations, the aim is to select not just one algorithm and its configuration, but rather a sequence of algorithms (or operators). An example of such a sequence could be to remove the outliers from the data, impute all missing values, and build a decision tree classifier on the dataset. Typically, such a sequence starts with one or several *data transformations* and ends with a machine learning algorithm (e.g., a classifier). Sometimes, post-processing operators are also defined. A typical example is a KDD task that may be resolved only by applying such a sequence.

The term *knowledge discovery from data* (KDD) was introduced at the first KDD workshop in 1989 (Piatetsky-Shapiro, 1991). This term emphasizes that “knowledge” is the

---

<sup>1</sup>The topic of this chapter was addressed already in the first edition, namely in Chapter 4 (Extending Metalearning to Data Mining and KDD (pp. 61–72)). This chapter was prepared by Christophe Giraud-Carrier, which we gratefully acknowledge. A part of this material was reused, revised, and reorganized. Besides, various new sections have been added.

end product of data-driven discovery. The reader may consult Figure 7.1 shown further on in this section. The term *data mining (DM)* is sometimes used as a synonym for KDD. Others consider it to be a part of the KDD process, focusing on the process of analyzing hidden patterns in the data according to different perspectives. Both terms, *KDD* and *DM*, have been popularized in AI and machine learning.

Designing sequences of operations has certain specificities, and these are addressed in this chapter. The sequences of algorithms are often referred to as *workflows* or *pipelines* of operations. In earlier work, some researchers called these sequences *DM processes* (Bernstein and Provost, 2001), others also *streams* (Engels et al., 1997; Wirth et al., 1997) or *plans*.

Many methods discussed in Chapters 5 and 6 are obviously also applicable to the more general task of designing workflows. However, we will not include the description of these methods here, simply to avoid duplication.

Chapter 14, which discusses the topic of automating data science (DS), addresses some specific problems that arise in the course of resolving typical DS tasks. So, the methods for the design of workflows (pipelines) may be reused in that setting too.

## Organization of this chapter

When creating entire machine learning/KDD workflows (pipelines), the number of configuration options grows dramatically. It is therefore important to exclude the useless branches from the search space or to avoid them when conducting the search. Section 7.2 is dedicated to this topic. In this section we discuss how ontologies and grammars (e.g., CFGs) can be exploited for this aim. It is possible to use carefully designed abstract and concrete operators that follow the principles of ontologies or grammars. This topic is discussed in Section 7.3. In the same section we also discuss effective methods based on hierarchical planning that can search effectively through the given search space.

As the search space may be rather large, it is important to leverage prior experience. This topic is addressed in Section 7.4, which discusses rankings of plans that have proved to be useful in the past. Many of the techniques discussed in Chapter 2, where the focus was on rankings of algorithms, are also applicable to rankings of workflows.

Let us start by examining the KDD process, to provide a more precise idea about the kind of workflows we are concerned about in this chapter.

## The KDD process

If we examine the KDD process in some detail (see Figure 7.1), it is clear that not all stages in it lend themselves naturally to automatic advice. Typically, both the early stages (e.g., problem formulation, domain understanding) and the late stages (e.g., interpretation, evaluation) require significant human input as they depend heavily on business knowledge.

The more algorithmic stages (i.e., preprocessing and model building), on the other hand, are ideal candidates for automation through adequate use of metaknowledge. Some decision systems focus exclusively on one of these stages, while others take a holistic approach, considering all stages of the KDD process collectively (i.e., as sequences of steps or workflows).

In this chapter we describe some basic concepts that have been followed in different data mining prototypes/systems that exploit metaknowledge to also provide decision support to users.

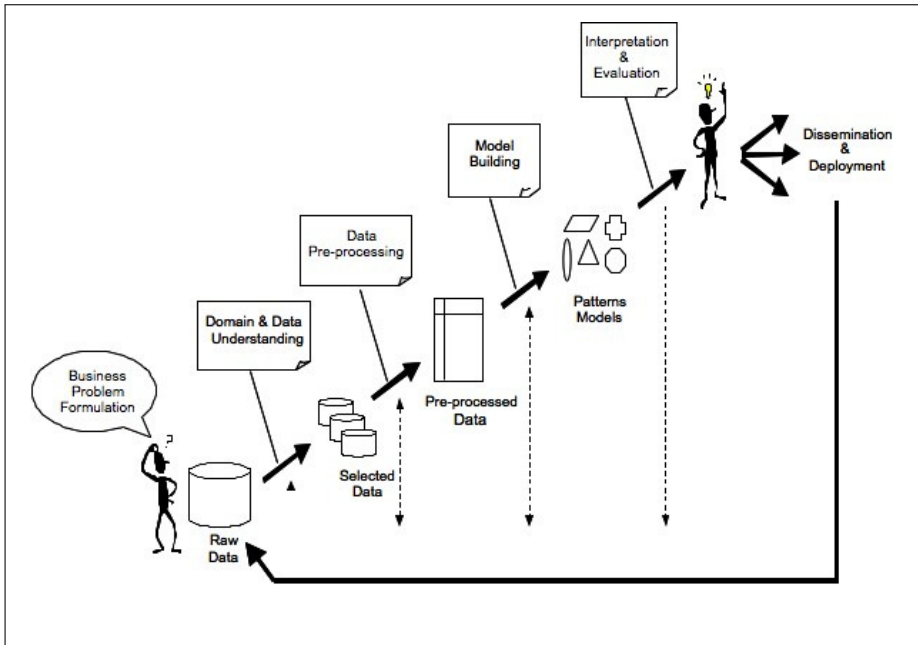


Fig. 7.1: The KDD process

## 7.2 Constraining the Search in Automatic Workflow Design

The designers of data mining systems need to address the following phases:

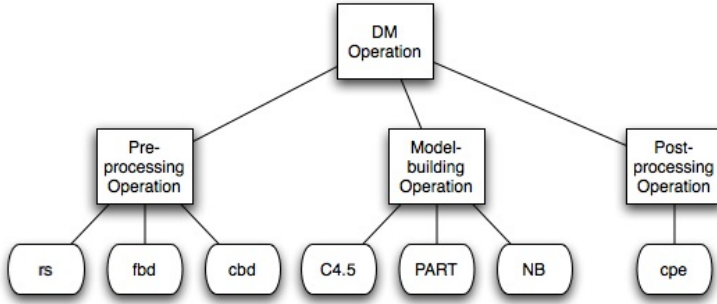
- Defining the space of alternative workflows (configuration space)
- Searching through the space of alternative workflows and selecting the most appropriate one(s) for the target dataset

Each of these is discussed in more detail in the following subsections.

### 7.2.1 Defining the space of alternatives (declarative bias)

The phase when the space of alternative workflows is defined normally precedes all the other stages. It takes into account the users' goals and determines which kind of meta-data should be collected so that we could come up with a useful system. The definition of a space of alternatives is related to what some researchers in ML would call *declarative bias*. It specifies the representation of the space of hypotheses and affects the size of the search space (Mitchell, 1982; Gordon and desJardins, 1995). In the context of workflows, the definition of a space of alternatives involves two issues:

- Definition of the basic constituents (operators) of workflows
- Definition of the ways these constituents (operators) can be combined to generate workflows



rs = random sampling (10%), fbd = fixed-bin discretization (10 bins), cbd = class-based discretization, cpe = CPE-thresholding post-processor

Fig. 7.2: Sample IDA ontology

Some researchers have introduced a notion of *algorithm portfolios* to represent different sets/lists of algorithms/workflows (Gomes and Selmany, 2001; Leyton-Brown et al., 2003). These could be regarded as the basic constituents mentioned above. The disadvantage of this approach is that we cannot group similar algorithms into closely related groups. This problem can be overcome by adopting ontologies, which are discussed next.

### Role of ontologies

Ontologies (Chandrasekaran and Jopheson, 1999) allow us to describe a set of constituents of workflows, often referred to as operators. They have been employed in various machine learning and data mining systems, including, for instance:

- DM ontology used in IDA (Bernstein and Provost, 2001)
- OWL-DL-based ontology used in the RDF system (Patel-Schneider et al., 2004)
- DMWF, Data Mining Workflow Ontology used in the eIDA system (Kietz et al., 2009)
- KDDONTO (Diamantini et al., 2012)
- DMOP (Hilario et al., 2009)
- KD ontology (Žáková et al., 2011)
- Exposé ontology (Vanschoren et al., 2012)
- Auto-Weka parameter space (Thornton et al., 2013)

More details about some of these ontologies can be found in an overview paper of Serban et al. (2013). Let us examine two ontologies in more detail.

**Ontology used in the IDA** A simplified ontology used in IDA system of Bernstein and Provost (2001) is shown in Figure 7.2. The bottom layer shows the *concrete operators*, such as *rs*, *fbd*, etc. The level above shows what some call *abstract operators*, which include *preprocessing*, *model-building* and *post-processing operation*. An example of a workflow resulting from this ontology is “*fbd; nb; cpe*”, meaning that first the data is discretized, afterwards a naive Bayes classifier is applied to it, and finally CPE-thresholding is applied to the predictions.

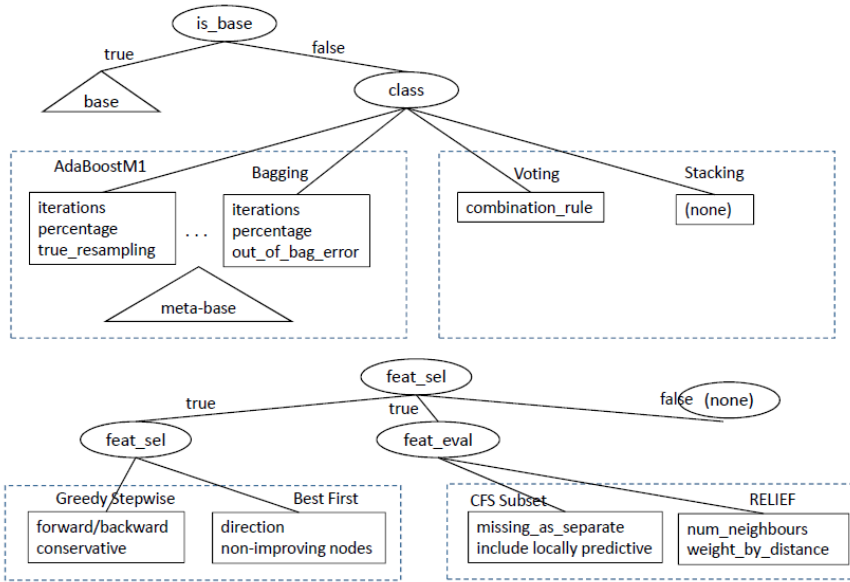


Fig. 7.3: Partial Auto-WEKA parameter space (redrawn following Thornton et al. (2013))

**Ontology used in Auto-WEKA** Figure 7.3 shows a part of the ontology used in Auto-WEKA (Thornton et al., 2013). The top part is concerned with the selection of classifications methods, which can be either *base classifiers* (left) or *ensemble methods* (right). The triangle with the word *base* inside represents a numeric parameter (index) determining which of the 27 base classifiers should be selected. Some components show the associated hyperparameters that need to be set. For example, the component “AdaBoostM1” requires the setting of three hyperparameters, namely “iterations”, “percentage” and “true resampling”. The bottom part of the figure is concerned with feature selection/evaluation methods.

### What ontologies usually do not express

We note that ontologies typically do not say anything about the order in which the operations should be applied. We may be inclined to assume that the elements are searched through in left-to-right order, but this may not be so. Also, ontologies do not specify whether we should apply *all* or *just some* operators belonging to some branch.<sup>1</sup>

Note that ontologies can be used to generate workflows. However many components include multiple hyperparameters that need to be set. So search and optimization methods are required in this process (see Chapter 6).

Although the ontology helps us to construct all valid workflows, such as “*fbd; nb; cpe*”, it permits to obtain various other workflows such as “*c4.5; nb*”, that may not be consid-

<sup>1</sup>Although the language used to describe ontologies could, in principle, be refined to capture such aspects, the ontologies encountered do not do this.

ered valid. So we need other ways to determine the order in which the search space should be searched through. Apart from declarative bias, we also need to determine what some researchers call *procedural bias* (Mitchell, 1997). This issue is discussed in the next subsection.

## 7.2.2 Different ways of imposing procedural bias

Different mechanisms can be used to control the way a given space can be searched through. It can be done using:

- Heuristic rankers
- Context-free grammars
- Operators with preconditions and post-conditions (effects)

### Using a heuristic ranker

The first possibility was explored in IDA (Bernstein and Provost, 2001). It uses a heuristic ranker which ranks the valid operations (processes) using a heuristic function which incorporates the user's preferences regarding speed, accuracy, and model comprehensibility. These characteristics are specified in the ontology for all operators. One problem with this approach is that these characteristics are acquired through experience on individual cases.

## 7.2.3 Context-free grammars (CFGs)

The formalism of context-free grammars (CFGs) enables to not only define the space to alternative models but also impose some restrictions regarding the search process, as we will see.

A CFG is a 4-tuple  $(S, V, \Sigma, R)$ , where  $S$  is the starting symbol,  $V$  a set of non-terminal symbols,  $\Sigma$  a set of terminal symbols, and  $R$  a set of production rules specifying replacements of symbols (Hopcroft and Ullman, 1979).

The difference between the two types of symbols is that non-terminal symbols can be substituted by other symbols, while terminal symbols cannot. Valid substitutions are expressed in the form of production rules. The left-hand side of these rules includes a non-terminal symbol, while the right-hand side can include either a terminal symbol or a combination of non-terminals and terminals.

### Example

Let us now see how we can use CFGs to represent the simple ontology shown in Figure 7.2. Let  $S$  represent the starting symbol. All non-terminals here are represented by strings starting with a capital letter. So, for instance, *Pre.Proc* represents a single non-terminal corresponding to *Pre-Processing Operation* in Figure 7.2. So the set  $V$  would include this symbol, among others.

All terminal symbols are represented by strings in lower case. So, for instance, "c4.5" and "part" are terminals, representing particular classifiers. The set  $\Sigma$  would include these terminals among others.

Sequences of symbols are represented by the operator ";". The symbol "|" represents alternatives (logical OR). The null operation is represented by *null*. One possible set  $R$  of grammar rules is shown below:



1.  $S \leftarrow DM.Oper$
2.  $DM.Oper \leftarrow Pre.Proc ; Model.Build$
3.  $Pre.Proc \leftarrow Sampling ; Discret$
4.  $Sampling \leftarrow rs \mid null$
5.  $Discret \leftarrow fbd \mid cbd \mid null$
6.  $Model.Build \leftarrow Cat.Classif \mid Prob.Classif.Post$
7.  $Cat.Classif \leftarrow c4.5 \mid part$
8.  $Prob.Classif.Post \leftarrow nb ; cpe$

Let us see how some of the rules above can be interpreted. Line 1 states that the workflow should start with a “DM.Oper”, which is defined on line 2. However, this is a non-terminal and hence needs to be substituted by other symbols. We note that this non-terminal corresponds to the abstract operator shown in the ontology in Fig. 7.2. Line 2 states that “DM.Oper” can be substituted by the sequence of “Pre.Proc” and “Model.Build”, defined on line 3 and line 6, respectively.

The operation “Pre.Proc” (line 3), in turn, consists of a sequence of “Sampling” and “Discret”, which are defined further on. Line 4 defines “Sampling” and introduces terminal symbols: either “rs” (representing random sampling) or “null” (null operation). These can be related to the concrete operators shown as *leaf nodes* in the ontology in Figure 7.2.

The grammar above enables to generate the workflows shown in Table 7.1, which correspond to a sample of DM processes generated by IDA. Note that the occurrences of “null” are typically not displayed.

Table 7.1: Sample of IDA-generated DM processes (workflows)

N°	Workflow
1	c4.5
2	part
3	nb; cpe
4	rs; c4.5
5	rs; part
6	rs; nb; cpe
7	fdb; c4.5
8	fdb; part
9	fdb; nb; cpe
10	cbd; c4.5
11	cbd; part
12	cbd; nb; cpe
13	rs; fbd; c4.5
14	rs; fbd; part
15	rs; fbd; nb; cpe
16	rs; cbd; c4.5
17	rs; cbd; part
18	rs; cbd; nb; cpe

We note that the rules allow us to use the operation “*cpe*” only in conjunction with the probabilistic classifier “*nb*” that outputs probabilities. Also, it is not possible to obtain some invalid operator sequences, such as “*c4.5; nb*”.<sup>2</sup> In other words, CFGs can capture certain aspects of procedural bias.

An alternative way of imposing procedural bias could be achieved by adopting the formalism of *context-sensitive grammars* (CSGs) (Martin, 2010; Linz, 2011).

## Inducing CFGs from examples of workflows

Some users may find it easier to provide a list of valid/invalid workflows, rather than some formal procedure or grammar that could be executed. Various papers address the issue of grammar induction (see, e.g., Duda et al. (2001)). The main idea is to spot frequent patterns, such as the sequence “*nb; cpe*” in the examples of workflows in Table 7.1. The frequent pattern identified can then be substituted by a new symbol, which could be renamed by the user to, say, *Prob.Classif.Post*. Besides, we also need to add a new rule showing how to interpret the new symbol. In our case we would get

$$Prob.Classif.Post \leftarrow nb; cpe.$$

In Chapter 15, we discuss various operations that enable to construct new concepts. These transformations could be used to introduce new higher-level concepts, corresponding to new non-terminals in CFGs and abstract operators discussed further on (see Subsection 7.3.4).

One problem with this approach is that, normally, many alternative grammars exist that are consistent with a given set of workflows. Another problem is that the new symbols introduced may not relate to the concepts of the user. Besides, we need a system that is capable of not only inducing a set of grammar rules but also revising them when new examples of workflows have been added.

## Limitations of CFGs

CFGs have various limitations. First, it can happen that a particular operation  $OP_i$  can be transformed into a sequence  $OP_j$  in some contexts or into a sequence  $OP_k$  in others. This problem can be resolved by specifying the conditions under which each particular transformation should occur. This difficulty can be circumscribed in a formalism which exploits operators. This formalism is discussed in the following two sections.

## 7.3 Strategies Used in Workflow Design

Workflows can be constructed in different ways. It can be done manually either from scratch, or by modifying an existing workflow, or automatically with recourse to a planner. The first two options are discussed in the following subsections. The use of planning approaches is discussed in a separate section further on.

---

<sup>2</sup>To achieve that, the *model-building operations* were separated into *categorical classifiers*, which include “*c4.5, part*”, and *probabilistic classifiers*, which include just “*nb*”. The ontology could, of course, be redesigned to capture this aspect.

### 7.3.1 Operators

Operators were already used in the 1970s in the area of *automated planning (AI planning)*, which is a branch of AI (Russell and Norvig, 2016; Fikes and Nilsson, 1971). Its aim is to design strategies or action sequences that could be executed by some system. In classical planning, the system may be an intelligent agent or a robot. Here the aim of the planning system is to design workflows.

The selection of operators is guided by pre- and post-conditions, sometimes also called *effects*. Preconditions are conditions that must be met for the operation to be applicable. Post-conditions (effects) are conditions that are true after the operation is applied, i.e., how the operation changes the state.

### 7.3.2 Manual selection of operators

Manual construction is usually done nowadays with the aid of a palette, which includes all possible operators. The user then constructs a workflow by connecting together operations selected from the palette. The system can check preconditions, can make suggestions as to what operations might be required, and essentially maintains the integrity of the workflow. This approach was used, for instance, in CITRUS (Engels et al., 1997; Wirth et al., 1997).

### 7.3.3 Manual modification of existing workflows

If this alternative is taken, the user needs to provide a high-level description of the task at hand. The system acts as a kind of case-based reasoning system which searches for and identifies closest matches in past experiments. These may be real tasks previously performed or basic templates designed by experts. The closest match is presented to the user, who in turn can adapt it to the new target task.

This approach was used for instance in CITRUS (Engels et al., 1997; Wirth et al., 1997), discussed earlier, and in MiningMart. The latter was a large European research project which focused on algorithm selection for preprocessing rather than for model building (Euler et al., 2003; Euler and Scholz, 2004; Morik and Scholz, 2004; Euler, 2005).

Preprocessing generally consists of nontrivial sequences of operations or data transformations, and is widely recognized as the most time-consuming part of the KDD process, accounting for up to 80% of the overall effort. Hence, automatic guidance in this area can indeed greatly benefit users.

The goal of MiningMart is to enable the reuse of successful preprocessing phases across applications through case-based reasoning. A model for metadata, called M4, is used to capture information about both data and operator chains through a user-friendly computer interface. The complete description of a preprocessing phase in M4 makes up a *case*, which can be added to MiningMart's case base (MiningMartCB, 2003; Morik and Scholz, 2004):

*“To support the case designer a list of available operators and their overall categories, e.g., feature construction, clustering or sampling is part of the conceptual case model of M4. The idea is to offer a fixed set of powerful preprocessing operators, in order to offer a comfortable way of setting up cases on the one hand, and ensuring re-usability of cases on the other.”*

Given a new mining task, the user may search through MiningMart’s case base for the case that seems most appropriate for the task at hand. M4 supports a kind of business level, at which connections between cases and business goals may be established. Its more informal descriptions are intended to “help decision makers to find a case tailored for their specific domain and problem.”

A less ambitious attempt at assisting users with preprocessing has been proposed, with a specific focus on data transformation and feature construction (Phillips and Buchanan, 2001). The system works at the level of the set of attributes and their domains, and an ontology is used to transfer across tasks and suggest new attributes (in new tasks based on what was done in prior ones).

### 7.3.4 Using planning in workflow design

The classical definition of a planning problem is often formulated as follows. Given a description of the possible initial states of the world, a description of the desired goals, and a description of a set of possible actions (represented by operators), the aim of the planning system is to synthesize a plan that can generate a state which contains the desired goals.

We note that this definition does not quite apply to our problem and needs to be suitably adapted. For instance, when searching for a suitable workflow for a classification task, the aim is to achieve the highest possible accuracy, but we do not know a priori what value should be set as a goal.

The most commonly used languages for classical planning are STRIPS (Fikes and Nilsson, 1971) and PDDL (McDermott et al., 1998). These representations suffer from the curse of dimensionality, and some researchers have resorted to hierarchical planning, discussed further on. However, before that, we need to clarify what are abstract and concrete operators.

#### Abstract and base-level operators

Hierarchical planning enables to decompose complex tasks into less complex ones. Complex tasks are realized by *abstract operators* and primitive tasks by base-level operators, which in our case are base-level algorithms. These notions existed already in the early robot planning system STRIPS. Abstract operators were referred to as *intermediate-level actions* (ILAs) and concrete operators as *low-level actions* (LLAs) (Russell and Norvig, 2016).

A typical process involves mapping higher-level tasks to higher-level (abstract) operators, representing groups of operators at a lower level. The selection of operators is guided by pre- and post-conditions, sometimes also called *effects*.

Preconditions are conditions that must be met for the operation to be applicable (e.g., a discretization operation expects continuous inputs, a naive Bayes classifier works only with nominal inputs<sup>3</sup>). Post-conditions (effects) are conditions that are true after the operation is applied, i.e., how the operation changes the state of the data (e.g., all inputs are nominal following a discretization operation; a decision tree is produced by a decision tree learning algorithm). One obvious method is to define operators manually. However, this is quite a laborious task, particularly if the required set of operators is

---

<sup>3</sup>In some implementations, a discretization step is integrated, essentially allowing the naive Bayes classifier to act on any type of input.

large and if, in addition, many interactions exist. As the operators need to conform to given ontologies, some approaches exploit the ontological descriptions of operators to obtain the descriptions used in planning.

As the design of a set of operators could be quite a lengthy process, it would be useful to have a kind of support system that would enable to generate a set of operators capable of generating all valid workflows but none of the invalid ones.

## How planning works

Many systems use a planner to design workflows. These include, for instance, CITRUS (Engels et al., 1997; Wirth et al., 1997), IDA (Bernstein and Provost, 2001), the eIDA system (Kietz et al., 2012), Auto-WEKA (Thornton et al., 2013; Kotthoff et al., 2016), and Auto-sklearn (Feurer et al., 2015), among others. A typical process involves the following steps.

The plan generator takes as input a dataset, a user-defined objective (e.g., build a classifier), and user-supplied information about the data, information that may not be obtained automatically. Starting with an empty process, it systematically searches for an operation whose preconditions are met and whose indicators are congruent with the user-defined preferences. Once an operation has been found, it is added to the current process and its post-conditions become the system's new conditions from which the search resumes. The search ends once a goal state has been reached or when it is clear that no satisfactory goal state may be reached.

In IDA the search of the planner is exhaustive: all valid workflows are returned. This was feasible at the time, as the problems studied did not involve large search spaces. However, even relatively simple problems may involve rather larger search spaces with hundreds or thousands of nodes, and so this approach is not really feasible. Hence we need to employ strategies that would help us to overcome these problems. These are discussed in the next subsection.

## Exploiting hierarchical planning

Hierarchical planning goes back to the 1970s (Sacerdoti, 1974) and can be seen as a sub-area of planning (Ghallab et al., 2004). As it exploits *hierarchical task networks* (HTN) (Kietz et al., 2009, 2012), it is sometimes referred to as HTN planning (Georgievski and Aiello, 2015). In some systems a reasoning engine is integrated in the planner directly by querying the ontology (Kietz et al., 2009; Žáková et al., 2011). Various other approaches are discussed by Serban et al. (2013).

The ML-Plan system (Mohr et al., 2018; Wever et al., 2018) is an AutoML system based on hierarchical planning. The authors have shown that it is highly competitive to some state-of-the-art systems, including Auto-WEKA, Auto-sklearn, and TPOT. Another planning system was discussed by Gil et al. (2018).

## Tree-based pipeline optimization tool (TPOT)

This is a technique that uses evolutionary search to address the CASH problem (Olson et al., 2016). It works by evolving several small machine learning workflows into larger workflows. It starts by sampling a large amount of small pipelines, typically consisting of a data preprocessing operator and a single classifier. The *cross-over function* is defined by

merging the processed elements from two pipelines. Figure 7.4 illustrates this by means of the *Combine Features* operator. Only one of the classifiers prevails. When this process is repeated for several generations, pipelines can grow larger and more complex.

There are also various *mutation operators* which can adapt the pipeline by adding more operators or dropping some.

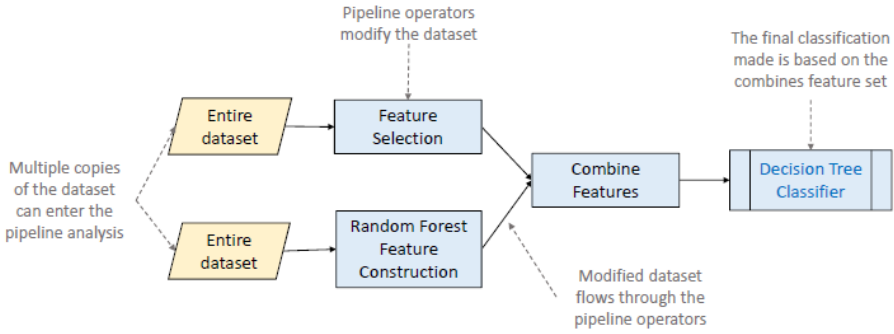


Fig. 7.4: An example of a tree-based pipeline, generated by TPOT (based on Olson et al. (2016))

Olson et al. (2016) focus on finding an appropriate pipeline with good preprocessing operations. The set of classifiers used in the experiments was restricted to decision trees and random forests.

### General automatic machine learning assistant (GAMA)

The *general automatic machine learning assistant* (GAMA) also leverages a multi-objective genetic programming technique, similar to TPOT (Gijsbers and Vanschoren, 2019). However, while TPOT uses a synchronous strategy, evaluating all candidate pipelines in each generation before selecting the best one, GAMA uses an asynchronous approach, which is often faster since it is not slowed down by slow pipelines (stragglers) in a population. It is also much more modular: it allows the user to switch to other optimization techniques, e.g. asynchronous successive halving algorithm (ASHA), which is often faster on larger datasets, and allows post-processing, such as building an ensemble out of the pipelines evaluated during the search, similar to Auto-sklearn. It also includes logging and visualization of the search process to help researchers understand what it is doing.

### Methods for reducing the search space

The main strategy for reducing the search space of planners relies on eliminating some parts of the search space, provided this does not affect the final outcome. This can be achieved in various ways. Previous strategies involved, for instance, prioritizing rules and also adding constraints (conditions) to rules to restrict their applicability (Brazdil, 1984; Clark and Niblett, 1989). These techniques can be reused in the design of operators.

Kietz et al. (2012) have pointed out that most of the preprocessing methods are recursive, handling one attribute at a time. It does not make sense to consider different orderings in which the attributes could be processed. If the dataset includes just two attributes, we could capture this by

$$O_p(At_1); O_p(At_2) \mid O_p(At_2); O_p(At_1), \quad (7.1)$$

where  $O_p(At_j)$  represents the application of a particular preprocessing operator to attribute  $At_j$ . The formulation  $O_p(At_1); O_p(At_2)$  restricts the search space and hence could be adopted, but does not quite express that all orderings lead to the same effect. The formulation in Eq. 7.1 is better, as it captures that the operations are commutative. The choice of the ordering that should be used is left to the interpreter.

If we were to reformulate this using operators with preconditions and effects, we would have to make sure that it is equivalent to what Eq. 7.1 expresses. In a more general case, which includes a set of attributes, simple enumeration of all alternatives, as is done in Eq. 7.1, is impractical, as for  $N$  items there are  $N!$  orderings. To resolve this case we need higher-level constructs, which allow us to select one possible ordering from a set.

The system discussed by Kietz et al. (2012) goes part of the way in this direction. The operator *CleanMissingValues* is applied to all attributes, but the formalism does not capture the fact that one of the orderings was selected from a set.

Furthermore, the HTN planning method of Kietz et al. (2012) prevents senseless operator combinations. For example, first normalizing the data and then discretizing it does not make sense, as it produces a similar effect as applying just discretization. Converting the scalar data to nominal and then converting it back is useless. Eliminating such possibilities from the search space speeds up the process of planning.

## Prioritizing the search

As planners can, in principle, generate a very large number of correct candidate workflows, other techniques need to be applied to control the search. Different possibilities can be exploited:

- Ask the planner to return a different subset of workflows (limited number) each time it is invoked;
- Exploit meta-learning to identify and rank the potentially best workflows. Various chapters (e.g., 2 and 5) in this book provide more details about this process;
- Exploit sequential model-based optimization (SMBO) to suggest the potentially best workflow(s) to test. Chapter 6 discusses this issue in more detail.

The reader can consult the respective chapters to obtain more information about each of these techniques.

## Exploiting metaknowledge in planning

Nguyen et al. (2014) exploit a meta-mining model which is used to guide the planner during the workflow planning. The meta-mining models are learned using a similarity learning approach. These models extract workflow descriptors by mining the workflows for generalized relational patterns. This mechanism allows us to focus the search on components recommended by the meta-learner.

## Methods for revision workflows (pipelines)

AlphaD3M (Drori et al., 2018) is an AutoML system that uses meta reinforcement learning on sequence models generated by self play. The current state is represented by the current pipeline, and possible actions include addition, deletion, or replacement of pipeline components. A Monte Carlo tree search (MCTS) (Silver et al., 2017; Anthony et al., 2017) generates pipelines, which are evaluated to train a recurrent neural network (LSTM) that can predict pipeline performance, in turn producing the action probabilities for the MCTS in the next round. The state description also includes metafeatures of the current task, allowing the neural network to learn across tasks. AlphaD3M was compared with state-of-the-art AutoML systems: Auto-sklearn, Autostacker, and TPOT on OpenML datasets. The authors claim that this system achieves competitive performance while being an order of magnitude faster.

Alternatively, Naive AutoML is a baseline for pipeline search, where each component is searched based on an independence assumption (Mohr and Wever, 2021). Each component is optimized independent from other components, shrinking the search space.

## 7.4 Exploiting Rankings of Successful Plans (Workflows)

This chapter extends the material presented in Chapter 2 which discussed rankings of algorithms. Here, we consider rankings of workflows rather than just individual algorithms. The basic idea consists of storing all potentially useful workflows that were identified in the past for future use. Normally the workflows are ranked according to a given estimate of how useful they were in the past, for instance, *average rank* across different datasets. The assumption here is that the ranking can help us to identify the potentially best workflow on the new dataset.

Serban et al. (2013) refer to these kinds of approaches as *case-based reasoning* (CBR) approaches. The idea of storing ranked cases is not new. The idea was explored already in the 1990s in the Statlog and Metal projects and in the subsequent design of Data Mining Advisor (DMA) (Brazdil and Henery, 1994; Giraud-Carrier, 2005) and AST (Lindner and Studer, 1999). These systems stored algorithms rather than workflows. Other systems, including, for instance, CITRUS (Engels et al., 1997; Wirth et al., 1997) or MiningMart (Euler et al., 2003; Euler and Scholz, 2004; Morik and Scholz, 2004; Euler, 2005), stored workflows.

The idea of storing more complex constructs which are potentially useful in future tasks has appeared in different forms in the past. The early planning system STRIPS introduced the possibility of storing the results of planning in the form of generalized macro-operators (Russell and Norvig, 2016; Fikes and Nilsson, 1971).

Tabling, also referred to as *tabulation* or *memoing*, was proposed already in late 1960s (Michie, 1968). It consists of storing intermediate results for subgoals so that they can be reused later when the same subgoal appears again. In the area of logic programming, tabling is a form of automatic caching or memorization of results of previous computations. Storing them can avoid unnecessary recomputation.

### Effectiveness of this approach

It was shown that a ranked case base of workflows can achieve better results than AutoWEKA. Cachada et al. (2017) have exploited the AR\* metalearning method in the com-



Table 7.2: Comparing AR\* with workflows and Auto-WEKA

Budget	Win	Loss	Tie
5	35	1	1
15	31	5	1
30	29	7	1
60	7	11	1

parisons. The advantage of the ranked case base approach was particularly significant when the given time budgets were small. More details about this study follow.

In one of the experiments the portfolio of workflows included 184 elements, corresponding to 62 algorithms with default configurations, 30 variants with varied hyperparameter configurations (3 versions of MLP, 7 of SVM, 7 of RFs, 8 of J48, and 5 of  $k$ -NN) and the same number of variants (62+30), which were created by also including feature selection (CFS method (Hall, 1999)). One hundred datasets were used from the OpenML benchmark suite (Vanschoren et al., 2014) in a leave-one-out mode in the evaluation.

Auto-WEKA was run and only the first recommendation was used for each dataset. The Auto-WEKA total runtime was computed by adding the search time to the recommended model runtime. This runtime was used to retrieve the actual performance of the AR\* system. The results for four different time budgets (in minutes) are shown in Table 7.2.

### Portfolios of successful workflows

The concept of a *portfolio of workflows* can be seen as generalizations of the concept of a *portfolio of algorithms*. So a question arises as to which workflows should be kept for future use.

In Chapter 8 we address the issue of how to set up configuration spaces and portfolios. Interested readers can consult this chapter to find answers to this issue.

## References

- Anthony, T., Tian, Z., and Barber, D. (2017). Thinking fast and slow with deep learning and tree search. In *Conference on Neural Information Processing Systems*.
- Bernstein, A. and Provost, F. (2001). An intelligent assistant for the knowledge discovery process. In Hsu, W., Kargupta, H., Liu, H., and Street, N., editors, *Proceedings of the IJCAI-01 Workshop on Wrappers for Performance Enhancement in KDD*.
- Brazdil, P. (1984). Use of derivation trees in discrimination. In O’Shea, T., editor, *ECAI 1984 - Proceedings of 6th European Conference on Artificial Intelligence*, pages 239–244. North-Holland.
- Brazdil, P. and Henery, R. J. (1994). Analysis of results. In Michie, D., Spiegelhalter, D. J., and Taylor, C. C., editors, *Machine Learning, Neural and Statistical Classification*, chapter 10, pages 175–212. Ellis Horwood.
- Cachada, M., Abdulrahman, S., and Brazdil, P. (2017). Combining feature and algorithm hyperparameter selection using some metalearning methods. In *Proc. of Workshop AutoML 2017, CEUR Proceedings Vol-1998*, pages 75–87.

- Chandrasekaran, B. and Jopheson, J. (1999). What are ontologies, and why do we need them? *IEEE Intelligent Systems*, 14(1):20–26.
- Clark, P. and Niblett, T. (1989). The CN2 induction algorithm. *Machine Learning*, 3(4):261–283.
- Diamantini, C., Potena, D., and Storti, E. (2012). KDDONTO: An ontology for discovery and composition of KDD algorithms. In *Proceedings of the ECML-PKDD'09 Workshop on Service-oriented Knowledge Discovery*, pages 13–24.
- Drori, I., Krishnamurthy, Y., Rampin, R., de Paula Lourenco, R., Ono, J. P., Cho, K., Silva, C., and Freire, J. (2018). AlphaD3M: Machine learning pipeline synthesis. In *Workshop AutoML 2018 @ ICML/IJCAI-ECAI*. Available at site <https://sites.google.com/site/automl2018icml/accepted-papers>.
- Duda, R. O., Hart, P. E., and Stork, D. G. (2001). *Pattern Classification (2 ed.)*. John Wiley & Sons, New York.
- Engels, R., Lindner, G., and Studer, R. (1997). A guided tour through the data mining jungle. In *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining*, pages 163–166. AAAI.
- Euler, T. (2005). Publishing operational models of data mining case studies. In *Proceedings of the ICDM Workshop on Data Mining Case Studies*, pages 99–106.
- Euler, T., Morik, K., and Scholz, M. (2003). MiningMart: Sharing successful KDD processes. In *LLWA 2003 – Tagungsband der GI-Workshop-Woche Lehren–Lernen–Wissen–Adaptivitat*, pages 121–122.
- Euler, T. and Scholz, M. (2004). Using ontologies in a KDD workbench. In *Proceedings of the ECML/PKDD Workshop on Knowledge Discovery and Ontologies*, pages 103–108.
- Feurer, M., Klein, A., Eggenberger, K., Springenberg, J., Blum, M., and Hutter, F. (2015). Efficient and robust automated machine learning. In Cortes, C., Lawrence, N., Lee, D., Sugiyama, M., and Garnett, R., editors, *Advances in Neural Information Processing Systems 28*, NIPS'15, pages 2962–2970. Curran Associates, Inc.
- Fikes, R. E. and Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3–4):189–208.
- Georgievski, I. and Aiello, M. (2015). HTN planning: Overview, comparison, and beyond. *Artificial Intelligence*, 222:124–156.
- Ghallab, M., Nau, D. S., and Traverso, P. (2004). *Automated planning - theory and practice*. Elsevier.
- Gijsbers, P. and Vanschoren, J. (2019). GAMA: Genetic automated machine learning assistant. *Journal of Open Source Software*, 4(33):1132.
- Gil, Y., Yao, K.-T., Ratnakar, V., Garijo, D., Steeg, G. V., Szekely, P., Brekelmans, R., Kejriwal, M., Luo, F., and Huang, I.-H. (2018). P4ML: A phased performance-based pipeline planner for automated machine learning. In *Workshop AutoML 2018 @ ICML/IJCAI-ECAI*. Available at site <https://sites.google.com/site/automl2018icml/accepted-papers>.
- Giraud-Carrier, C. (2005). The Data Mining Advisor: Meta-learning at the Service of Practitioners. In *Proceedings of the International Conference on Machine Learning and Applications (ICMLA)*, page 113–119.
- Gomes, C. P. and Selman, B. (2001). Algorithm portfolios. *Artificial Intelligence*, 126(1–2):43–62.
- Gordon, D. and desJardins, M. (1995). Evaluation and selection of biases in machine learning. *Machine Learning*, 20(1/2):5–22.
- Hall, M. (1999). *Correlation-based feature selection for machine learning*. PhD thesis, University of Waikato.

- Hilario, M., Kalousis, A., Nguyen, P., and Woznica, A. (2009). A data mining ontology for algorithm selection and meta-mining. In *Proceedings of the ECML-PKDD'09 Workshop on Service-Oriented Knowledge Discovery*, page 76–87.
- Hopcroft, J. E. and Ullman, J. D. (1979). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley.
- Kietz, J., Serban, F., Bernstein, A., and Fisher, S. (2009). Towards cooperative planning of data mining workflows. In *Proceedings of ECML-PKDD'09 Workshop on Service Oriented Knowledge Discovery*, pages 1–12.
- Kietz, J.-U., Serban, F., Bernstein, A., and Fischer, S. (2012). Designing KDD-Workflows via HTN-Planning for Intelligent Discovery Assistance. In Vanschoren, J., Brazdil, P., and Kietz, J.-U., editors, *PlanLearn-2012, 5th Planning to Learn Workshop WS28 at ECAI-2012, Montpellier, France*.
- Kotthoff, L., Thornton, C., Hoos, H. H., Hutter, F., and Leyton-Brown, K. (2016). AutoWEKA 2.0: Automatic model selection and hyperparameter optimization in WEKA. *Journal of Machine Learning Research*, 17:1–5.
- Leyton-Brown, K., Nudelman, E., Andrew, G., McFadden, J., and Shoham, Y. (2003). A portfolio approach to algorithm selection. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 1542–1543.
- Lindner, G. and Studer, R. (1999). AST: Support for algorithm selection with a CBR approach. In Giraud-Carrier, C. and Pfahringer, B., editors, *Recent Advances in Meta-Learning and Future Work*, pages 38–47. J. Stefan Institute.
- Linz, P. (2011). *An Introduction to Formal Languages and Automata*. Jones & Bartlett Publishers.
- Martin, J. C. (2010). *Introduction to Languages and the Theory of Computation (4th ed.)*. McGraw-Hill.
- McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., and Wilkins, D. (1998). PDDL—the planning domain definition language. Technical report, New Haven, CT: Yale Center for Computational Vision and Control.
- Michie, D. (1968). Memo functions and machine learning. *Nature*, 2018:19–22.
- MiningMartCB (2003). MiningMart Internet case base. <http://mmart.cs.uni-dortmund.de/end-user/caseBase.html>.
- Mitchell, T. (1982). Generalization as Search. *Artificial Intelligence*, 18(2):203–226.
- Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill.
- Mohr, F. and Wever, M. (2021). Naive automated machine learning—a late baseline for automl. *arXiv preprint arXiv:2103.10496*.
- Mohr, F., Wever, M., and Hüllermeier, E. (2018). ML-plan: Automated machine learning via hierarchical planning. *Machine Learning*, 107(8-10):1495–1515.
- Morik, K. and Scholz, M. (2004). The MiningMart approach to knowledge discovery in databases. In Zhong, N. and Liu, J., editors, *Intelligent Technologies for Information Analysis*, chapter 3, pages 47–65. Springer. Available from <http://www-ai.cs.uni-dortmund.de/MMWEB>.
- Nguyen, P., Hilario, M., and Kalousis, A. (2014). Using meta-mining to support data mining workflow planning and optimization. *Journal of Artificial Intelligence Research*, 51:605–644.
- Olson, R. S., Bartley, N., Urbanowicz, R. J., and Moore, J. H. (2016). Evaluation of a tree-based pipeline optimization tool for automating data science. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pages 485–492.
- Patel-Schneider, P., Hayes, P., and Horrocks, I. e. a. (2004). OWL web ontology language semantics and abstract syntax. W3C recommendation 10.

- Phillips, J. and Buchanan, B. G. (2001). Ontology-guided knowledge discovery in databases. In *Proceedings of the First International Conference on Knowledge Capture*, pages 123–130.
- Piatetsky-Shapiro, G. (1991). Knowledge discovery in real databases. *AI Magazine*.
- Russell, S. J. and Norvig, P. (2016). *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3rd edition.
- Sacerdoti, E. (1974). Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5(2):115–135.
- Serban, F., Vanschoren, J., Kietz, J., and Bernstein, A. (2013). A survey of intelligent assistants for data analysis. *ACM Comput. Surv.*, 45(3):1–35.
- Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., and et al., T. G. (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. In *Conference on Neural Information Processing Systems*.
- Thornton, C., Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2013). Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 847–855. ACM.
- Vanschoren, J., Blockeel, H., Pfahringer, B., and Holmes, G. (2012). Experiment databases: a new way to share, organize and learn from experiments. *Machine Learning*, 87(2):127–158.
- Vanschoren, J., van Rijn, J. N., Bischl, B., and Torgo, L. (2014). OpenML: networked science in machine learning. *ACM SIGKDD Explorations Newsletter*, 15(2):49–60.
- Wever, M., Mohr, F., and Hüllermeier, E. (2018). ML-plan for unlimited-length machine learning pipelines. In *AutoML Workshop at ICML-2018*.
- Wirth, R., Shearer, C., Grimmer, U., Reinartz, T. P., Schlosser, J., Breitner, C., Engels, R., and Lindner, G. (1997). Towards process-oriented tool support for knowledge discovery in databases. In *Proceedings of the First European Conference on Principles and Practice of Knowledge Discovery in Databases*, pages 243–253.
- Žáková, M., Křemen, P., Železný, F., and Lavrač, N. (2011). Automating knowledge discovery workflow composition through ontology-based planning. *IEEE Transactions on Automation Science and Engineering*, 8:253–264.

**Advanced Techniques and Methods**



## Setting Up Configuration Spaces and Experiments

**Summary.** This chapter discusses the issues relative to so-called configuration spaces that need to be set up before initiating the search for a solution. It starts by introducing some basic concepts, such as discrete and continuous subspaces. Then it discusses certain criteria that help us to determine whether the given configuration space is (or is not) adequate for the tasks at hand. One important topic which is addressed here is *hyperparameter importance*, as it helps us to determine which hyperparameters have a high influence on the performance and should therefore be optimized. This chapter also discusses some methods for reducing the configuration space. This is important as it can speed up the process of finding the potentially best workflow for the new task. One problem that current systems face nowadays is that the number of alternatives in a given configuration space can be so large that it is virtually impossible to gather complete metadata. This chapter discusses the issue of whether the system can still function satisfactorily even when the metadata is incomplete. The final part of this chapter discusses some strategies that can be used for gathering metadata that originated in the area of multi-armed bandits, including, for instance, SoftMax, upper confidence bound (UCB) and pricing strategies.

### 8.1 Introduction

The configuration space includes all possible workflows (pipelines) that can be constructed by combining the given set of base-level algorithms with all admissible configurations of their hyperparameters.

The search space is of great influence to the result of the hyperparameter optimization algorithm (Yu et al., 2020; Yang et al., 2020). Designing this configuration space too small can be harmful, as the search procedure might not yield good workflows for some datasets. On the other hand, designing this configuration space too large can also be problematic, as the search procedure might require a lot of time before it converges to something good. This chapter aims to address the issue of how to set up an adequate configuration space.

#### Organization of this chapter

Section 8.2 clarifies some basic notions that are useful when discussing configuration spaces. First, it discusses the difference between discrete and continuous subspaces. Then

the issue of sampling the continuous subspaces is taken up. Finally, we address the issue of how configuration spaces can be described.

Section 8.3 discusses certain criteria that enable us to affirm whether the given configuration space is (or is not) adequate for the tasks at hand.

Section 8.4 discusses the notion of *hyperparameter importance*. This notion helps us to determine which hyperparameters have a high influence on performance and should therefore be optimized. Hyperparameters that have a relatively small influence on performance could potentially be neglected, or else, less resources could be allocated to them.

Section 8.5 discusses some methods for reducing the configuration space. This is important as it can speed up the process of finding the potentially best workflow for each new task. This is simply because the system does not need to consider the alternatives that do not make a difference. Having a simple set up also has other advantages. Many users may want to know why a given recommendation system has come up with a particular solution, following the trend called *explainable AI* (Došilović et al., 2018). Obviously, if the system is simpler, so are the explanations that can be given.

The success of a particular recommendation system also depends on the datasets used in the process of generating metadata. The issue of which datasets are needed is discussed in Section 8.7. One problem that current systems face nowadays is that the number of alternatives in a given configuration space can be so large that it is virtually impossible to gather complete metadata.

Section 8.8 discusses the issue of whether the system can still function satisfactorily even when the metadata is incomplete.

The final section (Section 8.9) discusses some strategies that can be used for gathering metadata that originated in the area of multi-armed bandits.

## 8.2 Types of Configuration Spaces

The term *configuration space* is used here to refer to the search space associated with a particular metalearning task. In this context we can distinguish among the following metalearning tasks:

- algorithm selection
- hyperparameter optimization and the combined hyperparameter optimization and algorithm selection (CASH) problem
- workflow design

Each of these tasks involves certain configuration space. The details on each are given in the following subsections.

### 8.2.1 Configuration spaces associated with algorithm selection

The configuration space associated with base-level algorithm selection consists of a set of base-level algorithms, which is usually referred to as a *portfolio*. The number of items in this portfolio determines the size of this space. In practical applications, there is a finite number of possibilities. As such, this is a *discrete* search space. Typically, there is only a finite number of values that can be transferred from one situation to another.

Chapters 2 and 5 described different ways of searching through this space. Section 8.5 describes a method for reducing this space, which is done by eliminating certain algorithms from the existing portfolio.



## 8.2.2 Configuration spaces associated with hyperparameter optimization and CASH

Base-level algorithms typically involve hyperparameters. Each algorithm has specific hyperparameters.

### Types of hyperparameters

Some hyperparameters are categorical and hence discrete. Some examples of this type are: the type of SVM kernel, the sampling method of a random forest, or the distance function in a  $k$ -NN classifier.

Other hyperparameters are continuous. Some examples of continuous hyperparameters are: the kernel width of a SVM, the learning rate of a neural network, or the number of trees in a random forest.

Some algorithms include both categorical/discrete and continuous hyperparameters. In many systems, categorical and continuous hyperparameters are mixed together, as is done, for example, in the configuration space of Auto-sklearn (Feurer et al., 2015, 2019).

### Continuous versus discrete spaces

The type of hyperparameter determines what kind of configuration space is involved in the associated task.

Discrete spaces consist of a fixed number of configurations, whereas continuous spaces consist potentially of an infinite number of configurations. Continuous spaces can be discretized. This has an advantage that it is relatively easy to gather metaknowledge on previous experiments, as there is only a finite number of configurations. The choice regarding whether to discretize (or not) is usually made by the designer of the configuration space.

So, typically we could say that hyperparameter optimization involves both *discrete* and *continuous spaces*.

### Conditional hyperparameters and spaces

Sometimes, a set of hyperparameters is dependent on the specific value of another hyperparameter. These hyperparameters are called *conditional hyperparameters*. For example, many hyperparameters of support vector machines that control the kernel will only be relevant if a certain kernel is selected. As a more complex example, consider the Auto-sklearn search space (Feurer et al., 2015, 2019), where the set of hyperparameters is the union of the hyperparameters of all algorithms involved, plus additional hyperparameters determining which algorithms and preprocessing operators should be selected. All the hyperparameters are conditional on the values of the latter ones.

### Sampling of continuous subspaces

Let us address a few issues that are relevant when dealing with numeric hyperparameters:

1. Type of sampling (uniform, log scale or other)
2. Level of detail

Continuous spaces are often sampled according to a chosen probability distribution. Some hyperparameters can be sampled uniformly, while others are sampled on a log scale, which seems more appropriate for others. For example, let us consider the task of selecting the number of trees for a gradient boosting classifier. We note that the change from 10 to 20 trees might produce a significant effect on performance, while the change from 1000 to 1010 trees would hardly lead to a significant effect. So this justifies the adoption of log-scale sampling for this parameter, which can form part of the set up. Snoek et al. (2014) proposed a method that determines what the optimal sampling rate is for every hyperparameter, freeing the user from this task.

Regarding the level of detail, numeric hyperparameters are often specified by the interval between the minimum and maximum value. Supposing we opt for uniform sampling in this range, the question is whether we should admit all possible values in this range or just some of them. On the one hand, we want to have a sufficiently fine resolution, so that we could observe all effects in performance. On the other hand, using too fine a resolution can complicate the search for the best setting.

Since the sampling is done according to a given probability distribution, sampling methods can keep running until a given time budget has been exhausted. The methods based on multi-armed bandits discussed in Chapter 8 (Section 8.9) provide other alternatives.

### 8.2.3 Configuration spaces associated with workflow design

Workflows (pipelines) are often defined as collections of steps (e.g., pre-processing steps, base-level algorithm) chained together. It is important to keep the size of this configuration space manageable. In general, certain formal structures including ontologies, grammars, or planning systems with operators can be used for this purpose (see Chapter 7). Each of these formal structures defines a configuration space. We note that Auto-sklearn (Feurer et al., 2015, 2019) uses a certain ontology that dictates the order and applicability of operators to include in a workflow (pipeline).

The aim is to design this space so that it would include all the required alternatives (here alternative workflows). It should not be unnecessarily “too big”, as this could make the search for potentially best solution (here workflow) more difficult. This issue regarding whether the given configuration space is adequate for a given set of tasks is discussed in the next section.

## 8.3 Adequacy of Configuration Spaces for Given Tasks

Let  $T_C$  represent the tasks that can be solved adequately by a system that has access to configuration space  $C$ . Similarly, let  $T_R$  represent various tasks that we expect to encounter in the near future. Let us suppose, that to solve them in an adequate way, we would need configuration space  $R$ .

Note that  $R$  is a hypothetical concept meant for illustrative purposes. We can never know the exact contents of  $R$ , as we will not know what tasks we will encounter in the future. But we may know some instances of  $R$ , namely the tasks we have encountered until now. Having the knowledge of some instances enables us to make a guess regarding the underlying distribution of tasks. This way may cover also future tasks under the assumption of stationarity, i.e. that the distribution will not change in the future. The following situations can arise:

- $R \equiv C$
- $R \subset C$
- $C \subset R$
- $ExCond \wedge C \cap R \neq \emptyset$

where  $ExCond$  guarantees that the three above cases are excluded. This condition can be defined as  $ExCond = (R \neq C) \wedge (R \not\subset C) \wedge (C \not\subset R)$ .

Let us analyze each case separately.

**Case  $R \equiv C$ :** If this case occurs, then we are well prepared for  $T_R$ . Nothing needs to be done.

**Case  $R \subset C$ :** If this case arises, at first sight it seems that everything is fine, as in principle we could solve all the required tasks. However, as not all elements in  $C$  may be needed, time may be wasted in the search for the right algorithm. For instance, if the required task is to distinguish between two classes only and the configuration space includes methods suitable for any number of classes, it is possible to pre-select the appropriate subset.

However, our configuration space may also include algorithms that have sub-standard performance (non-competitive algorithms), or algorithms with good performance but which are redundant. So, the objective here is to identify a reduced configuration space  $C'$  which can still solve all the required tasks, namely  $(R \equiv C') \wedge (C' \subset C)$ . Reducing the configuration space is beneficial in general, as it reduces the time required in the search for the potentially best solution. However, this process incurs certain risks. If it is reduced too much, the optimal method (e.g., classifier) may be missed out. Section 8.5 discusses the reduction method in detail.

**Case  $C \subset R$ :** In this case the problem is more serious, as we are unable to solve all the required tasks with recourse to the algorithms in the current configuration space. For instance, if our configuration space includes only methods for classification tasks and the required tasks include both classification and regression, then it is necessary to extend the current portfolio by the inclusion of appropriate methods.

**Case  $ExCond \wedge C \cap R \neq \emptyset$ :** Let us call this intersection  $C'$ . This means that only some problems in  $R$  can be solved with  $C'$  (as in case  $C \subset R$ ). This means that only some problems in  $R$  can be solved with  $C'$  (as in case  $C \subset R$ ). Moreover,  $C'$  includes some algorithms that are not really needed for  $R$  (as in case  $R \subset C$ ).

### 8.3.1 General principles for the constitution of configuration spaces

It is possible to define a set of general principles that should be followed in the constitution of discrete configuration spaces. For continuous configurations, the situation is a bit different as it is defined by ranges, rather than a set of configurations.

Let us assume that a set of datasets and tasks is given. Let us consider a configuration space containing the following set of alternatives  $C = c_1, \dots, c_m$ , where each  $c_i$  represents a workflow with some specific hyperparameter settings. The general principles for including an element  $c_i$  are:

1. **Minimal relevance:** For most datasets there should be a  $c_i$  that obtains better performance than a suitable baseline. A *baseline* is a simple method which establishes a reference for minimally acceptable results. In supervised learning problems, for instance, it consists of predicting the most frequent class.
2. **Positive marginal contribution/individual relevance:** Each item  $c_i$  should provide some marginal contribution to the set  $C$  without  $C_i$  ( $C - c_i$ ). Effectively this means that  $c_i$  should be the best option for at least one task.

3. **Impossibility to improve on marginal contribution:** Given some preselected set  $C$ , the results cannot be further significantly improved by adding additional elements  $c_j$  to it.
4. **Impossibility to improve on individual relevance:** For every  $c_i$  there should not exist a  $c_j$  such that the performance of  $c_i$  is never significantly better than that of  $c_j$  for all tasks considered.

We note that the principles are oriented towards the available datasets and tasks. In other words, as we do not know the tasks that we will encounter, this is usually established for the metadata of previously seen tasks. If we want to be well prepared for future datasets, it is advisable to relax somewhat the last two principles. Some competitive algorithms may be useful in future tasks, even if they have not been included in the best equivalence group on any past dataset.

Some of these issues are taken up again in subsequent subsections, where we discuss specific methods for constructing portfolios of algorithms (workflows).

## 8.4 Hyperparameter Importance and Marginal Contribution

In this section we cover the marginal contribution of certain elements in a given configuration space. Subsection 8.4.1 discusses the marginal contribution of algorithms (workflows), while Subsection 8.4.2 is oriented towards hyperparameter importance for a specific dataset. Subsection 8.4.3 generalizes the notion of hyperparameter importance across datasets.

### 8.4.1 Marginal contribution of algorithms (workflows)

Some researchers have investigated the issue of assessing the complementarity of algorithms. Xu et al. (2012), for instance, have defined a notion of *marginal contribution* to the performance, i.e., how much the performance of an existing portfolio is improved by adding a new algorithm to it. This approach has the disadvantage of being dependent on a fixed portfolio. A broader view of an algorithm contribution, which extends the marginal contribution analysis, involves a so-called *Shapley value* (Fréchette et al., 2016). This value determines the marginal contribution of an algorithm to any subset of the algorithm portfolio.

Shapley values come from the area of cooperative game theory. The setup involves a coalition of players who cooperate and obtain a certain overall gain from that cooperation. As some players may contribute more to the coalition than others or may possess different bargaining power (for example, threatening to destroy the whole surplus), a question arises of how important each player is to the overall cooperation and what the payoff is. The Shapley value provides one possible answer to this question.

The techniques described in this section work on continuous configuration spaces.

### 8.4.2 Determining hyperparameter importance on a given dataset

The problem of how to automatically optimize hyperparameters of algorithms has drawn a lot of attention in recent years. The reader can consult Chapter 6, where some of the more important methods are discussed. Most of these techniques require that the relevant hyperparameters for each algorithm are given and accompanied by a specification

of the possible settings that can be considered. This can be done either by specifying the intervals of values or simply by enumerating all possible settings.

Various techniques exist that enable to identify, for a given dataset and algorithm, the most important hyperparameters. These include approaches using

- *Forward selection* (Hutter et al., 2013)
- *functional ANOVA* (Sobol, 1993; Hutter et al., 2014)
- *Ablation analysis* (Biedenkapp et al., 2017; Fawcett and Hoos, 2016)

The three techniques described further on enable us to determine the importance of hyperparameters for a given dataset. All require metadata about configurations and the corresponding performance values on the particular dataset. Note that these techniques operate on a continuous configuration space.

### Forward selection

Forward selection (Hutter et al., 2013) trains a surrogate model to map hyperparameter values to performance values. The work assumes that including important hyperparameters as input to the surrogate model has a high positive impact on performance. This setting is similar to the experiment of Breiman (2001) on feature importance.

The method starts with an empty set, and greedily adds the hyperparameter that has the highest impact on the predictive performance of the surrogate model. This process continues in an iterative manner until no further improvement can be made. This yields a ranking of hyperparameters ordered by importance to this surrogate model.

### Ablation analysis

Ablation analysis (Fawcett and Hoos, 2016) calculates a so-called *ablation trace*. The method first executes the default configuration on the dataset and establishes its performance. Afterwards, it determines the optimal hyperparameter setting using an optimization procedure, such as sequential model-based optimization (SMBO) discussed in Chapter 6 (Section 6.8). The goal is to determine which of the hyperparameters influence performance the most when changing the value from the default setting to the optimal setting. It starts with the optimal configuration, and considers all configurations that can be reached from the optimal configuration by switching one hyperparameter value to the default value. It continues in that fashion until all hyperparameters have been switched and the default configurations has been reached. This results in a ranking of hyperparameters in decreasing importance. This is called the ablation trace. Note that this method requires that, after the optimal hyperparameters have been determined, several models along the ablation trace are trained. This makes the procedure potentially time consuming. Biedenkapp et al. (2017) show how surrogate models can be used to avoid this and run ablation analysis faster.

### Functional ANOVA

Hutter et al. (2014) apply functional ANOVA (Sobol, 1993) to establish hyperparameter importance. Functional ANOVA determines how much each hyperparameter (and each combination of hyperparameters) contributes to the variance of the performance. It works on the concept of the *marginal* of a hyperparameter. In the context of machine

learning, a marginal reflects the relation between the value of an hyperparameter and the performance of that algorithm. Specifically, for each value for that hyperparameter, it reflects how the algorithm would perform, averaged over all possible combinations of the other hyperparameters and their settings. This might not seem feasible, as there are an exponential number of combinations. However, Hutter et al. (2014) showed how this can be calculated efficiently using tree-based surrogate models trained on the performance data of configurations on the dataset. Hyperparameters that have a large variance across the marginal are deemed important. The opposite is also true: hyperparameters that have a low variance are considered unimportant. Note that functional ANOVA determines the importance of hyperparameters globally; the conclusions that are drawn do not depend on specific values of other hyperparameters.

### 8.4.3 Establishing hyperparameter importance across datasets

All three aforementioned techniques are post hoc techniques; i.e., when confronted with a new dataset, they do not reveal which hyperparameters are important prior to experimenting on that particular dataset. In this subsection, we describe efforts made towards establishing hyperparameter importance across datasets.

In order to gain a better understanding about which hyperparameters are important in general, van Rijn and Hutter (2018) and Probst et al. (2019) apply the following procedure:

1. Determine a suitable set of datasets;
2. Gather ample configurations and their performance on these datasets;
3. Apply a hyperparameter importance framework on it;
4. Aggregate the results in a human-understandable format.

For determining a suitable set of datasets, there are several considerations to take into account. On the one hand, it would be interesting to consider a broad set of datasets. For example, the OpenML-CC18 (Bischl et al., 2021) seems a suitable choice. However, in some specific studies, it makes sense to consider only a subset of datasets. For instance, Probst et al. (2019) are specifically interested in binary classification and hence use a subset of the “OpenML-100” with a binary target. Similarly, Sharma et al. (2019) are interested in image classification, and therefore define a set of ten image datasets.

As most methods for establishing hyperparameter importance rely on the use of a surrogate model, we need to have ample data gathered on the given datasets. The data consist of pairs of items, namely a configuration and the corresponding measure of performance. As surrogates become increasingly more accurate when trained on a larger number of configuration and performance pairs, we need a sufficient number of such pairs. As for the methods for establishing hyperparameter importance, all the aforementioned techniques (forward selection, ablation analysis, and functional ANOVA) can be used. However, all these methods are based on certain assumptions, and as such, the results may differ depending on which choice was made. Still, a quick investigation of the results shows that the methods based on tunability and functional ANOVA discussed earlier seem to agree on the most important hyperparameters (van Rijn and Hutter, 2018; Probst et al., 2019). Chapter 17 looks at this in more detail.

Regarding the aggregation of results, one could simply aggregate the results per hyperparameter and per dataset into a boxplot. Functional ANOVA returns a clear and interpretable fraction, representing the contribution to the overall variance. For ablation analysis and forward selection, the outcome is in the form of a ranking of hyperparameters according to their importance. Then a ranking-based aggregation can be used, which can lead to *critical distance* plots, as suggested by Demšar (2006).

## 8.5 Reducing Configuration Spaces

### 8.5.1 Reducing portfolios of algorithms/configurations

The issue of how to identify and eliminate certain algorithms (configurations, workflows) from a given portfolio and evaluate the effects was addressed by Brazdil et al. (2001) and Abdulrahman et al. (2019). The method involves two steps. The aim of the first step is to identify competitive algorithms. The non-competitive algorithms are effectively dropped at this stage. The aim of the second step is to seek a small number of *specialists/experts* for each dataset. This step results in the elimination of many potentially redundant algorithms.

More details on both steps are given in the following subsections. Both methods are defined for discrete configuration spaces.

#### Identifying competitive algorithms

The assumption followed here is that the non-competitive algorithms have little chance to achieve a competitive result on the new target dataset. This is done by applying Algorithm 8.1, which calls Algorithm 8.2.

**input** : Datasets  $D_s$   
 Portfolio of algorithms  $A_{in}$   
**output**: Portfolio of competitive algorithms  $A_c$

- 1: Initialize  $A_c$  to empty list
- 2: **for all**  $d_i \in D_s$  **do**
- 3:    $A_{c_i} \leftarrow$  Identify competitive algorithm ( $A_{in}, d_i$ )
- 4:    $A_c \leftarrow A_c + A_{c_i}$
- 5: **end for**

**Algorithm 8.1:** Identifying competitive algorithms for all datasets

Algorithm 8.1 requires as input the datasets  $D_s$  and a set of algorithms  $A_{in}$  and outputs a subset of algorithms  $A_c$ . The for-loop (lines 2–5) includes a call to Algorithm 8.2 (on line 3), which returns list  $A_{c_i}$  representing the most competitive algorithms of  $A_{in}$  for the dataset  $d_i$ . Any performance measure can be used to identify such algorithms. It can be, for instance, accuracy, or as Abdulrahman et al. (2019) have shown, a combined measure of accuracy and runtime. This list includes the topmost algorithm in the average ranking and all algorithms with equivalent performance. Finally, list  $A_{c_i}$  is added (using the operator  $+$ ) to the  $A_c$ , representing a list of lists.

Algorithm 8.2 works as follows: First,  $A_{c_i}$  is initialized to an empty list. Then the ranking  $R_{d_i}$  is constructed on the basis of the test results of the algorithms of  $A_{in}$  on dataset  $d_i$ . There is no need to conduct tests, as the test results can be simply retrieved from the meta-database. The next goal is to initialize  $a_{best}$  to be the topmost algorithm in  $R_{d_i}$ .

The method proceeds by identifying all algorithms ( $a_{eq}$ ) with an equivalent performance (e.g., a combined measure of accuracy and runtime) to  $a_{best}$ . This is done by processing all algorithms (configurations)  $a_j \in A_{in}$ , and conducting a statistical test

- input** : Algorithms  $A_{in}$ , Dataset  $d_i$   
**output**: Competitive algorithms  $A_{c_i}$
- 1: Initialize  $A_{c_i}$  to empty list
  - 2: Construct a ranking  $R_{d_i}$  of algorithm in  $A_{in}$  on  $d_i$
  - 3: Identify the topmost algorithm  $a_{best}$  in  $R_{d_i}$
  - 4: Use a statistical test to identify algorithms in  $a_{eq}$  with equivalent performance
  - 5:  $A_{c_i} \leftarrow a_{best} + a_{eq}$

**Algorithm 8.2:** Identify competitive algorithms for a specific dataset

(e.g., Wilcoxon signed-rank test with a confidence level of 95% (Demšar, 2006)) to determine whether the performance of  $a_j$  is equivalent to  $a_{best}$ . This test is done on the basis of *fold* information of the cross-validation procedure that is available in the meta-database. All algorithms that have an equivalent performance to  $a_{best}$  are included in the candidate set  $A_{c_i}$ . When all pairs of algorithms have been processed, list  $A_{c_i}$  is returned.

**Example**

To explain how the non-competitive algorithms are removed from the ranking, as detailed in Algorithm 8.1, we show an example. For simplicity, our example includes only six algorithms ( $a_1, a_2, \dots, a_6$ ) and six datasets ( $D_1, D_2, \dots, D_6$ ) (see Table 8.1). The algorithms in dark-grey slots represent the topmost algorithms ( $a_{best}$ ) identified for each dataset.

The topmost algorithm for each dataset is then compared with the other algorithms in the ranking. For example, when dealing with  $R_{D_1}$ , algorithm  $a_2$  is identified as  $a_{best}$ . It is compared with  $a_6, a_4, a_3, a_5,$  and  $a_1$  using the Wilcoxon signed-rank test. Any algorithm that has a similar performance to the topmost algorithm is maintained in the ranking, while all the others are dropped. In Table 8.1 the algorithms that have been maintained are shown in grey slots.

This process is repeated for all datasets used in the experiment, and the competitive algorithms for each dataset are identified. In our example the competitive set of

Table 8.1: Competitive algorithms (in gray) identified using statistical test

Rank	$R_{D_1}$	$R_{D_2}$	$R_{D_3}$	$R_{D_4}$	$R_{D_5}$	$R_{D_6}$
1	$a_2$	$a_5$	$a_4$	$a_6$	$a_4$	$a_1$
2	$a_6$	$a_1$	$a_2$	$a_5$	$a_2$	$a_2$
3	$a_4$	$a_3$	$a_1$	$a_1$	$a_3$	$a_4$
4	$a_3$	$a_6$	$a_5$	$a_2$	$a_5$	$a_5$
5	$a_5$	$a_4$	$a_3$	$a_3$	$a_6$	$a_3$
6	$a_1$	$a_2$	$a_6$	$a_4$	$a_1$	$a_6$

algorithms includes

$$A_c = \{(a_2)(a_5, a_1)(a_4, a_2, a_1)(a_6, a_5, a_1)(a_4, a_2)(a_1)\}. \tag{8.1}$$



If we eliminate the duplicates, we obtain

$$A_c = \{a_2, a_5, a_1, a_4, a_5\}. \quad (8.2)$$

A question arises as to whether all these algorithms are needed or whether we can still drop some. This issue is covered in the next section.

### Using covering algorithm to select “non-redundant” algorithms

If a portfolio is constructed by adding various algorithms to it, it could include various versions of the same algorithm with very similar performance. Their inclusion in the algorithm portfolio may not really be desirable.

The issue of whether two algorithms (configurations) have similar performance can be determined on a macro- or micro-level. Comparisons on a macro-level involve measures (e.g., accuracy or area under the ROC curve) on the whole dataset. They represent aggregated measures across different examples. One approach that uses this notion of similarity is discussed in the next subsection.

Alternatively, it is possible to exploit the notion of similarity on a micro-level, i.e., by taking into account the performance on individual examples. This approach is discussed further on.

Both approaches seek to “cover” each dataset by at least one algorithm. The term “algorithm covers a dataset” is used here to indicate that the algorithm appears in the subset of the algorithms with the best performance. All algorithms identified as “similar” are dropped.

In addition, both approaches give preference to algorithms that cover preferably many datasets. This is based on the assumption, that by assembling a set of algorithms that work well for many past datasets, it is likely that one of these will be a good choice for the new datasets.

### Using covering algorithm with macro-level similarity

This approach of Abdulrahman et al. (2019) followed a covering approach and a hill-climbing strategy. In the first step, the algorithm that covers the largest number of datasets is selected. This approach assumes that it is sufficient to cover each dataset using just one algorithm. All other algorithms with a similar macro-level performance are not considered and hence effectively dropped from further consideration. All datasets covered are eliminated from further consideration. The process is repeated in an iterative manner until all datasets have been covered. The authors have shown that a particular metalearning system, AR\* (see Chapter 2), can identify well-performing algorithms sooner with a reduced portfolio than similar metalearning systems that use a full (unreduced) portfolio.

We note that this approach may eliminate algorithms that appear similar on a macro-level but are rather different on a micro-level. This shortcoming can be avoided by adopting micro-level in the detection of similarity.

### Using a covering algorithm with micro-level similarity

One good measure to detect micro-level similarity is *classifier output difference* (COD) (Peterson and Martinez, 2005), which determines the proportion of cases on which two classifiers differ in their predictions. It is defined as follows:

$$COD(i, j, d) = \frac{|x \in d \text{ s.t. } \hat{f}_i(x) \neq \hat{f}_j(x)|}{|d|}. \quad (8.3)$$

Here,  $d$  is the dataset at hand and  $x$  represents an instance (example) from  $d$ . The two classifiers that are being compared are denoted by  $i$  and  $j$ , and their predictions on a given instance  $x$  are denoted by  $\hat{f}_i(x)$  and  $\hat{f}_j(x)$ , respectively. So if this measure is 0 (or very near to this value) it indicates that the two classifiers generate virtually the same predictions on the given dataset. Lee and Giraud-Carrier (2011) use this measure to determine a set of different classifiers, which would complement each other in a portfolio.

### 8.5.2 A reduction method oriented towards a combination of measures

The reduction method described above works with the combined measure of accuracy and time. However, we note that the relative importance of accuracy and runtime is fixed. This is done by setting the value of the parameter  $Q$  (see Eq. 2.3 in Chapter 2). So this represents a certain limitation, as only a certain region of points on the so-called *Pareto front* is considered important.

The concept of *Pareto front* (or *frontier*) comes from the area of multiobjective optimization (Miettinen, 1999). A solution is called *nondominated*, *Pareto optimal*, if none of the objective functions can be improved without degrading some of the other objective values. A set of Pareto-optimal solutions constitutes the Pareto front.

So a question arises as to how to extend the method discussed in the previous subsection to cover the algorithms on the Pareto front (or in its vicinity). We describe two approaches in the following subsections.

#### Approach based on an envelopment curve

This approach tries to identify points (algorithms) on the envelopment curve or in its vicinity. Each point (algorithm) is characterized by two coordinates: runtime and accuracy. Various methods exist that can identify the points on the envelopment curve. Brazdil and Cachada (2018) have used a relatively simple method that led to quite satisfactory results. This method first orders all points (algorithms) by their runtime. Then all points are examined one by one, and if the accuracy of the successor is higher than that of all its predecessor, it is assumed to represent a competitive algorithm. Only the competitive algorithms are returned.

Figure 8.1 shows an example of both an unreduced set of algorithms (blue points) and the corresponding reduced set (red triangles) for one dataset.

This approach can be regarded as a simple solution for a rather complex problem. Although it can identify the points on the Pareto front, it does not use any notion of an uncertainty band below this front.

## 8.6 Configuration Spaces in Symbolic Learning

### Version spaces

T. Mitchell has defined so-called *version spaces* in the context of concept learning (Mitchell, 1980, 1982, 1990, 1997). The version space contains all possible hypotheses consistent with the given positive and negative examples and shows that all these

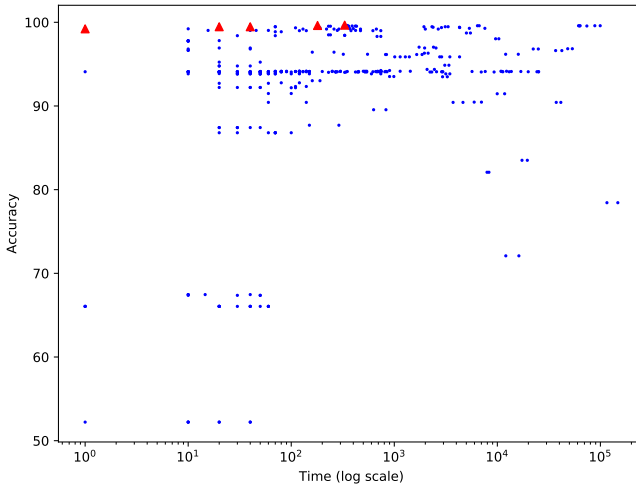


Fig. 8.1: Example of an unreduced set (blue points) and the corresponding reduced set (red triangles) of algorithms for one dataset

hypotheses can be organized in a lattice.<sup>1</sup> Thus, it is possible to follow the links from more specific to more general (or vice versa). Besides, as was shown, it is possible to define the general boundary  $G$  and the specific boundary  $S$ . Concept learning that exploits the version space proceeds by reducing the version space as each new example is analyzed.

Various researchers have considered the conditions needed by a given concept learning system so that it can generate the target hypothesis. In this context, the term *bias* was often used (Mitchell, 1990; Russell and Grosz, 1990b,a; Gordon and desJardins, 1995). Mitchell (1990) distinguishes between *weak* and *strong* bias, depending on whether weak or strong assumptions need to be made to be able to generate a model capable of classifying well all examples.

### Controlling the domain-specific language bias

Various researchers have proposed to control the language bias in various ways. For instance, these included *determinations* (Davies and Russell, 1987), *relational clichés* (Silverstein and Pazzani, 1991), *clause schemata* (Kramer and Widmer, 2001), *metapredicates* that define a translation between a *metafact* and a domain-level rule (Morik et al., 1993), and *topologies* (Morik et al., 1993), representing abstracted graphs of rules. Other

<sup>1</sup>A lattice consists of a partially ordered set of points in which every two elements have a unique supremum (also called a least upper bound) and a unique infimum (also called a greatest lower bound).

researchers have proposed various approaches based on grammars, including, for instance, the proposal of Cohen (1994). Some of these grammar-based approaches restrict the concepts that can be introduced (e.g., Jorge and Brazdil (1996)); Others impose restrictions on the variables also, such as the *DLAB* formalism (De Raedt and Dehaspe, 1997). Grammar-based formalisms were discussed in Chapter 7.

Although many proposals have been made in the past, to the best of our knowledge, no large-scale comparative studies have been carried out to determine which representation would be the best one. The areas of AutoML and metalearning open new possibilities, so it is conceivable that new comparative studies will be carried out in the future.

### Extending the domain-specific language bias

In concept learning, when the version space has shrunk and the resulting version space is empty, it can be taken as an indication that something must be altered. Mitchell (1990) suggests that, for instance, the representation language should be changed to also allow, for instance, disjunctive concepts apart from disjunctive ones. As was pointed out, this analysis is applicable only to noise-free data. However, it can be extended to noisy settings by assuming that the data may contain a certain given proportion of noisy examples, as, for instance, Mitchell (1977) and Hirsh (1994) have shown.

Here, we are concerned with enriching the descriptive language. Too many errors can be taken as a sign that the descriptors of cases should be extended and, consequently, it is possible to arrive at the required target concept. This was already suggested by Russell and Grosz (1990b):

*“It is important to note that the deductive process needs to be under some higher-level control in order to handle the case of the collapse of the version space when the observations are inconsistent with the initial concept language bias. In such contexts it becomes necessary to weaken the concept language bias, by relaxing the constraints on the form of concept definitions, or by extending the allowed predicate vocabulary”.*

## 8.7 Which Datasets Are Needed?

In Section 8.3 we discussed the relationship between tasks  $T_C$  that can be solved by a given system that can exploit the configuration space  $C$  and tasks  $T_R$  that we expect to encounter in the future. The main issue there was whether the given set of algorithms is adequate to solve  $T_R$ . Basically, we wish to have  $T_R \subset T_C$ .

We note that metalearning approaches require not only a set of algorithms, but also metaknowledge containing information regarding the performance on different datasets associated with particular task. As was shown in some of the previous chapters (e.g., Chapters 2 and 5), the metaknowledge is used to provide recommendations on the new dataset. This approach is successful if the new dataset and some dataset encountered in the past deal with the same *task*, such as classification that involves an unbalanced class distribution. In addition, the two datasets should be sufficiently similar. So, we need a sufficient number of datasets for each type of task we may encounter in the future.

Researchers and practitioners working in this area have used various strategies to come up with an answer to this problem. Some strategies that have been considered in the past are:

- Relying on existing repositories of datasets

- Generating synthetic datasets
- Generating variants of existing datasets
- Segmenting a large dataset or data stream
- Searching for datasets with discriminatory power

All the alternatives above are discussed in more detail in the following subsections.

### 8.7.1 Relying on existing dataset repositories

Currently, various dataset repositories exist from which different datasets may be retrieved. Some well-known repositories are:

- University of California at Irvine (UCI) Machine Learning Repository (Asuncion and Newman, 2007)
- OpenML (Vanschoren et al., 2014)
- UCI Knowledge Discovery in Databases Archive (Hettich and Bay, 1999)
- University of California at Riverside (UCR) Time Series Data Mining Archive
- UCR Time Series Data Mining Archive (Keogh and Folias, 2002)

among others. In the UCI repository, there are several hundred datasets. Although this is sufficient for many purposes, it is not much for metalearning. We cannot expect to obtain a general model for such a complex problem as algorithm recommendation using a limited number of datasets. Also, there is no guarantee that the datasets constitute a representative sample for each possible task we may encounter in future.

Dataset repositories are discussed in more detail in Chapter 16.

### 8.7.2 Generating synthetic datasets

The generation of synthetic datasets could be regarded as a natural way to extend the number of datasets needed in metalearning. In the proposal by Vanschoren and Blockeel (2006), new datasets are generated by varying a set of characteristics that describe the concepts to be represented in the data. The characteristics include the *concept model* and the *size* of the model. The datasets generated should have similar properties to natural (i.e., real-world) data.

Vanschoren and Blockeel (2006) propose the use of existing techniques for experimental design as an inspiration to guide dataset generation for metalearning studies. However, they recognize that building such a generator is a challenging task. Partial solutions have been proposed, in which the correlation between features and concepts is obtained by recursive partitioning on the space of features (Scott and Wilkins, 1999).

Given that it is difficult to make sure that the datasets generated are similar to natural ones, this approach is more suitable for understanding algorithm behavior than for the purpose of algorithm recommendation.

### 8.7.3 Generating variants of existing datasets

An alternative method to obtain more metadata is to generate new datasets by manipulating existing ones. This may be done either by changing the values of a particular feature, which may affect its distribution, or by changing the structure of the data (e.g., adding irrelevant or noisy features) (Aha, 1992; Hilario and Kalousis, 2000). Usually the changes are done separately on independent features and on a dependent one (i.e.,

the target feature) by, for instance, adding noise. Usually the changes are focused on a certain aspect of the behavior of the given algorithms. For instance, the addition of redundant features can be used to investigate the resilience of some algorithms to redundancy.

Soares (2009) proposed a method to obtain a larger number of datasets using a simple transformation of the existing datasets. The new datasets generated are referred to as *datasetoids*. The author tested the approach on the problem of using metalearning to predict when to prune decision trees. The results show a significant improvement when using datasetoids.

However, the increase in the number of datasets raises a problem, as it is necessary to estimate the performance of the algorithms on these datasets. Running all candidate algorithms on all new datasets can be computationally very expensive. Prudêncio et al. (2011) have proposed to use active learning, which in this context represents *active metalearning*. The authors have shown that it is possible to significantly reduce the computational cost not only without loss of metalearning accuracy but with potential gains.

### 8.7.4 Segmenting a large dataset or data stream

New datasets may be generated by segmenting a large dataset or data stream. In the area referred to as *extreme data mining* (Fogelman-Soulié, 2006), a large database is segmented into a number of subsets (e.g., by customer or product) and different models are generated for each subset.

In massive data streams, large volumes of data are continuously available. Some data streams include a concept shift, where some aspect of the data changes. Such a data stream can be used to generate different datasets corresponding to somewhat diverse portions of the data.

More details about algorithm recommendation for data streams can be found in Chapter 11.

### 8.7.5 Searching for datasets with discriminatory power

In this subsection we discuss two different approaches. The first one uses dataset characteristics and algorithm performance to obtain so-called 2D footprints. The authors show that many algorithms have overlapping footprints. The second approach uses pairs of rankings to characterize the diversity of both datasets and algorithms.

#### Using datasets characteristics and 2D footprints

In order to guarantee good performance of a metalearning system relying on a given portfolio, we need sufficiently diverse datasets that enable to discriminate well between different alternatives so as to provide the best possible recommendation for each case. This problem was addressed by Muñoz et al. (2018).

Their study included 235 datasets in total, most of which were from the UCI.<sup>2</sup> The methodology proposed relies on a good characterization of all datasets, so the authors have considered quite a large number of features (509) for this task. Their aim was to select a small subset that would characterize well the hardness of the classification task.

---

<sup>2</sup>The authors refer to the datasets as *instances*.

The details regarding the selection process are described in their paper. This way the authors identified just ten features is discussed in Section 4.7.

In the next step, the ten-dimensional space was projected onto a two-dimensional space, thus enabling to visualize classification datasets as points in a two-dimensional space. This reveals pockets of hard and easy datasets in an area in the 2D space, referred to as a *footprint*, where a particular algorithm is expected to do well. Quantitative metrics capturing, for instance, the area of the footprint provide objective measures of the relative advantage of an algorithm across the given set of datasets.

The results presented in this paper demonstrate the lack of diversity of the given test datasets, as most algorithms have similar footprints. This may be due to three possible reasons: (1) the algorithms are all essentially rather similar, (2) the datasets are not revealing the strengths and weaknesses of each algorithm as much as is desired, or (3) the features used may not be sufficiently discriminative.

The authors have proposed a method to generate new test datasets, aiming to enrich the diversity of datasets. The proposed method uses a Gaussian mixture model (GMM), which can be tuned. The sample from GMM is characterized by  $f_S$ . The method requires defining the target vector of features, which drives the tuning process. As the authors have shown, this process can lead to datasets covering well the 2D space. Future work should show that the richer metadata is useful and indeed facilitates the process of selecting algorithms for new datasets.

### Using correlation of rankings to characterize diversity

Abdulrahman et al. (2018) have proposed to characterize a given metalearning problem in the following way. They have observed that, if two datasets are very similar, the algorithm rankings will also be similar and, consequently, the pairwise correlation coefficient will be near 1. On the other hand, if the two datasets are quite different, the correlation will be low. In the extreme case, it will be  $-1$  if one ranking is the inverse of the other. So the distribution of pairwise correlation coefficients provides an estimate of how difficult the metalearning task is for a given portfolio of algorithms.

Figure 8.2 shows a histogram of correlation values, reproduced from Abdulrahman et al. (2018). The histogram can be characterized by its median value and appropriate percentiles (e.g., 25% and 75%). A metalearning task can be considered easy if the median value is high and if the inter-percentile range is small. Let us consider why this is so. If we select any dataset and consider it similar to the target dataset, many other datasets are similar to it. So we can reuse the metaknowledge acquired on these datasets.

However, we note that this method can be used only if the new target datasets are of the same kind as the datasets analyzed so far (i.e., those used to construct the distribution).

## 8.8 Complete versus Incomplete Metadata

In Chapters 1, 2, and 5 we have presented a basic scheme of metalearning, which involves the following steps: generation of metadata, generation of meta-model, and application of the meta-model to the target dataset. The first step involves running experiments of some algorithms (or workflows) on some of the available datasets. If we were to run all algorithms (or workflows) on all available datasets, we would obtain *complete results* and hence complete metadata.

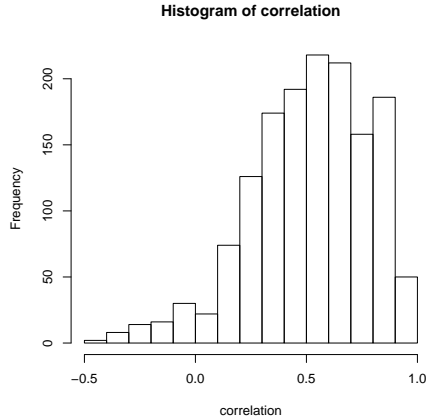


Fig. 8.2: Spearman's rank correlation coefficient between rankings for pairs of datasets

Incomplete results would arise if, for some pair (or pairs) of datasets and algorithms (or workflows), the performance result would not be available. The aim of this section is to address the following questions:

- Is it possible to obtain complete metadata?
- Is it necessary to have complete metadata?
- Does the order of the tests matter?
- How can we exploit the ideas from multi-armed bandits to schedule tests?
- Should we delegate the responsibility of gathering test results to the community?

The questions above are addressed in subsequent sections. The last one is discussed in a separate chapter (Chapter 16).

### 8.8.1 Is it possible to obtain complete metadata?

The answer to the above question is, in general, negative. A complete set of results could only be obtained in situations that involve a limited number of algorithms and datasets. Our answer is negative for various reasons that are explained in the following subsections.

#### There may be too many experiments to carry out

The number of experiments depends on the size of the search space. When dealing with continuous configuration spaces, there is an infinite number of configurations. As such, there is no way to enumerate all possible configurations or store them, unless the configuration space is discretized in some way. Even in the domain of discrete search spaces, if we were to deal with a modest number of 100 algorithms, each with 100



different hyperparameter settings, and conducted tests on 100 datasets, we would have to conduct  $100^3$  (i.e.,  $10^6$ ) tests.

This number would grow further if we were to consider different preprocessing operations. If we decided, for practical reasons, to conduct a subset of these tests, the metadata can be incomplete.

### Some experiments may result in failure

Some experiments result in failure or do not terminate in a given time budget. Failures do sometimes occur in the execution of base-level algorithms. In some cases, it is possible to recover from such failures (e.g., insufficient memory), but there are cases where the performance of an algorithm on a dataset cannot be obtained (e.g., software bug). If an algorithm fails to run on a dataset, its performance is not quantifiable, although it is not missing. One approach to deal with this issue is to penalize such algorithms in some way. The simplest strategy could be to use the appropriate default strategy, which is normally based on simple statistics of the data. In classification, this would be predicting the most frequent class, and in regression, it would be predicting the mean target value. The estimated performance of this default strategy would be used to replace the performance of the algorithms that fail.

### New dataset(s) have been introduced

It can be expected that extending our set up with a new dataset and the corresponding metadata could somewhat improve the ability of the system to provide good recommendations for the new problems. Therefore, it is important to carry out such extensions whenever new datasets become available. However, whenever new datasets have been introduced into our set up, the metadata becomes incomplete. This requires that all the available algorithms are run on the new dataset. This may be computationally rather expensive, particularly if the dataset is large and the number of alternatives to test is large too (the number of algorithms or workflows and its variants).

When a new base-level algorithm becomes available, it is necessary to update the metaknowledge so that the system could consider it in the recommendations provided. For that purpose, the metadata describing the performance of algorithms on known datasets (i.e., meta-examples) must be extended with information concerning the new algorithm. It is therefore necessary to run it on those datasets, which may require significant computational effort. One approach is to run all experiments off-line and update the metadata only after the results become available.

### Using estimates instead of real values

The process of obtaining test results is computationally expensive, and so different strategies exist to try to alleviate the problem. This can be done in two ways. Firstly, it can be used to generate *estimates of the performance* and use them as substitutes for the true performance until the true performance becomes available. Curiously enough, this idea was mentioned in the first edition of this book. A similar idea was followed in the area of hyperparameter optimization, where the estimates were provided by *surrogate models* (see Chapter 6 for details).

### 8.8.2 Is it necessary to have complete metadata?

To answer this question satisfactorily, it is necessary to consider different systems and determine what this system can (cannot) do when the metadata is incomplete. Not many systems are accompanied by such study, so it is difficult to provide an all-encompassing answer to this question here.

We will limit this exposition to one study carried out by Abdulrahman et al. (2018) which involves the average ranking method  $AR^*$ , discussed in Chapter 2. The authors have shown that the performance of this method was not affected even by 50% of omissions in the metadata. However, as was shown, it was necessary to modify the aggregation method so that it would cater for incomplete rankings.

### 8.8.3 Does the order of tests matter?

In Chapter 6, we discussed various approaches including random search on the one hand and various “more informed” methods, such as sequential model-based optimization (SMBO), on the other hand. Various authors have shown that the more informed methods achieve, in general, better results than uninformed ones (e.g., random search). This is done by reordering the tests on the basis of information that has been gathered in previous tests. Further details on this topic are given in the following section.

## 8.9 Exploiting Strategies from Multi-armed Bandits to Schedule Experiments

This section addresses some ideas for how to obtain performance results of algorithms on earlier datasets. As argued, there is a large number of possibilities, and smart planning might improve the quality of the metadata. The process of gathering test results can be compared to the process of gathering knowledge about different “arms” in multi-armed bandit (MAB) problems. The MAB problem includes a gambler whose aim is to decide which arm of a  $k$ -slot machine to pull to maximize his total reward in a series of trials (Katehakis and Veinott, 1987; Vermorel and Mohri, 2005). The aim is to find a good compromise between *exploration* (i.e., examining different arms) and *exploitation* (using the best arm(s) for the target problem).

Many real-world learning and optimization problems can be modeled using this paradigm, and algorithm selection/configuration is one of them. Different workflows (pipelines) configurations can be compared to different arms. In consequence, the solution to the MAB problem can inspire new effective solutions to the problem of algorithm selection/configuration.

### 8.9.1 Some concepts and strategies of MAB

One important notion in this area is the concept of *reward*, which is received after an arm has been pulled. The difference from the optimal is often referred to as the *regret* or *loss*. Typically, the aim is to maximize the accumulated reward, which is equivalent to minimizing the accumulated loss, as different arms have been pulled. Table 8.2 shows the correspondence of some terms used in the area of MAB and the terms used in the area of algorithm selection/configuration and metalearning. Some common MAB strategies are discussed in the following subsections.

Table 8.2: Correspondence of terms in MAB and this book

$N$ levers	$N$ algorithms
Pulling a lever	Evaluating an algorithm (configuration, workflow)
Reward	Performance (e.g., accuracy)
Regret	Loss
Accumulated regret	Area under the loss curve
Horizon	Time budget
Contextual MAB	Metalearning problem that exploits features

### Strategy $\epsilon$ -greedy

This strategy chooses a random lever with frequency  $\epsilon$ , where  $\epsilon \in (0, 1)$  is set by the user. For the remaining  $(1 - \epsilon)$  proportion of cases, the best arm is chosen.

### Strategy $\epsilon$ -first

This strategy can be seen as a variant of  $\epsilon$ -greedy. It carries out all the exploration at the beginning. For a given number  $\epsilon T \in N$  of rounds, the levers are randomly pulled during the  $T$  first rounds. This pure exploration phase is followed by an exploitation phase. That is, during the remaining  $(1 - \epsilon)T$  rounds, the lever with the highest estimated mean is selected.

### Strategy $\epsilon$ -decreasing

This variant is similar to  $\epsilon$ -greedy except that the value of  $\epsilon$  decreases as the experiment progresses, resulting in high exploration in the initial rounds, while exploitation is preferred later. This can be captured by  $\epsilon_t = \min(1, \epsilon_t/t)$ , where  $t$  is the index of the current round.

### Probability matching method (SoftMax)

The SoftMax strategy was discussed already by Luce (1959), but many variants of this method were described later. This strategy exploits a random choice from Gibbs distribution. The lever  $k$  is chosen with probability

$$p_k = e^{\hat{\mu}_k/\tau} / \sum_{i=1}^n e^{\hat{\mu}_i/\tau} \quad (8.4)$$

where  $\hat{\mu}_k$  is the estimated mean of the rewards brought by pulling lever  $k$  and  $\tau$  is a parameter called the *temperature*, which is set by the user.

This method belongs to the so-called *probability matching methods*, as the choice is made according to a probability distribution, reflecting how likely the choice is optimal. The SoftMax strategy is sometimes also called *Boltzmann exploration*. One variant of SoftMax is referred to as *decreasing SoftMax*. In this variant, the temperature decreases with the number of rounds played.

## Interval estimation and upper confidence bound (UCB) methods

The interval estimation method attributes to each lever an *optimistic reward estimate* within a certain confidence interval and then chooses the lever with the *highest optimistic mean* (Kaelbling, 1993). Unobserved or infrequently observed levers have an over-valued reward mean, which stimulates further exploration. The more frequently a lever is pulled, the closer its optimistic reward estimate will be to the true reward mean. The original approach of Kaelbling (1993) was applied to Boolean awards. Vermorel and Mohri (2005) have applied it to real values and assumed that rewards are normally distributed. The upper bound estimate is based on that assumption. Assuming that a lever is observed  $n$  times with  $\hat{\mu}$  as the empirical mean and  $\hat{\sigma}$  as the empirical standard deviation, the upper bound is defined by

$$u_\alpha = \hat{\mu} + \frac{\hat{\sigma}}{\sqrt{n}} c^{-1}(1 - \alpha), \quad (8.5)$$

where  $c$  is the cumulative normal distribution function.

The upper confidence bound algorithms (Agrawal, 1995; Auer et al., 2002; Lai and Robbins, 1985) work in a similar way. Specifically, in each trial  $t$ , these algorithms estimate both the mean payoff  $|\hat{\mu}_{t,a}|$  of each arm  $a$  as well as a corresponding confidence interval  $c_{t,a}$ , so that  $|\hat{\mu}_{t,a} - \mu_a| < c_{t,a}$  holds with high probability. They then select the arm that achieves the highest upper confidence bound (UCB), namely  $a_t = \operatorname{argmax}_a (|\hat{\mu}_{t,a} + c_{t,a}|)$ .

## Pricing strategies (POKER)

Pricing strategies establish a price for each lever. The approach of Vermorel and Mohri (2005), called *Price of Knowledge and Estimated Reward* (Poker), relies on three main ideas: pricing uncertainty capturing value of information, the lever distribution and horizon.

The idea behind pricing uncertainty is to assign a value to the knowledge gained while pulling a particular lever. The notion of *value of information* or *exploration bonuses* has been studied in various domains and in the bandit literature (Meuleau and Bourguine, 1999). The objective is to quantify the uncertainty in the same units as the rewards.

The second idea is that the properties of unobserved levers could potentially be estimated, to a certain extent, from the levers already observed. This is particularly useful when there are many more levers than rounds.

The third observation is that the strategy should explicitly take into account the horizon  $H$ , that is, the number of rounds that remain to be played. The amount of exploration clearly depends on  $H$ . If, for instance, the play is reaching the horizon ( $H=1$ ), it does not make sense to explore more, and the strategy should be to rely on pure exploitation, i.e., choosing the lever with the highest estimated reward. So the horizon value affects the estimate of the value of the knowledge that could be acquired.

## Contextual-bandit problem

Some researchers have introduced the so-called *contextual-bandit problem*, where different arms are characterized by features. These are represented in the form of a feature vector (context vector).

The agent uses these context vectors together with the rewards of the arms played in the past to make the choice of the arm to play in the current iteration. Over time, the learner's aim is to collect enough information about how the context vectors and rewards relate to each other, so that it can predict the next best arm to play by looking at the feature vectors (Langford and Zhang, 2007).

This approach has been applied, for instance, to personalized recommendation of news articles (Li et al., 2010). The learning algorithm sequentially selects articles for users based on contextual information about the users and articles, while simultaneously adapting its article selection strategy based on user-click feedback.

Contextual approaches can be compared to metalearning approaches that exploit dataset features and tests.

## 8.10 Discussion

In this chapter, we addressed two components from the framework of Rice (1976), i.e., the problem space and the algorithm space. The algorithm space is commonly known as the configuration space. In traditional metalearning systems, the configuration space is often a discrete set of algorithms or workflows, whereas in AutoML systems, often a continuous set is considered. While these representations are similar on some level, they also require different approaches and invoke different biases. Both of these representations have been studied extensively.

For discrete spaces, Abdulrahman et al. (2019) aimed to reduce the configuration space, so that the search for the best algorithm or workflow would converge faster. For continuous spaces, various authors have tried to determine which hyperparameters are generally important.

Recently, the neural architecture search community started to address the problem of search space construction (Yu et al., 2020; Yang et al., 2020). In order to keep this chapter focused towards traditional metalearning approaches, we did not go into details here, but the interested reader might find the references a good starting point.

Finally, this chapter looked at various aspects regarding the problem space, in particular, which datasets should be included in the metadata, how many experiments should be carried out, and in which order, and terminated with the discussion of the multi-armed bandit methods, which suggest some answers to the last issue.

## References

- Abdulrahman, S., Brazdil, P., van Rijn, J. N., and Vanschoren, J. (2018). Speeding up algorithm selection using average ranking and active testing by introducing runtime. *Machine Learning*, 107(1):79–108.
- Abdulrahman, S., Brazdil, P., Zainon, W., and Alhassan, A. (2019). Simplifying the algorithm selection using reduction of rankings of classification algorithms. In *ICSCA '19, Proceedings of the 2019 8th Int. Conf. on Software and Computer Applications, Malaysia*, pages 140–148. ACM, New York.
- Agrawal, R. (1995). Sample mean based index policies with  $O(\log n)$  regret for the multi-armed bandit problem. *Advances in Applied Probability*, 27(4):1054–1078.
- Aha, D. W. (1992). Generalizing from case studies: A case study. In Sleeman, D. and Edwards, P., editors, *Proceedings of the Ninth International Workshop on Machine Learning (ML92)*, pages 1–10. Morgan Kaufmann.

- Asuncion, A. and Newman, D. (2007). UCI machine learning repository.
- Auer, P., Cesa-Bianchi, N., and Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2-3):235–256.
- Biedenkapp, A., Lindauer, M., Eggenberger, K., Fawcett, C., Hoos, H., and Hutter, F. (2017). Efficient parameter importance analysis via ablation with surrogates. In *Thirty-First AAAI Conference on Artificial Intelligence*, pages 773–779.
- Bischi, B., Casalicchio, G., Feurer, M., Gijsbers, P., Hutter, F., Lang, M., Mantovani, R. G., van Rijn, J. N., and Vanschoren, J. (2021). OpenML benchmarking suites. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, NIPS’21.
- Brazdil, P. and Cachada, M. (2018). Simplifying the algorithm portfolios with a method based on envelopment curves (working notes).
- Brazdil, P., Soares, C., and Pereira, R. (2001). Reducing rankings of classifiers by eliminating redundant cases. In Brazdil, P. and Jorge, A., editors, *Proceedings of the 10th Portuguese Conference on Artificial Intelligence (EPIA2001)*. Springer.
- Breiman, L. (2001). Random forests. *Machine learning*, 45(1):5–32.
- Cohen, W. W. (1994). Grammatically biased learning: Learning logic programs using an explicit antecedent description language. *Artificial Intelligence*, 68(2):303–366.
- Davies, T. R. and Russell, S. J. (1987). A logical approach to reasoning by analogy. In McDermott, J. P., editor, *Proceedings of the 10th International Joint Conference on Artificial Intelligence, IJCAI 1987*, pages 264–270, Freiburg, Germany. Morgan Kaufmann.
- De Raedt, L. and Dehaspe, L. (1997). Clausal discovery. *Machine Learning*, 26:99–146.
- Demšar, J. (2006). Statistical comparisons of classifiers over multiple data sets. *The Journal of Machine Learning Research*, 7:1–30.
- Došilović, F., Brčić, M., and Hlupič, N. (2018). Explainable artificial intelligence: A survey. In *Proc. of the 41st Int. Convention on Information and Communication Technology, Electronics and Microelectronics MIPRO*.
- Fawcett, C. and Hoos, H. (2016). Analysing differences between algorithm configurations through ablation. *Journal of Heuristics*, 22(4):431–458.
- Feurer, M., Klein, A., Eggenberger, K., Springenberg, J., Blum, M., and Hutter, F. (2015). Efficient and robust automated machine learning. In Cortes, C., Lawrence, N., Lee, D., Sugiyama, M., and Garnett, R., editors, *Advances in Neural Information Processing Systems 28*, NIPS’15, pages 2962–2970. Curran Associates, Inc.
- Feurer, M., Klein, A., Eggenberger, K., Springenberg, J. T., Blum, M., and Hutter, F. (2019). Auto-sklearn: Efficient and robust automated machine learning. In Hutter, F., Kotthoff, L., and Vanschoren, J., editors, *Automated Machine Learning: Methods, Systems, Challenges*, pages 113–134. Springer.
- Fogelman-Soulié, F. (2006). Data mining in the real world: What do we need and what do we have? In Ghani, R. and Soares, C., editors, *Proceedings of the Workshop on Data Mining for Business Applications*, pages 44–48.
- Fréchette, A., Kotthoff, L., Rahwan, T., Hoos, H., Leyton-Brown, K., and Michalak, T. (2016). Using the Shapley value to analyze algorithm portfolios. In *30th AAAI Conference on Artificial Intelligence*.
- Gordon, D. and desJardins, M. (1995). Evaluation and selection of biases in machine learning. *Machine Learning*, 20(1/2):5–22.
- Hettich, S. and Bay, S. (1999). The UCI KDD archive. <http://kdd.ics.uci.edu>.
- Hilario, M. and Kalousis, A. (2000). Quantifying the resilience of inductive classification algorithms. In Zighed, D. A., Komorowski, J., and Zytrowski, J., editors, *Proceedings of the Fourth European Conference on Principles of Data Mining and Knowledge Discovery*, pages 106–115. Springer-Verlag.

- Hirsh, H. (1994). Generalizing version spaces. *Machine Learning*, 17(1):5–46.
- Hutter, F., Hoos, H., and Leyton-Brown, K. (2013). Identifying key algorithm parameters and instance features using forward selection. In *Proc. of International Conference on Learning and Intelligent Optimization*, pages 364–381.
- Hutter, F., Hoos, H., and Leyton-Brown, K. (2014). An efficient approach for assessing hyperparameter importance. In *Proceedings of the 31st International Conference on Machine Learning*, ICML'14, pages 754–762.
- Jorge, A. M. and Brazdil, P. (1996). Architecture for iterative learning of recursive definitions. In De Raedt, L., editor, *Advances in Inductive Logic Programming*, volume 32 of *Frontiers in Artificial Intelligence and applications*. IOS Press.
- Kaelbling, L. P. (1993). *Learning in Embedded Systems*. MIT Press.
- Katehakis, M. N. and Veinott, A. F. (1987). The multi-armed bandit problem: Decomposition and computation. *Mathematics of Operations Research*, 12(2):262–268.
- Keogh, E. and Folias, T. (2002). The UCR time series data mining archive. <http://www.cs.ucsb.edu/~eamonn/TSDMA/index.html>. Riverside CA. University of California – Computer Science & Engineering Department.
- Kramer, S. and Widmer, G. (2001). Inducing classification and regression trees in first order logic. In Džeroski, S. and Lavrač, N., editors, *Relational Data Mining*, pages 140–159. Springer.
- Lai, T. L. and Robbins, H. (1985). Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics*, 6(1):4–22.
- Langford, J. and Zhang, T. (2007). The epoch-greedy algorithm for contextual multi-armed bandits. In *Advances in Neural Information Processing Systems 20*, NIPS'07, page 817–824. Curran Associates, Inc.
- Lee, J. W. and Giraud-Carrier, C. (2011). A metric for unsupervised metalearning. *Intelligent Data Analysis*, 15(6):827–841.
- Li, L., Chu, W., and Schapire, R. E. (2010). A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the International Conference on World Wide Web (WWW)*.
- Luce, D. (1959). *Individual Choice Behavior*. Wiley.
- Meuleau, N. and Bourguine, P. (1999). Exploration of multi-state environments: Local measures and back-propagation of uncertainty. *Machine Learning*, 35(2):117–154.
- Miettinen, K. (1999). *Nonlinear Multiobjective Optimization*. Springer.
- Mitchell, T. (1977). *Version spaces: A candidate elimination approach to rule learning*. PhD thesis, Electrical Engineering Department, Stanford University.
- Mitchell, T. (1980). The need for biases in learning generalizations. Technical Report CBM-TR-117, Rutgers Computer Science Department.
- Mitchell, T. (1982). Generalization as Search. *Artificial Intelligence*, 18(2):203–226.
- Mitchell, T. (1990). The need for biases in learning generalizations. In Shavlik, J. and Dietterich, T., editors, *Readings in Machine Learning*. Morgan Kaufmann.
- Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill.
- Morik, K., Wrobel, S., Kietz, J., and Emde, W. (1993). *Knowledge Acquisition and Machine Learning: Theory, Methods and Applications*. Academic Press.
- Muñoz, M., Villanova, L., Baatar, D., and Smith-Miles, K. (2018). Instance Spaces for Machine Learning Classification. *Machine Learning*, 107(1).
- Peterson, A. H. and Martinez, T. (2005). Estimating the potential for combining learning models. In *Proc. of the ICML Workshop on Meta-Learning*, pages 68–75.
- Probst, P., Boulesteix, A.-L., and Bischl, B. (2019). Tunability: Importance of hyperparameters of machine learning algorithms. *Journal of Machine Learning Research*, 20(53):1–32.

- Prudêncio, R. B. C., Soares, C., and Ludermit, T. B. (2011). Combining meta-learning and active selection of datasetoids for algorithm selection. In Corchado, E., Kurzyński, M., and Woźniak, M., editors, *Hybrid Artificial Intelligent Systems. HAIS 2011.*, volume 6678 of *LNCS*, pages 164–171. Springer.
- Rice, J. R. (1976). The algorithm selection problem. *Advances in Computers*, 15:65–118.
- Russell, S. and Grosof, B. (1990a). Declarative bias: An overview. In Benjamin, P., editor, *Change of Representation and Inductive Bias*. Kluwer Academic Publishers.
- Russell, S. and Grosof, B. (1990b). A sketch of autonomous learning using declarative bias. In Brazdil, P. and Konolige, K., editors, *Machine Learning, Meta-Reasoning and Logics*. Kluwer Academic Publishers.
- Scott, P. D. and Wilkins, E. (1999). Evaluating data mining procedures: techniques for generating artificial data sets. *Information & Software Technology*, 41(9):579–587.
- Sharma, A., van Rijn, J. N., Hutter, F., and Müller, A. (2019). Hyperparameter importance for image classification by residual neural networks. In Kralj Novak, P., Šmuc, T., and Džeroski, S., editors, *Discovery Science*, pages 112–126. Springer International Publishing.
- Silverstein, G. and Pazzani, M. J. (1991). Relational clichés: Constraining induction during relational learning. In Birnbaum, L. and Collins, G., editors, *Proceedings of the Eighth International Workshop on Machine Learning (ML'91)*, pages 203–207, San Francisco, CA, USA. Morgan Kaufmann.
- Snoek, J., Swersky, K., Zemel, R., and Adams, R. (2014). Input warping for Bayesian optimization of non-stationary functions. In Xing, E. P. and Jebara, T., editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *ICML'14*, pages 1674–1682, Beijing, China. JMLR.org.
- Soares, C. (2009). UCI+ +: Improved support for algorithm selection using datasetoids. In *Proceedings of Pacific-Asia Conference on Knowledge Discovery and Data Mining*.
- Sobol, I. M. (1993). Sensitivity estimates for nonlinear mathematical models. *Mathematical Modelling and Computational Experiments*, 1(4):407–414.
- van Rijn, J. N. and Hutter, F. (2018). Hyperparameter importance across datasets. In *KDD '18: The 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM.
- Vanschoren, J. and Blockeel, H. (2006). Towards understanding learning behavior. In *Proceedings of the Fifteenth Annual Machine Learning Conference of Belgium and the Netherlands*.
- Vanschoren, J., van Rijn, J. N., Bischl, B., and Torgo, L. (2014). OpenML: networked science in machine learning. *ACM SIGKDD Explorations Newsletter*, 15(2):49–60.
- Vermorel, J. and Mohri, M. (2005). Multi-armed bandit algorithms and empirical evaluation. In *Machine Learning: ECML-94, European Conference on Machine Learning, LNAI 3720*. Springer.
- Xu, L., Hutter, F., Hoos, H., and Leyton-Brown, K. (2012). Evaluating component solver contributions to portfolio-based algorithm selectors. In Cimatti, A. and Sebastiani, R., editors, *Theory and Applications of Satisfiability Testing – SAT 2012*, pages 228–241. Springer Berlin Heidelberg.
- Yang, A., Esperança, P. M., and Carlucci, F. M. (2020). NAS evaluation is frustratingly hard. In *International Conference on Learning Representation, ICLR 2020*.
- Yu, K., Sciuto, C., Jaggi, M., Musat, C., and Salzmann, M. (2020). Evaluating the search phase of neural architecture search. In *International Conference on Learning Representation, ICLR 2020*.



---

# Combining Base-Learners into Ensembles

Christophe Giraud-Carrier

**Summary.** This chapter discusses ensembles of classification or regression models, because they represent an important area of machine learning. They have become popular as they tend to achieve high performance when compared with single models. Besides, they also play an essential role in data-streaming solutions. This chapter starts by introducing ensemble learning and presents an overview of some of its most well-known methods. These include bagging, boosting, stacking, cascade generalization, cascading, delegating, arbitrating and meta-decision trees.

## 9.1 Introduction

Model combination consists of creating a single learning system from a collection of learning algorithms. In some sense, model combination may be viewed as a variation on the theme of combining data mining operations discussed in Chapter 7. There are two basic approaches to model combination. The first one exploits variability in the application's data and combines multiple copies of a single learning algorithm applied to different subsets of that data. The second one exploits variability among learning algorithms and combines several learning algorithms applied to the same application's data.

The main motivation for combining models is to reduce the probability of misclassification based on any single induced model by increasing the system's area of expertise through combination. Indeed, one of the implicit assumptions of model selection in meta-learning is that there exists an optimal learning algorithm for each task. Although this clearly holds in the sense that, given a task  $\phi$  and a set of learning algorithms  $\{A_k\}$ , there is a learning algorithm  $A_\phi$  in  $\{A_k\}$  that performs better than all of the others on  $\phi$ , the actual performance of  $A_\phi$  may still be poor. In some cases, one may mitigate the risk of settling for a suboptimal learning algorithm by replacing single model selection with model combination.

Because it draws on information about base-level learning — in terms of either the characteristics of various subsets of data or the characteristics of various learning algorithms — model combination is often considered a form of metalearning. This chapter is dedicated to a brief overview of model combination. We limit our presentation to a description of each individual technique and leave it to the interested reader to follow the references and other relevant literature for discussions of comparative performance among them.

To help with understanding and to motivate the organization of this chapter, Table 9.1 summarizes, for each combination technique, the underlying philosophy, the type of base-level information used to drive the combination at the meta level (i.e., metadata), and the nature of the metaknowledge generated, whether explicitly or implicitly. Further details are in the corresponding sections.

Table 9.1: Model combination techniques summary

Technique	Philosophy	Metadata	Metaknowledge
Bagging	Variation in data		Implicit in voting scheme
Boosting		Errors (updated distribution)	Voting scheme's weights
Stacking	Variation among learners (multi-expert)	Class predictions or probabilities	Mapping from metadata to class predictions
Cascade generalization		Class probabilities and base-level attributes	Mapping from metadata to class predictions
Cascading	Variation among learners (multistage)	Confidence on predictions (updated distribution)	Implicit in selection scheme
Delegating		Confidence on predictions	Implicit in delegation scheme
Arbitrating	Variation among learners (refereed)	Correctness of class predictions, base-level attributes, and internal propositions	Mappings from metadata to correctness (one for each learner)
Meta-decision trees	Variation in data and among learners	Class distribution properties (from samples)	Mapping from metadata to best model

## 9.2 Bagging and Boosting

Perhaps the most well-known techniques for exploiting variation in data are bagging and boosting. Both bagging and boosting combine multiple models built from a single learning algorithm by systematically varying the training data.

### 9.2.1 Bagging

Bagging, which stands for bootstrap aggregating, is due to Breiman (1996). Given a learning algorithm  $A$  and a set of training data  $T$ , bagging first draws  $N$  samples  $S_1, \dots, S_N$ , with replacement, from  $T$ . It then applies  $A$  independently to each sample to induce  $N$  models  $h_1, \dots, h_N$ .<sup>1</sup> When classifying a new query instance  $q$ , the induced models are combined via a simple voting scheme, where the class assigned to the new instance is the class that is predicted most often among the  $N$  models, as illustrated in Figure 9.1. The bagging algorithm for classification is shown in Figure 9.2.

Bagging is easily extended to regression by replacing the voting scheme of line 5 of the algorithm by an average of the models' predictions:

$$\text{Value}(q) = \frac{\sum_{i=1}^N h_i(q)}{N}.$$

---

<sup>1</sup>To be consistent with the literature, note that we shall use the term *model* rather than *hypothesis* throughout this chapter. However, we shall retain our established mathematical notation and denote a model by  $h$ .

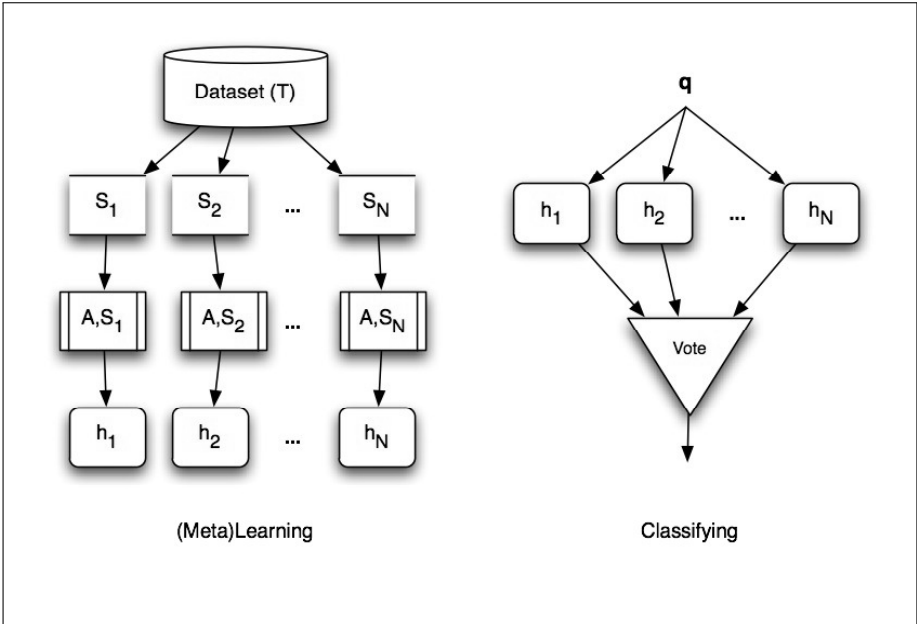


Fig. 9.1: Bagging

Algorithm Bagging( $T, A, N, d$ )

1. For  $k = 1$  to  $N$
2.  $S_k =$  random sample of size  $d$  drawn from  $T$ , with replacement
3.  $h_k =$  model induced by  $A$  from  $S_k$
4. For each new query instance  $q$
5.  $\text{Class}(q) = \text{argmax}_{y \in \mathcal{Y}} \sum_{k=1}^N \delta(y, h_k(q))$

where:

$T$  is the training set

$A$  is the chosen learning algorithm

$N$  is the number of samples or bags, each of size  $d$ , drawn from  $T$

$\mathcal{Y}$  is the finite set of target class values

$\delta$  is the generalized Kronecker function ( $\delta(a, b) = 1$  if  $a = b$ ; 0 otherwise)

Fig. 9.2: Bagging algorithm for classification

Bagging is most effective when the base-learner is *unstable*. A learner is unstable if it is highly sensitive to data, in the sense that small perturbations in the data cause large changes in the induced model. One simple example of instability is order dependence, where the order in which training instances are presented has a significant impact on the learner's output.

Bagging typically increases accuracy. However, if  $A$  produces interpretable models (e.g., decision trees, rules), that interpretability is lost when bagging is applied to  $A$ .

### 9.2.2 Boosting

Boosting is due to Schapire (1990). While bagging exploits data variation through a learner’s instability, boosting tends to exploit it through a learner’s *weakness*. A learner is weak if it generally induces models whose performance is only slightly better than random. Boosting is based on the observation that finding many rough rules of thumb (i.e., weak learning) can be a lot easier than finding a single, highly accurate prediction rule (i.e., strong learning). Boosting then assumes that a weak learner can be made strong by repeatedly running it on various distributions  $D_i$  over the training data  $T$  (i.e., varying the focus of the learner), and then combining the weak classifiers into a single composite classifier, as illustrated in Figure 9.3.

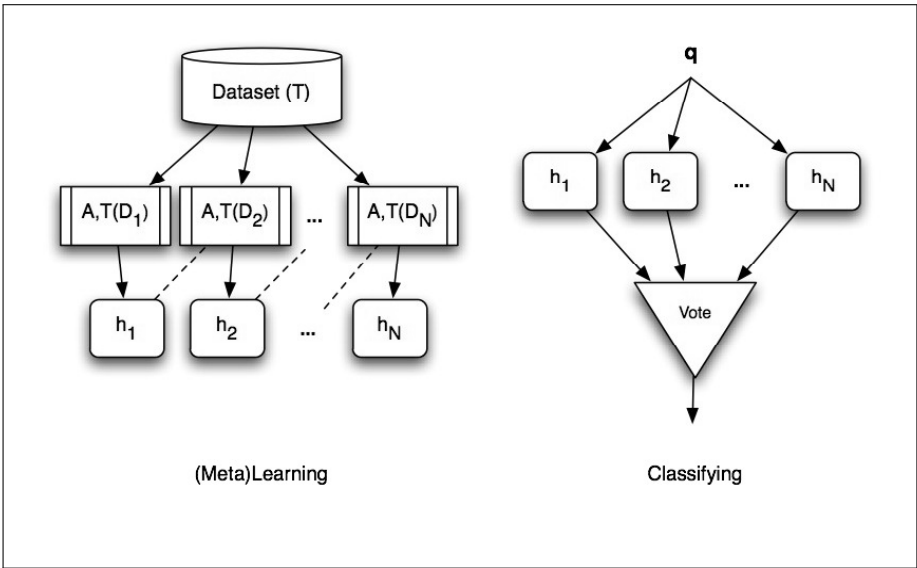


Fig. 9.3: Boosting

Unlike bagging, boosting tries actively to force the (weak) learning algorithm to change its induced model by changing the distribution over the training instances as a function of the errors made by previously generated models. The initial distribution  $D_1$  over the dataset  $T$  is uniform, with each instance assigned a constant weight, i.e., probability of being selected for training, of  $1/|T|$ , and a first model is induced. At each subsequent iteration, the weights of misclassified instances are increased, thus focusing the next model’s attention on them. This procedure goes on until either a fixed number of iterations has been performed or the total weight of the misclassified instances exceeds 0.5. The popular AdaBoost.M1 boosting algorithm for classification (Freund and Schapire, 1996b) is shown in Figure 9.4.

Algorithm AdaBoost.M1( $T, A, N$ )

1. For  $k = 1$  to  $|T|$
2.  $D_1(x_k) = \frac{1}{|T|}$
3. For  $i = 1$  to  $N$
4.  $h_i =$  model induced by  $A$  from  $T$  with distribution  $D_i$
5.  $\epsilon_i = \sum_{k: h_i(x_k) \neq y_k} D_i(x_k)$
6. If  $\epsilon_i > .5$
7.      $N = i - 1$
8.     Abort loop
9.      $\beta_i = \frac{\epsilon_i}{1 - \epsilon_i}$
10. For  $k = 1$  to  $|T|$
11.      $D_{i+1}(x_k) = \frac{D_i(x_k)}{Z_i} \times \begin{cases} \beta_i & \text{if } h_i(x_k) = y_k \\ 1 & \text{otherwise} \end{cases}$
12. For each new query instance  $q$
13.     Class( $q$ ) =  $\operatorname{argmax}_{y \in \mathcal{Y}} \sum_{i: h_i(q) = y} \log \frac{1}{\beta_i}$

where:

$T$  is the training set

$A$  is the chosen learning algorithm

$N$  is the number of iterations to perform over  $T$

$\mathcal{Y}$  is the finite set of target class values

$Z_i$  is a normalization constant, chosen so that  $D_{i+1}$  is a distribution

Fig. 9.4: Boosting algorithm for classification (AdaBoost.M1)

The class of a new query instance  $q$  is given by a weighted vote of the induced models. The case of regression is more complex. The regression version of AdaBoost, known as AdaBoost.R, is based on decomposition into infinitely many classes. The reader is referred to the paper of Freund and Schapire (1996a) for details.

Although the argument for boosting originated with weak learners, boosting may actually be successfully applied to any learner.

## 9.3 Stacking and Cascade Generalization

While bagging and boosting exploit variation in the data, stacking and cascade generalization exploit differences among learners. They make explicit two levels of learning: the base level where learners are applied to the task at hand, and the meta level where a new learner is applied to data obtained from learning at the base level.

### 9.3.1 Stacking

The idea of stacked generalization is due to Wolpert (1992). Stacking takes a number of learning algorithms  $\{A_1, \dots, A_N\}$  and runs them against the dataset  $T$  under consideration (i.e., base-level data) to produce a series of models  $\{h_1, \dots, h_N\}$ . Then, a

new dataset  $\mathcal{T}$  is constructed by replacing the description of each instance in the base-level dataset by the predictions of each base-level model for that instance.<sup>2</sup> This new metadataset is in turn presented to a new learner  $A_{meta}$  that builds a metamodel  $h_{meta}$  mapping the predictions of the base-level learners to target classes, as illustrated in Figure 9.5. The stacking algorithm for classification is shown in Figure 9.6.

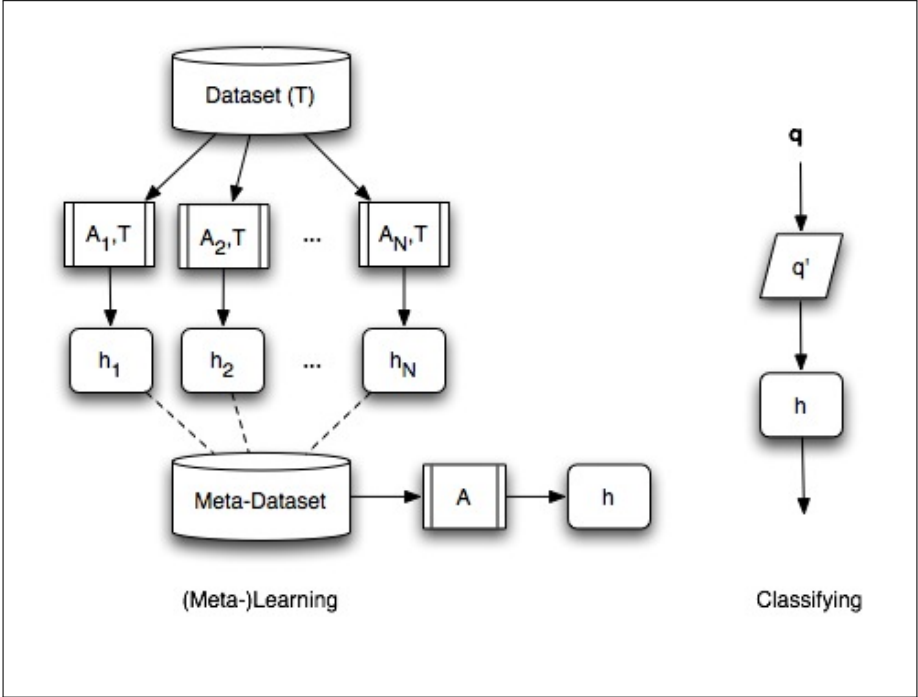


Fig. 9.5: Stacking

A new query instance  $q$  is first run through all the base-level learners to compose the corresponding query meta-instance  $q'$ , which serves as input to the metamodel to produce the final classification for  $q$ .

Note that the base-level models' predictions in line 5 (Figure 9.6) are obtained by running each instance through the models induced from the base-level dataset (lines 1 and 2). Alternatively, more statistically reliable predictions could be obtained through cross-validation as proposed in Efron (1983). In this case, lines 1 through 6 are replaced with the following:

1. For  $i = 1$  to  $N$
2. For  $k = 1$  to  $|T|$

<sup>2</sup>In some versions of stacking, the base-level description is not replaced by the predictions, but rather the predictions are appended to the base-level description, resulting in a kind of hybrid meta-example.

Algorithm Stacking( $T, \{A_1, \dots, A_N\}, A_{meta}$ )

1. For  $i = 1$  to  $N$
2.  $h_i =$  model induced by  $A_i$  from  $T$
3.  $\mathcal{T} = \emptyset$
4. For  $k = 1$  to  $|T|$
5.  $E_k = \langle h_1(x_k), h_2(x_k), \dots, h_N(x_k), y_k \rangle$
6.  $\mathcal{T} = \mathcal{T} \cup \{E_k\}$
7.  $h_{meta} =$  model induced by  $A_{meta}$  from  $\mathcal{T}$
8. For each new query instance  $q$
9.  $\text{Class}(q) = h_{meta}(\langle h_1(q), h_2(q), \dots, h_N(q) \rangle)$

where:

$T$  is the base-level training set  
 $N$  is the number of base-level learning algorithms  
 $\{A_1, \dots, A_N\}$  is the set of base-level learning algorithms  
 $A_{meta}$  is the chosen meta-level learner

Fig. 9.6: Stacking algorithm

3.  $E_k[i] = h_i(x_k)$  obtained by cross-validation
4.  $\mathcal{T} = \emptyset$
5. For  $k = 1$  to  $|T|$
6.  $\mathcal{T} = \mathcal{T} \cup \{E_k\}$

A variation on stacking is proposed in Ting and Witten (1997), where the predictions of the base-level classifiers in the metadataset are replaced by class probabilities. A meta-level example thus consists of a set of  $N$  (the number of base-level learning algorithms) vectors of  $m = |\mathcal{Y}|$  (the number of classes) coordinates, where  $p_{ij}$  is the posterior probability, as given by learning algorithm  $A_i$ , that the corresponding base-level example belongs to class  $j$ . Other forms of stacking, based on using partitioned data rather than full datasets, or using the same learning algorithm on multiple, independent data batches, have also been proposed (e.g., see Chan and Stolfo (1997); Ting and Low (1997)).

The transformation applied to the base-level dataset, whether through the addition of predictions or class probabilities, is intended to give information about the behavior of the various base-level learners on each instance, and thus constitutes a form of metaknowledge.

### 9.3.2 Cascade generalization

Gama and Brazdil (2000) proposed another model combination technique known as cascade generalization that also exploits differences among learners. In cascade generalization, the classifiers are used in sequence rather than in parallel as in stacking. Instead of the data from the base-level learners feeding into a single meta-level learner, each base-level learner  $A_{i+1}$  (except for the first one, i.e.,  $i > 0$ ) also acts as a kind of

meta-level learner for the base-level learner  $A_i$  that precedes it. Indeed, the inputs to  $A_{i+1}$  consist of the inputs to  $A_i$  together with the class probabilities produced by  $h_i$ , the model induced by  $A_i$ . A single learner is used at each step, and there is, in principle, no limit on the number of steps, as illustrated in Figure 9.7. The basic cascade generalization algorithm for two steps is shown in Figure 9.8.

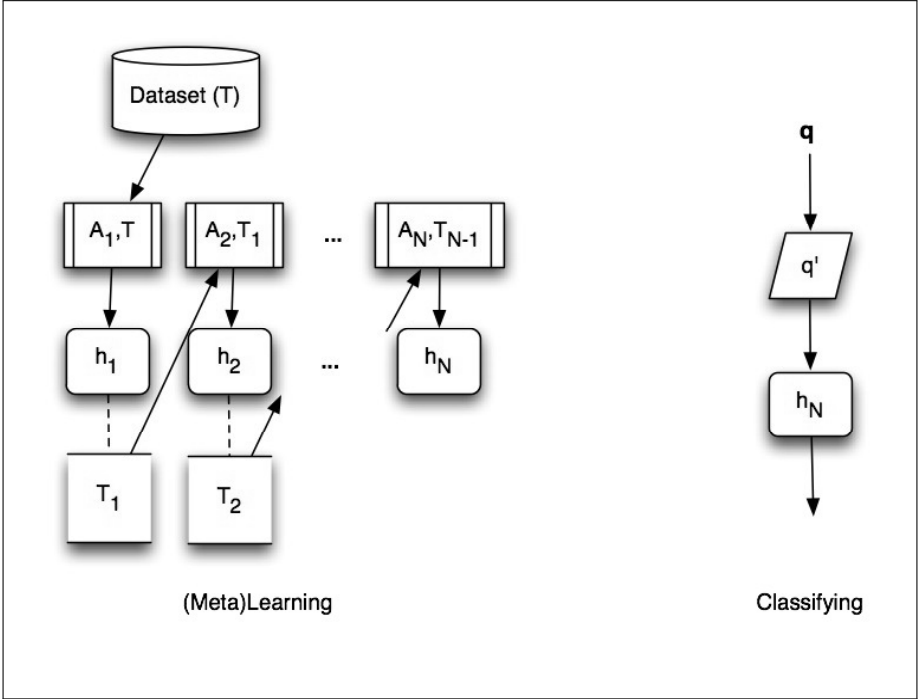


Fig. 9.7: Cascade generalization

This two-step algorithm is easily extended to an arbitrary number of steps — defined by the number of available classifiers — through successive invocation of the Extend-Dataset function, as illustrated in Figure 9.9, where the recursive algorithm begins with  $i = 1$ .<sup>3</sup>

A new query instance  $q$  is first extended into a meta-instance  $q'$  as it gathers metadata through the steps of the cascade. The final classification is then given by the output of the last model in the cascade on  $q'$ .

<sup>3</sup>To use this  $N$ -step version of cascade generalization for classification, it may be advantageous to implement it iteratively rather than recursively, so that intermediate models may be stored and used when extending new queries.



Algorithm CascadeGeneralization( $\{A_1, A_2\}, T$ )

1.  $h_1 =$  model induced by  $A_1$  from  $T$
2.  $T_1 = \text{ExtendDataset}(h_1, T)$
3.  $h_2 =$  model induced by  $A_2$  from  $T_1$
4. For each new query instance  $q$
5.  $q' = \text{ExtendDataset}(h_1, \{q\})$
6.  $\text{Class}(q) = h_2(q')$

where:

- $T$  is the original base-level training set
- $A_1$  and  $A_2$  are base level learning algorithms

Algorithm ExtendDataset( $h, T$ )

1.  $newT = \emptyset$
2. For each  $e = (\mathbf{x}, y) \in T$
3. For  $j = 1$  to  $|\mathcal{Y}|$
4.  $p_j =$  probability that  $e$  belongs to  $y_j$  according to  $h$
5.  $e' = (\mathbf{x}, p_1, \dots, p_{|\mathcal{Y}|}, y)$
6.  $newT = newT \cup \{e'\}$
7. Return  $newT$

where:

- $h$  is a model induced by a learning algorithm
- $T$  is the dataset to be extended with data generated from  $h$
- $\mathcal{Y}$  is the finite set of target class values

Fig. 9.8: Cascade generalization algorithm (two steps)

## 9.4 Cascading and Delegating

Like stacking and cascade generalization, cascading and delegating exploit differences among learners. However, whereas the former produce multi-expert classifiers (all constituent base classifiers are used for classification), the latter produce multistage classifiers in which not all base classifiers need be consulted when predicting the class of a new query instance. Hence, classification time is reduced.

### 9.4.1 Cascading

Alpaydin and Kaynak (1998) and Kaynak and Alpaydin (2000) developed the idea of cascading, which may be viewed as a kind of multilearner version of boosting. Like boosting, cascading varies the distribution over the training instances, here as a function of the confidence of the previously generated models.<sup>4</sup> Unlike boosting, however, cascad-

<sup>4</sup>This is a generalization of boosting's function of the errors of the previously generated models. Rather than biasing the distribution to only those instances the previous

Algorithm CascadeGeneralizationN( $\{A_1, \dots, A_N\}, T, i$ )

1.  $h =$  model induced by  $A_i$  from  $T$
2. If ( $i == N$ )
3.     Return  $h$
4.  $T' =$  ExtendDataset( $h, T$ )
5. CascadeGeneralizationN( $\{A_1, \dots, A_N\}, T', i + 1$ )

where:

$T$  is the original base-level training set

$N$  is the number of steps in the cascade

$\{A_1, \dots, A_N\}$  is the set of base-level learning algorithms

Fig. 9.9: Cascade generalization for arbitrary number of steps

ing does not strengthen a single learner but uses a small number of different classifiers of increasing complexity, in a cascade-like fashion, as shown in Figure 9.10.

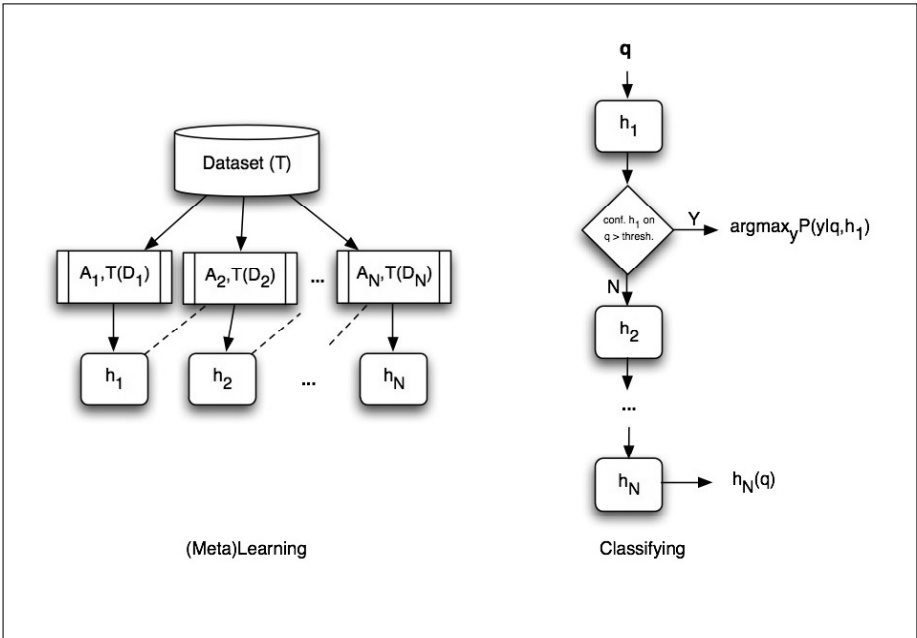


Fig. 9.10: Cascading

layers *misclassify*, cascading biases the distribution to those instances the previous layers are *uncertain* about.

Algorithm Cascading( $T, \{A_1, \dots, A_N\}$ )

1. For  $k = 1$  to  $|T|$
2.  $D_1(x_k) = \frac{1}{|T|}$
3. For  $i = 1$  to  $N - 1$
4.  $h_i$  = model induced by  $A_i$  from  $T$  with distribution  $D_i$
5. For  $k = 1$  to  $|T|$
6.  $D_{i+1}(x_k) = \frac{1 - \delta_i(x_k)}{\sum_{m=1}^{|T|} 1 - \delta_i(x_m)}$
7.  $h_N = k$ -NN
8. For each new query instance  $q$
9.  $i = 1$
10. While  $i < N$  and  $\delta_i(q) < \Theta_i$
11.  $i = i + 1$
12. If  $i = N$  Then
13. Class( $q$ ) =  $h_N(q)$
14. Else
15. Class( $q$ ) =  $\operatorname{argmax}_{y \in \mathcal{Y}} P(y|q, h_i)$

where:

$T$  is the base-level training set

$N$  is the number of base-level learning algorithms

$A_1, \dots, A_N$  are the base-level learning algorithms

$\Theta_i$  is the confidence threshold associated with  $A_i$ , s.t.  $\Theta_{i+1} \geq \Theta_i$

$\mathcal{Y}$  is the finite set of target class values

$\delta_i(x) = \max_{y \in \mathcal{Y}} P(y|x, h_i)$  is the confidence function for model  $h_i$

Fig. 9.11: Cascading algorithm

The initial distribution  $D_1$  over the dataset  $T$  is uniform, with each training instance assigned a constant weight of  $1/|T|$ , and a model  $h_1$  is induced with the first base-level learning algorithm  $A_1$ . Then, each base-level learner  $A_{i+1}$  is trained from the same dataset  $T$ , but with a new distribution  $D_{i+1}$ , determined by the confidence of the base-level learner  $A_i$  that precedes it. The confidence of the model  $h_i$ , induced by  $A_i$ , on a training instance  $x$  is defined as  $\delta_i(x) = \max_{y \in \mathcal{Y}} P(y|x, h_i)$ . At step  $i + 1$ , the weights of instances whose classification is uncertain under  $h_i$  (i.e., below a predefined confidence threshold) are increased, thus making them more likely to be sampled when training  $A_{i+1}$ . Early classifiers are generally semi-parametric (e.g., multilayer perceptrons) and the final classifier is always non-parametric (e.g.,  $k$ -nearest-neighbor). Thus, a cascading system can be viewed as creating rules, which account for most instances, in the early steps, and catching exceptions at the final step. The generic cascading algorithm is shown in Figure 9.11.

When classifying a new query instance  $q$ , the system sends  $q$  to all of the models and looks for the first model,  $h_k$ , from 1 to  $N$ , whose confidence on  $q$  is above the confidence threshold. If  $h_k$  is an intermediate model in the cascade, the class of the new query instance is the class with highest probability (line 15, Figure 9.11). If  $h_k$  is the final

(non-parametric) model in the cascade, the class of the new query instance is the output of  $h_k(q)$  (line 13, Figure 9.11).

Although the weighted iterative approach is similar, cascading differs from boosting in several significant ways. First, cascading uses different learning algorithms at each step, thus increasing the variety of the ensemble. Second, the final  $k$ -NN step can be used to place a limit on the number of steps in the cascade, so that a small number of classifiers is used to reduce complexity. Finally, when classifying a new instance, there is no vote across the induced models; only one model is used to make the prediction.

### 9.4.2 Delegating

A cautious, delegating classifier is a classifier that provides classifications only for instances above a predefined confidence threshold, and passes (or delegates) other instances to another classifier. The idea of delegating classifiers comes from Ferri et al. (2004). It is similar in spirit to cascading. In cascading, however, all instances are (re)weighted and processed at each step. In delegating, the next classifier is specialized to those instances for which the previous one lacks confidence, through training *only* on the delegated instances, as illustrated in Figure 9.12. The delegation stops either when there are no instances left to delegate or when a predefined number of delegation steps has been performed. The delegating algorithm is shown in Figure 9.13.

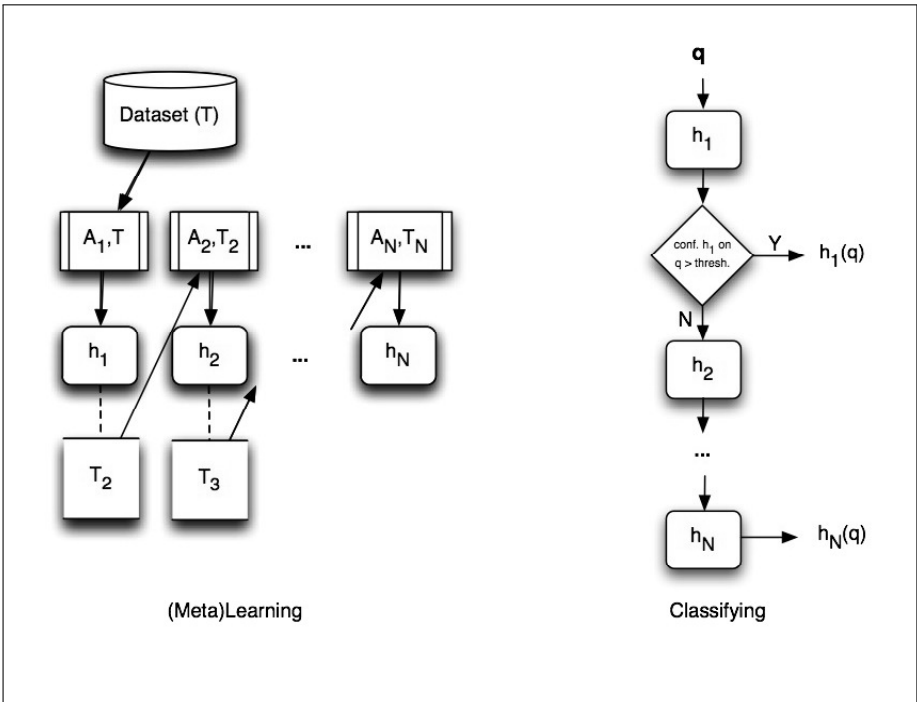


Fig. 9.12: Delegating

Algorithm Delegating( $T, \{A_1, \dots, A_N\}, N, Rel$ )

1.  $T_1 = T$
2.  $i = 0$
3. Repeat
4.    $i = i + 1$
5.    $h_i =$  model induced by  $A_i$  from  $T_i$
6.   If ( $Rel = \text{True}$  and  $i > 1$ ) Then
7.      $\tau_i = \text{getThreshold}(h_i, T_{i-1})$
8.   Else
9.      $\tau_i = \text{getThreshold}(h_i, T)$
10.    $T_{h_i}^> = \{e \in T_i : h_i^{CONF}(e) > \tau_i\}$
11.    $T_{h_i}^{\leq} = \{e \in T_i : h_i^{CONF}(e) \leq \tau_i\}$
12.    $T_{i+1} = T_{h_i}^{\leq}$
13. Until  $T_{h_i}^> = \emptyset$  or  $i > N$
14. For each new query instance  $q$
15.    $m = \min_k \{h_k(q) \geq \tau_k\}$
16.   Class( $q$ ) =  $h_m(q)$

where:

$T$  is the base-level training set

$N$  is the maximum number of delegating stages

$A_1, \dots, A_N$  are the base-level learning algorithms

$h_i^{CONF}(e)$  is the confidence of the prediction of model  $h_i$  for example  $e$

$Rel$  is a Boolean flag (true if  $\tau_i$  is to be computed relative to delegated examples)

$\text{getThreshold}(h, T)$  returns a confidence threshold for classifier  $h$  relative to  $T$

Fig. 9.13: Delegating algorithm

The function  $\text{getThreshold}(h, T)$  may be implemented in two different ways as follows:

- *Global percentage.*  $\tau = \max\{t : |\{e \in T : h^{CONF}(e) > t\}| \geq \rho \cdot |T|\}$ , where  $\rho$  is a user-defined fraction.
- *Stratified percentage.* For each class  $c$ ,  $\tau^c = \max\{t : |\{e \in T_c : h^{PROB^c}(e) > t\}| \geq \rho \cdot |T_c|\}$ , where  $h^{PROB^c}(e)$  is the probability of class  $c$  under model  $h$  for example  $e$ , and  $T_c$  is the set of examples of class  $c$  in  $T$ .

Note that there are actually four ways to compute the threshold, based on the value of the parameter  $Rel$ . When  $Rel$  is true (i.e., each threshold is computed relative to the examples delegated by the previous classifier), the approaches are called global relative percentage and stratified relative percentage, respectively; and when  $Rel$  is false, they are called global absolute percentage and stratified absolute percentage, respectively.

When classifying a new query instance  $q$ , the system first sends  $q$  to  $h_1$  and produces an output for  $q$  based on one of several delegation mechanisms, generally taken from the following alternatives:

- Round-rebound (only applicable to two-stage delegation):  $h_1$  defers to  $h_2$  when its confidence is too low, but  $h_2$  rebounds to  $h_1$  when its own confidence is also too low.
- Iterative delegation:  $h_1$  defers to  $h_2$ , which in turn defers to  $h_3$ , which in turn defers to  $h_4$ , and so on until a model  $h_k$  is found whose confidence on  $q$  is above threshold or  $h_N$  is reached. The algorithm of Figure 9.13 implements this mechanism (lines 14 to 16).

Delegation may be viewed as a generalization of divide-and-conquer methods (e.g., see Frank and Witten (1998); Fürnkranz (1999)), with a number of advantages including:

- Improved efficiency: each classifier learns from a decreasing number of examples,
- No loss of comprehensibility: there is no combination of models; each instance is classified by a single classifier, and
- Possibility to simplify the overall multi-classifier: see, for example, the notion of grafting for decision trees (Webb, 1997).

## 9.5 Arbitrating

A mechanism for combining classifiers by way of arbitration, originally introduced as *model applicability induction*, has been proposed by Ortega (1996); Ortega et al. (2001).<sup>5</sup> As with delegating, the basic intuition behind arbitrating is that various classifiers have different areas of expertise (i.e., portions of the input space on which they perform well). However, unlike in delegating, where successive classifiers are specialized to instances for which previous classifiers lack confidence, all classifiers in arbitrating are trained on the full dataset  $T$  and specialization is performed at run time when a query instance is presented to the system. At that time, the classifier whose confidence is highest in the area of input space close to the query instance is selected to produce the classification. The process is illustrated in Figure 9.14.

The area of expertise of each classifier is learned by its corresponding referee. The referee, although it can be any learned model, is typically a decision tree which predicts whether the associated classifier is correct or incorrect on some subset of the data, and with what reliability. The features used in building the referee decision tree consist of at least the primitive attributes that define the base-level dataset, possibly augmented by computed features (e.g., activation values of internal nodes in a neural network, conditions at various nodes in a decision tree) known as internal propositions, which assist in diagnosing examples for which the base-level classifier is unreliable (see Ortega et al. (2001) for details). The basic idea is that a referee holds meta-information on the area of expertise of its associated classifier and can thus tell when that classifier reliably predicts the outcome. Several classifiers are then combined through an arbitration mechanism, in which the final prediction is that of the classifier whose referee is the most reliably correct. The arbitrating algorithm is shown in Figure 9.15.

Interestingly, the neural network community has also proposed techniques that employ referee functions to arbitrate among the predictions generated by several classifiers. These are generally known as mixture of experts (e.g., see Jacobs et al. (1991); Jordan and Jacobs (1994); Waterhouse and Robinson (1994)).

<sup>5</sup>Interestingly, two other sets of researchers developed very similar arbitration mechanisms independently. See Koppel and Engelson (1997) and Tsybmal et al. (1998).

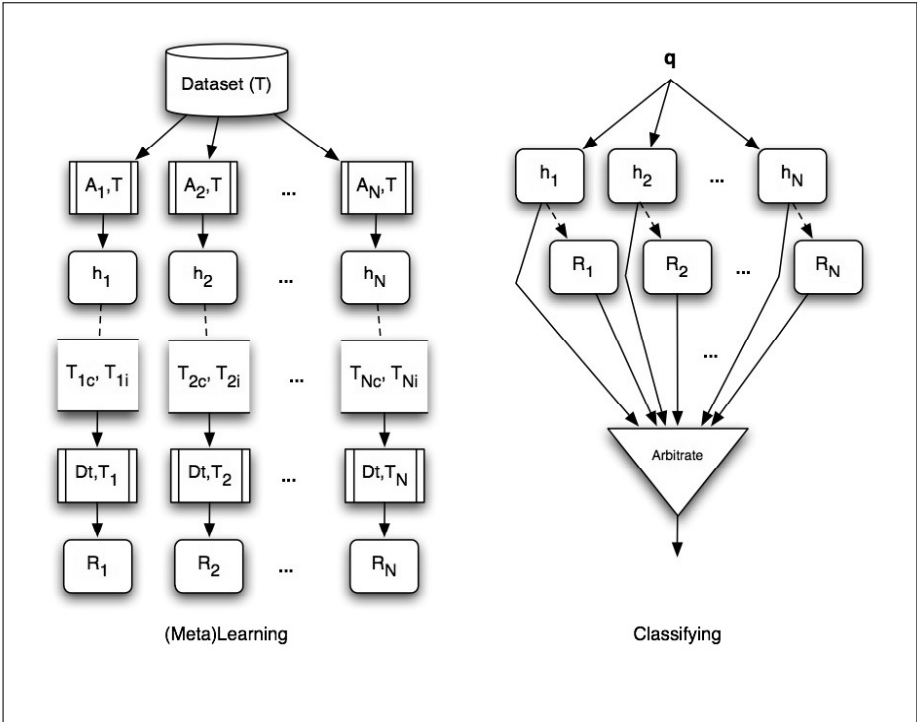


Fig. 9.14: Arbitrating

Finally, note that a different approach to arbitration was proposed by Chan and Stolfo (1993, 1997), where there is generally a unique arbiter for the entire set of  $N$  base-level classifiers. The arbiter is just another classifier learned by some learning algorithm on training examples that cannot be reliably predicted by the set of base-level classifiers. A typical rule for selecting training examples for the arbiter is as follows: select example  $e$  if none of the target classes gather a majority vote (i.e.,  $> N/2$  votes) for  $e$ . The final prediction for a query example is then generally given by a plurality of votes on the predictions of the base-level classifiers and the arbiter, with ties being broken by the arbiter. An extension involving the notion of an arbiter tree is also discussed, where several arbiters are built recursively in a tree-like structure. In this case, when a query example is presented, its prediction propagates upward in the tree from the leaves (base learners) to the root, with arbitration taking place at each level along the way.

### 9.6 Meta-decision Trees

Another approach to combining inductive models is found in the work of Todorovski and Džeroski (2003) on meta-decision trees (MDTs). The general idea in MDT is similar to stacking in that a metamodel is induced from information obtained using the results of base-level learning, as shown in Figure 9.16. However, MDTs differ from stacking in the choice of what information to use, as well as in the metalearning task. In particular,

Algorithm Arbitrating( $T, \{A_1, \dots, A_N\}$ )

1. For  $i = 1$  to  $N$
2.  $h_i =$  model induced by  $A_i$  from  $T$
3.  $R_i = \text{LearnReferee}(h_i, T)$
4. For each new query instance  $q$
5. For  $i = 1$  to  $N$
6.  $c_i =$  correctness of  $h_i$  on  $q$  as per  $R_i$
7.  $r_i =$  reliability of  $h_i$  on  $q$  as per  $R_i$
8.  $h^* = \text{argmax}_{h_i: c_i \text{ is "correct"} r_i}$
9.  $\text{Class}(q) = h^*(q)$

where:

$T$  is the base-level training set

$N$  is the number of base-level learning algorithms

$A_1, \dots, A_N$  are the base-level learning algorithms

$\text{LearnReferee}(A, T)$  returns a referee for learner  $A$  and dataset  $T$

Function  $\text{LearnReferee}(h, T)$

1.  $T_c =$  examples in  $T$  correctly classified by  $h$
2.  $T_i =$  examples in  $T$  incorrectly classified by  $h$
3. Select a set of features, including the attributes defining the examples and class, as well as additional features
4.  $Dt =$  pruned decision tree induced from  $T$
5. For each leaf  $L$  in  $Dt$
6.  $N_c(L) =$  number of examples in  $T_c$  classified to  $L$
7.  $N_i(L) =$  number of examples in  $T_i$  classified to  $L$
8.  $r = \frac{\max(|N_c(L), N_i(L)|)}{|N_c(L)| + |N_i(L)| + \frac{1}{2}}$
9. If  $|N_c(L)| > |N_i(L)|$  Then
10.  $L$ 's correctness is "correct"
11. Else
12.  $L$ 's correctness is "incorrect"
13. Return  $Dt$

Fig. 9.15: Arbitrating algorithm

MDTs build decision trees where each leaf node corresponds to a classifier rather than a classification. Hence, given a new query example, a meta-decision tree indicates the classifier that appears most suitable for predicting the example's class label. The MDT building algorithm is shown in Figure 9.17.

Class distribution properties are extracted from examples using the base-level learners on different subsets of the data (lines 7 to 9, Figure 9.17). These properties, in turn, become the attributes of the metalearning task. Unlike metalearning for algorithm selection, where these attributes are extracted from complete datasets (and thus there is one meta-example per dataset), MDTs have one meta-example per base-level example, simply substituting the base-level attributes with the new computed properties. The



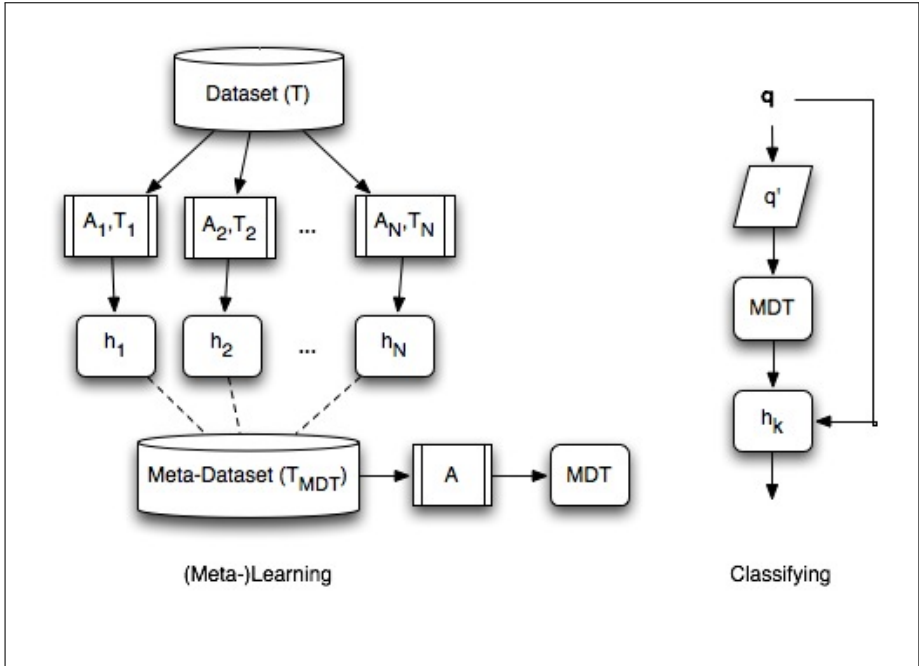


Fig. 9.16: Meta-decision tree

metamodel  $MDT$  is induced from these meta-examples,  $T_{MDT}$ , with a metalearning algorithm  $\mathcal{A}$ . Typically,  $\mathcal{A}$  is MLC4.5, an extension of the well-known C4.5 decision tree learning algorithm (Quinlan, 1993).

Interestingly, in addition to improving accuracy, MDTs, being comprehensible, also provide some insight about base-level learning. In some sense, each leaf of the MDT captures the relative area of expertise of one of the base-level learners (e.g., C4.5, LTree, CN2,  $k$ -NN, and naïve Bayes).

### 9.7 Discussion

The list of methods presented in this chapter is not intended to be exhaustive. Methods included have been selected because they represent classes of model combination approaches and are most closely connected to the subject of metalearning. A number of so-called *ensemble* methods have been proposed that combine many algorithms into a single learning system (e.g., see Kittler et al. (1998); Opitz and Maclin (1999); Caruana et al. (2004); Brown (2005)). The interested reader is referred to the literature for descriptions and evaluations of other combination and ensemble methods.

Because it uses results at the base level to construct a classifier at the meta level, model combination may clearly be regarded as a form of metalearning. However, its motivation is generally rather different from that of traditional metalearning. Whereas metalearning explicitly attempts to derive knowledge about the learning process itself, model combination focuses almost exclusively on improving base-level accuracy. Although they

```

Algorithm MDTBuilding( $T, \{A_1, \dots, A_N\}, m$ )
1.  $\{T_1, \dots, T_m\} = \text{StratifiedPartition}(T, m)$ 
2.  $T_{MDT} = \emptyset$ 
3. For  $i = 1$  to  $m$ 
4.   For  $j = 1$  to  $N$ 
5.      $h_j = \text{model induced by } A_j \text{ from } T - T_i$ 
6.     For each  $x \in T_i$ 
7.        $\text{maxprob}(x) = \max_{y \in \mathcal{Y}} P_{h_j}(y|x)$ 
8.        $\text{entropy}(x) = - \sum_{y \in \mathcal{Y}} P_{h_j}(y|x) \log P_{h_j}(y|x)$ 
9.        $\text{weight}(x) = \text{fraction of training examples used by } h_j \text{ to}$ 
           estimate the class distribution of  $x$ 
10.       $E_j(x) = \langle \text{maxprob}(x), \text{entropy}(x), \text{weight}(x) \rangle$ 
11.       $E_j = \cup_{x \in T_i} E_j(x)$ 
12.       $T_{MDT} = T_{MDT} \cup \text{join}_{j=1}^N E_j$ 
13.       $MDT = \text{model induced by MLC4.5 from } T_{MDT}$ 
14. Return  $MDT$ 
15. For each new query instance  $q$ 
16.    $\text{Class}(q) = MDT(\langle E_1(q), E_2(q), \dots, E_N(q) \rangle)$ 

```

where:

$T$  is the base-level training set

$N$  is the number of base-level learning algorithms

$A_1, \dots, A_N$  are the base-level learning algorithms

$m$  is the number of disjoint subsets into which  $T$  is partitioned

StratifiedPartition( $T, m$ ) returns a stratified partition of  $T$  into  $m$  equally sized subsets

Fig. 9.17: Meta-decision tree building algorithm

do learn at the meta level, most model combination methods fail to produce any real generalizable insight about learning, except in the case of arbitrating and meta-decision trees, where new metaknowledge is explicitly derived in the combination process. As stated in Vilalta et al. (2004),

“by learning or explaining what causes a learning system to be successful or not on a particular task or domain, [metalearning seeks to go] beyond the goal of producing more accurate learners to the additional goal of understanding the conditions (e.g., types of example distributions) under which a learning strategy is most appropriate.”

## References

- Alpaydin, E. and Kaynak, C. (1998). Cascading classifiers. *Kybernetika*, 34:369–374.
- Breiman, L. (1996). Bagging predictors. *Machine Learning*, 24(2):123–140.
- Brown, G. (2005). Ensemble learning – on-line bibliography. <http://www.cs.bham.ac.uk/gxb/ensemblebib.php>.

- Caruana, R., Niculescu-Mizil, A., Crew, G., and Ksikes, A. (2004). Ensemble selection from libraries of models. In *Proceedings of the 21st International Conference on Machine Learning*, ICML'04, pages 137–144. ACM.
- Chan, P. and Stolfo, S. (1993). Toward parallel and distributed learning by meta-learning. In *Working Notes of the AAAI-93 Workshop on Knowledge Discovery in Databases*, pages 227–240.
- Chan, P. and Stolfo, S. (1997). On the accuracy of meta-learning for scalable data mining. *Journal of Intelligent Information Systems*, 8:5–28.
- Efron, B. (1983). Estimating the error of a prediction rule: Improvement on cross-validation. *Journal of the American Statistical Association*, 78(382):316–330.
- Ferri, C., Flach, P., and Hernandez-Orallo, J. (2004). Delegating classifiers. In *Proceedings of the 21st International Conference on Machine Learning*, ICML'04, pages 289–296.
- Frank, E. and Witten, I. H. (1998). Generating accurate rule sets without global optimization. In *Proceedings of the 15th International Conference on Machine Learning*, ICML'98, pages 144–151.
- Freund, Y. and Schapire, R. (1996a). A decision-theoretic generalization of on-line learning and an application to boosting. In *Proceedings of the European Conference on Computational Learning Theory*, pages 23–37.
- Freund, Y. and Schapire, R. (1996b). Experiments with a new boosting algorithm. In *Proceedings of the 13th International Conference on Machine Learning*, ICML'96, pages 148–156.
- Fürnkranz, J. (1999). Separate-and-conquer rule learning. *Artificial Intelligence Review*, 13:3–54.
- Gama, J. and Brazdil, P. (2000). Cascade generalization. *Machine Learning*, 41(3):315–343.
- Jacobs, R. A., Jordan, M. I., Nowlan, S. J., and Hinton, G. E. (1991). Adaptive mixture of local experts. *Neural Computation*, 3(1):79–87.
- Jordan, M. I. and Jacobs, R. A. (1994). Hierarchical mixtures of experts and the EM algorithm. *Neural Computation*, 6:181–214.
- Kaynak, C. and Alpaydin, E. (2000). Multistage cascading of multiple classifiers: One man's noise is another man's data. In *Proceedings of the 17th International Conference on Machine Learning*, ICML'00, pages 455–462.
- Kittler, J., Hatef, M., Duin, R. P. W., and Matas, J. (1998). On combining classifiers. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20:226–239.
- Koppel, M. and Engelson, S. P. (1997). Integrating multiple classifiers by finding their areas of expertise. In *Proceedings of the AAAI-96 Workshop on Integrating Multiple Learned Models*.
- Opitz, D. and Maclin, R. (1999). Popular ensemble methods: An empirical study. *Journal of Artificial Intelligence Research*, 11:169–198.
- Ortega, J. (1996). *Making the Most of What You've Got: Using Models and Data to Improve Prediction Accuracy*. PhD thesis, Vanderbilt University.
- Ortega, J., Koppel, M., and Argamon, S. (2001). Arbitrating among competing classifiers using learned referees. *Knowledge and Information Systems Journal*, 3(4):470–490.
- Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Francisco, CA.
- Schapire, R. (1990). The strength of weak learnability. *Machine Learning*, 5(2):197–227.
- Ting, K. and Witten, I. (1997). Stacked generalization: When does it work? In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 866–871.

- Ting, K. M. and Low, B. T. (1997). Model combination in the multiple-data-batches scenario. In *Proceedings of the Ninth European Conference on Machine Learning (ECML-97)*, pages 250–265.
- Todorovski, L. and Džeroski, S. (2003). Combining classifiers with meta-decision trees. *Machine Learning*, 50(3):223–249.
- Tsymbol, A., Puuronen, S., and Terziyan, V. (1998). A technique for advanced dynamic integration of multiple classifiers. In *Proceedings of the Finnish Conference on Artificial Intelligence (STeP'98)*, pages 71–79.
- Vilalta, R., Giraud-Carrier, C., Brazdil, P., and Soares, C. (2004). Using meta-learning to support data-mining. *International Journal of Computer Science Applications*, I(1):31–45.
- Waterhouse, S. R. and Robinson, A. J. (1994). Classification using hierarchical mixtures of experts. In *IEEE Workshop on Neural Networks for Signal Processing IV*, pages 177–186.
- Webb, G. I. (1997). Decision tree grafting. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 846–851.
- Wolpert, D. H. (1992). Stacked generalization. *Neural Networks*, 5(2):241–259.

## Metalearning in Ensemble Methods

**Summary.** This chapter discusses some approaches that exploit metalearning methods in ensemble learning. It starts by presenting a set of issues, such as the ensemble method used, which affect the process of ensemble learning and the resulting ensemble. In this chapter we discuss various lines of research that were followed. Some approaches seek an ensemble-based solution for the whole dataset, others for individual instances. Regarding the first group, we focus on metalearning in the construction, pruning and integration phase. Modeling the interdependence of models plays an important part in this process. In the second group, the dynamic selection of models is carried out for each instance. A separate section is dedicated to hierarchical ensembles and some methods used in their design. As this area involves potentially very large configuration spaces, recourse to advanced methods, including metalearning, is advantageous. It can be exploited to define the competence regions of different models and the dependencies between them.

### 10.1 Introduction

In the previous chapter (Chapter 9) we introduced several methods that combine base-level classifiers into *ensemble models*, or simply *ensembles*. Some researchers refer to these methods as *ensemble learning*. Most ensembles focus on classification only, hence the term *multiple classifier systems* is also used (Mendes-Moreira et al., 2012; Cruz et al., 2018). Ensemble models have become very popular, as they often obtain superior performance to the best base-level model that can be identified. Typically, the prediction of an ensemble is generated by combining the predictions of multiple and diverse models.

A different perspective on this issue can be obtained by generalizing the scope of the *no free lunch* (NFL) theorem (Wolpert, 1996) from tasks to subtasks, as defined by subspaces of the data. In other words, as the NFL theorem suggests that the best algorithm for different problems may vary, we can assume that the best algorithm for different subspaces of the data could vary too. Thus, one can say that the goal of ensemble learning approaches is to reduce the probability of misclassification based on any single induced model, by increasing the system's area of expertise through combination of the (typically more localized) expertise of multiple models.

The process of learning and using an ensemble clearly involves two levels. Base-level models are obtained by analyzing the data from the ML tasks at hand. On the other hand, the combination of those models is a meta-level operation, even if, in some cases,

it may be rather simple (i.e., voting or averaging). For this reason, the term *metalearning* has sometimes been used to characterize ensemble learning methods. However, we note that the definition of metalearning used in this book is more restrictive (see the *Preface* to this book). Consequently, if we used our definition, not all ensemble systems would be characterized as metalearning systems.

Nevertheless, given the relevance of ensemble learning, a question arises as to what the role of metalearning/AutoML approaches is in this process. Different answers can be obtained depending on which perspective is taken. If we consider an ensemble as an algorithm, then the general aim of metalearning/AutoML methods is to recommend the most suitable algorithm (i.e., an ensemble) for the given task. However, as ensemble learning is a complex process involving multiple steps and models, metalearning can play an important role in this process too. So, from this perspective, metalearning can be used (1) to select a subset of models to make a prediction for a particular observation, (2) to estimate how accurate the prediction of a base-level model for a given observation is and use this information in the process of ensemble learning, or (3) to recommend the best possible method at each step of ensemble learning.

In this chapter we provide more details on different approaches that have been proposed on this topic. But first, let us review some basic characteristics that can be used to categorize different ensemble learning systems.

## 10.2 Basic Characteristics of Ensemble Systems

### Do we want to exploit an existing portfolio of ensembles?

Many users often deal with multiple tasks, which may be rather similar. If they opt for a solution that involves ensembles, they may have a portfolio of various methods already available when a new task is encountered. The user then has a choice of two possible approaches. One involves a search through the existing portfolio for the best possible solution. More details on this are given in Section 10.3. The other involves ensemble learning with the objective of designing the best possible ensemble for the current task. This issue is discussed in Section 10.4.

### Are predictions for the whole dataset or for each instance?

Some ensemble learning systems come up with a solution (i.e., an ensemble) for the whole dataset. Others take into account the characteristics of each instance and tailor the ensemble to this. This process is often referred to as *dynamic selection*, or *dynamic classifier selection* if the underlying task is classification. More details on this are given in Section 10.5.

### Which ensemble method is used?

In Chapter 9 we described various types such as bagging, where the votes are weighed, boosting and stacking, among other approaches. Metalearning approaches for algorithm selection include many of these methods from the given pool of alternatives.

In approaches where metalearning is used as part of the ensemble learning process, the most popular method is stacking (Pinto et al., 2016; Narassiguin et al., 2017; Wistuba

et al., 2017; Khiari et al., 2019). However, other methods, such as bagging (e.g., Pinto et al. (2014)), have also been used.

Ensemble systems, such as ALMA (Houeland and Aamodt, 2018) and metalearning algorithm templates (Kordík et al., 2018), are more general, as they can involve many different ensemble methods and can also exploit metalearning.

### **Are the models generated with a single or different algorithms?**

Heterogeneous ensembles are often preferred, as in principle, they promote more diverse components (e.g., classifiers) (Kuncheva and Whitaker, 2003).

### **Is the metadata from the current or past datasets?**

We can distinguish two kinds of systems: some use just the metadata obtained on the current dataset, while others also explore the metaknowledge obtained on other datasets in previous experiments. Many of the metalearning approaches in ensemble learning learn from the current dataset only.

### **What is the base-level learning task?**

The area of machine learning includes different tasks, such as, classification, multi-label classification and regression, among others. Some ensemble learning approaches are specific to tasks of one type, while others are more general and can deal with different task types (i.e., regression and classification).

## **10.3 Selection-Based Approaches for Ensemble Generation**

Selection-based approaches rely on a good portfolio which can include good representatives of both base-level and ensemble-based methods. Methods exist that enable to explore vast configuration spaces and identify a useful subset of models for a given set of tasks (see Chapter 8).

Once a portfolio has been defined, it can be reused for new tasks. Various methods described in this book allow us to identify the best possible model (here an ensemble of models). One of these is the simple ranking approach discussed in Chapter 2. The approach described in Chapter 5 allows to identify the best possible algorithm (here a particular ensemble method) by also taking into account the existing metaknowledge associated with the current dataset and the past datasets (e.g., dataset characteristics, etc.). More details about this can be found in Chapter 4.

One disadvantage of this approach is that it requires the existence of a portfolio, which could be very large if we were to include all possible useful variants. This problem can be minimized by applying a portfolio reduction technique described in Chapter 8 (Section 8.5), which works like pruning. It eliminates sub-standard and redundant algorithms (here ensemble methods). The experiments of Cachada et al. (2017) described briefly in Chapter 7 (Section 7.4) have shown that the average ranking method AR\* can outperform AutoWeka when the time budget is low. The portfolio used in these experiments included various algorithms, while about a half were different ensemble models.

This approach has the limitation that the configuration space is finite and hence may have a difficulty in keeping up with other methods that explore much larger configuration spaces.

## 10.4 Ensemble Learning (per Dataset)

Ensemble learning approaches build the best possible ensemble from given base-level models. This process typically involves various phases and can be iterative. More details on this are given in the next subsection.

### Phases of ensemble learning

The phases of ensemble learning can be divided into three parts: generation, pruning, and integration (Mendes-Moreira et al., 2012) (Figure 10.1). More details on each are given in the following subsections.

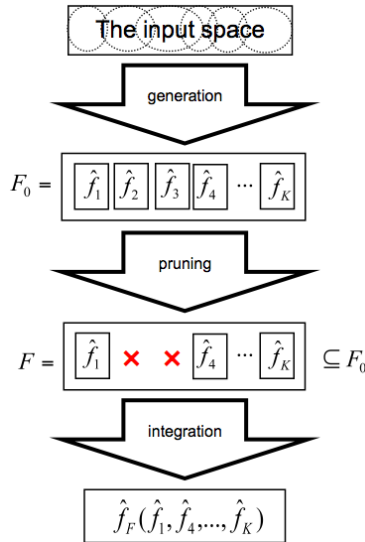


Fig. 10.1: The ensemble learning process (reproduced from Mendes-Moreira et al. (2012))

### 10.4.1 Metalearning in construction and pruning phases

#### Generation and pruning

Generation is concerned with obtaining a pool of diverse and sufficiently accurate models (Dietterich, 2000). This involves selecting the data (e.g., a dataset or some part), selecting an appropriate machine learning (ML) algorithm, and conducting training. The selection of the ML algorithm needs to obey various criteria. First, it needs to consider the given task. If, for instance, the aim is to obtain an ensemble for classification tasks, we need to consider a pool of classification algorithms. Pruning removes some of the models which are not considered useful (Mendes-Moreira et al., 2012).



Pinto et al. (2015) proposed a variant of bagging, in which a kind of pre-pruning is integrated with model generation. This is done by eliminating bootstrap samples that are not expected to generate useful models. Metalearning was used to predict whether a sample would generate a model that could improve the accuracy of the ensemble or not. This approach aims to reduce the computational cost of generating useless models.

Automatic Frankenstein of Wistuba et al. (2017) employs a multi-layer stacking of tuned models with weighting. In the generation phase, meta-level models are used to predict the execution time of runs of hyperparameter optimization. This permits to decide which sets of hyperparameters to test. The technique of *sequential model-based optimization* (SMBO), discussed in Chapter 6 (Section 6.4), is used in this process.

In general, metalearning can be used to identify which models should be generated. In order for the ensemble to be better than the models included in it, the models need to be diverse (Kuncheva and Whitaker, 2003). So we need measures of the diversity of two (or more) elements. Various measures have been proposed in the past. One is so-called *classifier output difference* (COD) (Peterson and Martinez, 2005). More details about this measure can be found in Chapter 8 (Section 8.5). Another possible measure is the  $Q$ -statistic (Kuncheva and Whitaker, 2003).

In some approaches, such as *boosting* (Chapter 9, Subsection 9.1.2), generation is done in an iterative fashion, similar to the process of building a decision tree, where a partially expanded tree is used as the basis for further extensions. The current model (a partially constructed ensemble) is used as the basis for various extensions, which are evaluated with respect to a given measure (or measures), and the best option is evaluated. So the metaknowledge acquired in this process is used to direct the search towards the most promising regions of the whole space. This has the advantage that the search is limited to only a small portion of the whole space.

A good solution to the problem of generating diverse models would possibly eliminate the need for the pruning phase, as was done in the approach proposed by Cruz et al. (2018).

## Reusing the selection-based approaches for ensemble learning

An important decision concerning the design of ensemble models is which base-level learning algorithm to consider for the generation of models. The type of base-level task significantly reduces the choices available for selection of ensemble elements. If the base-level task is classification (regression), we need to consider solely classification (regression) algorithms as potential members of the ensemble. The method described in Section 10.3 can be adapted to generate candidates for an ensemble.

As we have mentioned, earlier selection-based approaches rely on a portfolio which should include both well-performing and diverse algorithms. More details about how this portfolio can be generated on the basis of past experiments are given in Chapter 8. Once a portfolio has been defined, it can be reused in new tasks. Various methods described in this book can be used to identify a subset of algorithms that is well suited for a given task. Typically, we would want to keep including algorithms in an iterative manner until some stopping condition has been satisfied. One possibility is to require that the ensemble includes, at most,  $n$  members. Another, probably better option is to require that each new member should contribute in some way to the ensemble. One possible criterion to use here is the estimate of *expected performance gain* discussed in Chapter 5 (Section 5.8).

We note that this approach can take into account also the existing dataset characteristics of the current task (dataset) and this way reuse meta-level information regarding

which base-level algorithms were useful on similar datasets used in the past (Pinto et al., 2014; Wistuba et al., 2017; Kordik et al., 2018; Houeland and Aamodt, 2018).

### Choice of ML algorithm at the meta-level

When used in the ensemble generation/pruning phases, metalearning usually involves a standard machine learning task, such as classification or regression. When the process of generating models is sequential, then label ranking could also be used, although, to the best of our knowledge, this has never been done.

Since the metalearning task is, most of the time, a standard one, this means that off-the-shelf algorithms are used, such as decision trees, support vector machines (SVMs), random forest, and lazy learning.

### Modeling interdependence of models

As explained earlier, the contribution of a single model to an ensemble depends on the remaining models. This represents an opportunity for metalearning, as it can be used to characterize the relation between models and/or data. It is possible to take into account characteristics of the current task (dataset) and this way reuse meta-level information regarding relations between models. In Pinto et al. (2014), a variant of bagging is proposed where this is done in an indirect way. Rather than comparing two models, they compare each sample with the original dataset. Metalearning is then used to identify situations guaranteeing that learning a new model from a sample is worthwhile.

### Metafeatures

Most metalearning approaches in ensemble learning use many of the metafeatures discussed in Chapter 4. The common ones are the simple, statistical, and information-theoretic metafeatures (Pinto et al., 2014; Wistuba et al., 2017), as well as landmarkers (Pinto et al., 2014).

As mentioned above, as ensembles involve multiple models, metafeatures that quantify relations between pairs of models should be considered, such as COD (Peterson and Martinez, 2005) and the  $Q$ -statistic (Kuncheva and Whitaker, 2003), discussed earlier in Subsection 10.4.1. They were used on landmarking models obtained with different bootstrap samples to estimate their redundancy (Pinto et al., 2014).

One approach to generate diversity of the models is by altering the data used to learn them (e.g., resampling the original dataset). In these cases, and for metalearning purposes, the diversity between those models can be estimated by quantifying the differences between those samples. This can be done by measuring the difference between data distributions using, for instance, the Kullback–Leibler divergence (Cruz et al., 2018), or between metafeature values (Pinto et al., 2014). The distance between the corresponding sample and the original dataset has been used to select the bootstrap samples that generate useful models (Pinto et al., 2014).

An alternative approach to algorithm/model selection is based on the prediction of performance or execution time (Wistuba et al., 2017). In these approaches, the configurations of the learning process represented by the meta-examples can also be used as metafeatures. So these can include the values of specific hyperparameter settings used in the experiment (Wistuba et al., 2017).

## Strategy employed in Auto-sklearn

The strategy employed in Auto-sklearn involves two phases. In the first one, the system seeks not just one, but a set of good base-level solutions. In the second phase some of these solutions are selected to form an ensemble. More details on this approach can be found in Feurer et al. (2015a) and Feurer et al. (2019).

### 10.4.2 Metalearning in the integration phase

#### Integration

Given the set of models that result from the generation and pruning phases of the ensemble learning process and a new observation, a prediction is obtained by combining the individual prediction of each of the models. The question is how to combine the predictions of the various models into a single one. Existing approaches range from simple approaches, such as voting, to complex ones that adjust the combination method to the specific observation at hand. The latter is discussed in Section 10.5.

As pointed out earlier, the contribution of a single model to the ensemble depends on the other models in the ensemble (Pinto et al., 2016). Therefore, every model that is integrated into the ensemble, or eliminated from it, affects not only the performance of the ensemble directly but also the contribution of all the other models in the ensemble. Besides, it also affects the contribution of other candidate models that could be considered in the future. More details on the so-called *marginal contribution* of algorithms can be found in Chapter 8 (Section 8.4).

The marginal contribution of individual elements is related to the so-called *competence region* of each model. In other words, it is necessary to determine subspaces of the given set of tasks and the associated datasets in which the model has good performance (e.g., a high mean and a low variance). This subspace is usually referred to as the *competence region* and is usually associated with a particular algorithm or the trained model.

The competence region of a particular algorithm may include, say, classification of a certain type of datasets (e.g., image datasets or datasets with correlated features), or just a certain type of instances of a given dataset. The second option is explored in the approaches that involve so-called *dynamic selection of models*, discussed in Section 10.5. The aim of these approaches is to identify a set of competent models for each example.

#### Metalearning method

Metalearning can also play an important role in the integration step. In fact, in its simplest form, it is clearly a metalearning problem: given an observation, which subset of the generated models should be used to make the prediction? We also discuss the metalearning method. The metafeatures, which are particularly challenging in this case, are discussed further on.

## 10.5 Dynamic Selection of Models (per Instance)

Dynamic selection approaches generate a large number of models and, given a new instance, combine them (e.g., assign weights or select a subset). The term *dynamic classifier selection* (DCS) is used when the approach is applied to classification settings.

## Reusing the selection-based approaches for ensemble learning

As explained earlier, DCS can be addressed as a classification/recommendation problem: given an observation, the meta-decision tree selects the most appropriate model to make a prediction. System MetaBags follows this approach, using *meta-decision trees*. Meta-decision trees were proposed by Todorovski and Džeroski (2000) and are described in Chapter 9 (Section 9.5). Curiously, MetaBags uses an ensemble method at the meta-level as well, by bagging the meta-decision trees.

### Layer structure in system ALMA

System ALMA (Houeland and Aamodt, 2018) is an abstract ML framework consisting of a hierarchical structure of components of learning systems and includes layers representing algorithms, , and meta-algorithms for dynamic classifier selection. This approach can be used in conjunction with voting and weighting. It involves a lazy metalearning approach for algorithm selection. The authors report that the system can use metaknowledge from both the current dataset and other datasets. The experiments presented, however, focused on metalearning from the current dataset.

As DCS is concerned with a choice of models for each example, the extension to streaming scenarios is easier than for other systems that exploit batch data.

### Modeling interdependence of models

As explained earlier, the contribution of a single model to an ensemble depends on the remaining models. This is true for the combination of models to make the prediction for a single instance, as well as for the construction and pruning of models.

DCS can also be addressed as a multi-target prediction problem: given an observation and a set of models, the question is which ones to use. The term *multi-target prediction* is a broad name for ML tasks that take into account the interdependence between multiple decisions (Waegeman et al., 2019). *Multi-label classification* (MLC) is a multi-target prediction task in which the goal is to predict which of the labels from a given set should be assigned to an observation (Read et al., 2019).

Therefore, dynamic classifier selection (DCS) can be addressed as a MLC problem, where the labels are the models (Pinto et al., 2016; Narassiguin et al., 2017). That is, given the set of  $N$  models  $H = \{h_1, \dots, h_N\}$  resulting from the generation and pruning phases, the goal is to select the right subset of those models  $H_i \subseteq H$  to make predictions for observation  $i$ .

The systems CHADE (Pinto et al., 2016) and PCC-DES (Narassiguin et al., 2017) use a multi-label classification-based metalearning approach to address the problem of dependency between the models. A stacking-inspired approach is used to predict whether a model in the ensemble will accurately predict a new example or not. The algorithms used were classifier chains (Pinto et al., 2016) and probabilistic classifier chains (Narassiguin et al., 2017).

#### 10.5.1 Metafeatures

In the case of metalearning for dynamic classifier selection, where the target is a single example, computing traditional metafeatures is a challenge. The reason for this is that

traditional metafeatures characterize sets of examples, so they cannot be applied directly to a single observation.

One approach is to compute statistics in the neighborhood of the target example (Khiari et al., 2019). Another possibility is to rely on a prior clustering of examples. Classifiers are not selected for an example, but rather for a group of examples that appear to be similar. These approaches enable the use of standard data characterization measures, such as the ones discussed in Chapter 4. Britto et al. (2014) also used a *problem complexity* metafeature.

Model-based metafeatures are not as common. This is not surprising, as this type of metafeatures is also not very common in traditional metalearning scenarios. An exception is in MetaBags (Khiari et al., 2019), which includes a new type of model-based metafeatures, referred to as *local landmarks*. That is: given an example, they characterize the leaf of a tree where it falls, namely its *depth* and *the number of examples in that leaf*. We note that, despite the name, these are actually model-based metafeatures.

## Using base-level features and predictions as metafeatures

The challenge of computing metafeatures on individual observations also creates an opportunity. Since each meta-example represents a single example, the original attributes can also be used as metafeatures (Pinto et al., 2016; Khiari et al., 2019). Furthermore, we note that in stacking approaches the predictions of models are also used as metafeatures (Narassiguin et al., 2017; Khiari et al., 2019).

But these metafeatures can be generalized as a different type of landmarks that characterize the behaviour of models for a particular example. For instance, a stacking-inspired set of metafeatures has been used by Pinto et al. (2016), consisting of predictions of whether each candidate model would make correct predictions or not.

## 10.6 Generation of Hierarchical Ensembles

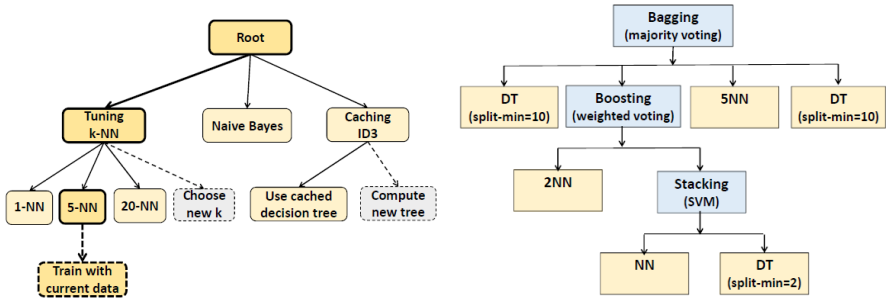
Metalearning methods have also been integrated into two general machine learning frameworks: *metalearning algorithm templates* (MAT) (Kordík et al., 2018) and ALMA (Houeland and Aamodt, 2018). Both approaches represent ensembles with a hierarchical structure (Subsection 10.6.1), but employ metalearning in different ways: MAT is a search-based ensemble construction method that uses metalearning to initialize the search with promising solutions (Section 10.6.2). On the other hand, in ALMA, metalearning is at the core of the ensemble construction method (Section 10.6.3).

### 10.6.1 Hierarchical Ensembles

Hierarchical ensembles are, as the name suggests, a hierarchical structure in the form of a tree. Examples are shown in Figure 10.2. The internal nodes can include ensembles, which in turn can include either base-level algorithms or other ensembles as members. The leaf nodes include just base-level algorithms.

### 10.6.2 Evolving hierarchical ensembles with evolutionary computing

Kordík et al. (2018) have developed a system that aims to generate the best possible hierarchical ensemble for the target dataset with recourse to *evolutionary computation*



(a) (reproduced from Kordík et al. (2018)) (b) (reproduced from Houeland and Aamodt (2018))

Fig. 10.2: Examples of hierarchical combinations of algorithms

(EC). This system can include different types of ensembles, including bagging, boosting, cascading, and arbitrating.

The process starts with a set of simple solutions, which are evolved into more complex ones. This process is controlled both by fitness and *templates* that embody the authors’ knowledge of how a given structure could be extended.

The templates can be represented in a form similar to ontologies or context-free grammar (CFG) rules discussed in Chapter 7 (Section 7.2). As pointed out in that chapter, they embody a certain declarative and procedural bias that constrains the search. The EC algorithm searches for the optimal hierarchical ensemble for the given task.

The system uses the most promising workflows identified on the past problems to initialize the search. These workflows can be regarded as metaknowledge acquired on past tasks. This strategy can be related to the process of initializing the search for the best parameter settings described by Feurer et al. (2015b). Chapter 6 (Section 6.8) discusses this in detail.

As this process can be rather slow when the dataset is large, the authors develop their solutions on smaller data samples and then apply them to full data.

The system has been applied to several concrete tasks. The authors have shown how one particular evolved template (simple ensemble of fast sigmoidal regression models) outperformed the state-of-the-art approaches on a rather large Airline dataset.

### 10.6.3 Metalearning in hierarchical ensemble methods

In ALMA (Houeland and Aamodt, 2018), metalearning is at the core of the ensemble generation process. It organizes this process into three layers, representing models, algorithms, and meta-algorithms, respectively. In the metalearning layer, a lazy learning algorithm is used for algorithm selection.

The selection-based approach described in Section 10.3 could be extended to generate hierarchical ensembles using a phased approach that would proceed in layers. In the first phase, it would generate a larger set of potentially useful ensembles. Caution could be taken to avoid branches in the search space that are not likely to lead to a useful outcome (e.g., not allow bagging ensembles of base-level algorithms with low variance, etc.). A smaller subset of this set would be identified using the reduction technique described in Section 10.3. This set would then be added to the existing portfolio, and the

process would then be repeated. It would be interesting to see how this approach would work in practice.

## 10.7 Conclusions and Future Research

Ensemble learning consists of learning systems that combine multiple diverse models to obtain more accurate predictions. This approach is based on the idea that different models specialize in different subspaces of data of the given task. In this chapter we have discussed various lines of research that were followed. As we have shown, some approaches seek an ensemble-based solution for the whole dataset, others for individual instances. We have used this criterion and discussed each group of approaches in a separate section. Hierarchical ensembles were also separated out, although the methods used to construct them are not so different from the simpler counterparts, although, of course, the configuration space is much larger.

So our aim was to clarify the role of metalearning in ensemble learning and how it can be integrated into the whole process. As this area involves potentially very large configuration spaces, recourse to advanced methods, including metalearning, is really a *must*. Metalearning can be exploited to define the competence regions of different models and the dependencies between them. The challenge is how to find a good way of doing this, so that the search for new and useful ensemble-based solutions would become easier.

## References

- Britto, A. S., Sabourin, R., and Oliveira, L. E. (2014). Dynamic selection of classifiers—A comprehensive review. *Pattern Recognition*, 47(11):3665–3680.
- Cachada, M., Abdulrahman, S., and Brazdil, P. (2017). Combining feature and algorithm hyperparameter selection using some metalearning methods. In *Proc. of Workshop AutoML 2017, CEUR Proceedings Vol-1998*, pages 75–87.
- Cruz, R. M., Sabourin, R., and Cavalcanti, G. D. (2018). Dynamic classifier selection: Recent advances and perspectives. *Information Fusion*, 41:195–216.
- Dietterich, T. G. (2000). Ensemble methods in machine learning. In *Multiple Classifier Systems. MCS 2000*, volume 1857 of LNCS. Springer.
- Feurer, M., Klein, A., Eggenberger, K., Springenberg, J., Blum, M., and Hutter, F. (2015a). Efficient and robust automated machine learning. In Cortes, C., Lawrence, N., Lee, D., Sugiyama, M., and Garnett, R., editors, *Advances in Neural Information Processing Systems 28*, NIPS’15, pages 2962–2970. Curran Associates, Inc.
- Feurer, M., Klein, A., Eggenberger, K., Springenberg, J. T., Blum, M., and Hutter, F. (2019). Auto-sklearn: Efficient and robust automated machine learning. In Hutter, F., Kotthoff, L., and Vanschoren, J., editors, *Automated Machine Learning: Methods, Systems, Challenges*, pages 113–134. Springer.
- Feurer, M., Springenberg, J., and Hutter, F. (2015b). Initializing Bayesian hyperparameter optimization via meta-learning. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, pages 1128–1135.
- Houeland, T. G. and Aamodt, A. (2018). A learning system based on lazy metareasoning. *Progress in Artificial Intelligence*, 7(2):129–146.

- Khiari, J., Moreira-Matias, L., Shaker, A., Ženko, B., and Džeroski, S. (2019). MetaBags: Bagged meta-decision trees for regression. In *Proceedings of ECML/PKDD 2018*, pages 637–652. Springer.
- Kordík, P., Černý, J., and Frýda, T. (2018). Discovering predictive ensembles for transfer learning and meta-learning. *Machine Learning*, 107(1):177–207.
- Kuncheva, L. I. and Whitaker, C. J. (2003). Measures of Diversity in Classifier Ensembles and Their Relationship with the Ensemble Accuracy. *Machine Learning*, 51(2):181–207.
- Mendes-Moreira, J., Soares, C., Jorge, A. M., and Sousa, J. F. D. (2012). Ensemble approaches for regression. *ACM Computing Surveys*, 45(1):1–40.
- Narassiguin, A., Elghazel, H., and Aussem, A. (2017). Dynamic Ensemble Selection with Probabilistic Classifier Chains. In *Machine Learning and Knowledge Discovery in Databases, European Conference, ECML PKDD 2017, Lecture Notes in Computer Science*, volume 10534 LNAI, pages 169–186. Springer, Cham.
- Peterson, A. H. and Martinez, T. (2005). Estimating the potential for combining learning models. In *Proc. of the ICML Workshop on Meta-Learning*, pages 68–75.
- Pinto, F., Soares, C., and Mendes-Moreira, J. (2014). An empirical methodology to analyze the behavior of bagging. In *Advanced Data Mining and Applications. ADMA 2014. Lecture Notes in Computer Science*, volume 8933. Springer, Cham.
- Pinto, F., Soares, C., and Mendes-Moreira, J. (2015). Pruning bagging ensembles with metalearning. In *International Conference on Advanced Data Mining and Applications, Lecture Notes in Computer Science*, volume 9132, pages 64–75. Springer, Cham.
- Pinto, F., Soares, C., and Mendes-Moreira, J. (2016). CHADE: Metalearning with classifier chains for dynamic combination of classifiers. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases, ECML PKDD 2016, Lecture Notes in Computer Science*, volume 9851 LNAI, pages 410–425. Springer, Cham.
- Read, J., Pfahringer, B., Holmes, G., and Frank, E. (2019). Classifier chains: A review and perspectives. *arXiv preprint arXiv:1912.13405*.
- Todorovski, L. and Džeroski, S. (2000). Combining multiple models with meta decision trees. In Zighed, D. A., Komorowski, J., and Zytchow, J., editors, *Proc. of the Fourth European Conf. on Principles and Practice of Knowledge Discovery in Databases*, pages 255–264. Springer-Verlag.
- Waegeman, W., Dembczyński, K., and Hüllermeier, E. (2019). Multi-target prediction: a unifying view on problems and methods. *Data Mining and Knowledge Discovery*, 33(2):293–324.
- Wistuba, M., Schilling, N., and Schmidt-Thieme, L. (2017). Automatic Frankensteining: Creating complex ensembles autonomously. In *Proceedings of the 2017 SIAM International Conference on Data Mining*, pages 741–749. Society for Industrial and Applied Mathematics, Philadelphia, PA.
- Wolpert, D. (1996). The lack of a priori distinctions between learning algorithms. *Neural Computation*, 8:1341–1390.



---

## Algorithm Recommendation for Data Streams

**Summary.** This chapter focuses on metalearning approaches that have been applied to data streams. This is an important area, as many real-world data arrive in the form of a stream of observations. We first review some important aspects of the data stream setting, which may involve online learning, non-stationarity, and concept drift. Then we focus on three types of approaches to algorithm recommendation that exploit metalearning. The first group splits the data stream into different parts and extracts meta-features for each part. This information is used to decide which machine learning method (e.g., a classifier) should be applied for the next part. The approaches in the second group build an ensemble in an online manner. Their performance is monitored and the ones that have good performance on the recent part of the data are selected for the next part of the data stream. The third group of approaches aims to exploit recurring concepts in the data. Indeed, many data have a seasonality effect (e.g., data measured on weekdays vs data measured during weekends), and by applying metalearning we can reuse older models whenever appropriate. Finally, this chapter closes with some open research questions and directions for future work.

### 11.1 Introduction

Real-time analysis of data streams is a key area of data mining research. Many real-world collected data is in fact a stream, where observations come in one by one, and algorithms processing these observations are often subject to time and memory constraints. Some examples of data streams are:

- Stock prices and trade markets. The price of a stock constantly fluctuates, and is influenced by factors in the recent past. A commonly used dataset that embodies this concept is the electricity dataset, in which the goal is to predict whether the price will rise or fall. This dataset is available on OpenML<sup>1</sup> and UCI.
- Chemical compounds. The composition of chemical compounds is determined with the help of various sensors under strictly controlled conditions. Over time, the sensors can deteriorate or even fail, but the model still needs to determine the composition correctly. This dataset is available on OpenML<sup>2</sup> and UCI.

---

<sup>1</sup><https://www.openml.org/d/151>

<sup>2</sup><https://www.openml.org/d/1476>

- Human body measurements, such as electroencephalography (EEG) data. EEG is a monitoring method to record the electrical activity of the brain by attaching several sensors onto the head. The assumption with this kind of data is that, based on recent brain waves, it can be used to predict several events in the body. A common example of EEG data is the *eye-state database*, where the goal is to predict whether the human subject will have his eyes open or closed, based on the EEG data. This dataset is available on OpenML<sup>3</sup> and UCI.

In all these examples above, the goal is to predict something in the future (electricity price, chemical compound, and the action of blinking or not). A classifier can be trained to make these predictions, but later on, as the correct value is observed, the model can be retrained or suitably adapted.

There are several key differences between data streams and conventional types of data that have been explored in various other chapters of this book. The most important differences, as highlighted by various authors (see, e.g., Domingos and Hulten (2003); Gama et al. (2009); Bifet et al. (2010); Read et al. (2012)), are:

**Non-stationarity** The data is of non-stationary nature, that is, the order of the examples matters. This, for example, prohibits the use of cross-validation in evaluation, as some of the basic assumptions are violated.

**Online learning** Observations come at different points in time. This means that the algorithm has to handle the observations on a one-by-one basis, and hence should be updatable.

**Infinity** An algorithm should expect an infinite stream of observations, limiting the options in terms of computational complexity.

**Concept drift** A learned concept can change over time. When analyzing financial data, after a disrupting event occurs on the market, the existing models can become obsolete. This phenomenon is called *concept drift*. Data stream classifiers should be able to detect these events and act accordingly (e.g., by updating or replacing the model).

These aforementioned properties impose some requirements on algorithms that model data streams, as pointed out by, e.g., Bifet et al. (2010) and Read et al. (2012):

**Data processing** Process one observation at a time, and inspect it at most once. As data streams generally consist of large amounts of data, it is not feasible to keep all data in memory. Of course, in many applications a pre-defined number of observations can be stored for later inspection (e.g.,  $k$ -NN), but the decision of whether to store or disregard an observation has to be made on the fly.

**Resources** Expect an infinite stream, but process it under finite resources. Ideally, the processing of a given observation (either for training or prediction) should require a constant amount of time. For large data streams, the memory requirements might also become an issue, so proper memory management needs to be applied.

**Prediction** Be ready to predict at any time. As observations come in one by one, the model might require a sufficient amount of data before it is capable of providing accurate predictions. This can occur especially at the beginning of a stream.

## Formalization

Formally, a data stream is an ordered collection of  $n$  base-level observations,  $\mathcal{D}^{base} = ((\mathbf{x}_i^{base}, y_i^{base}) \mid i = 1, \dots, n)$ , to map an input  $\mathbf{x}_i^{base}$  to an output  $f(\mathbf{x}_i^{base})$ , which closely

<sup>3</sup><https://www.openml.org/d/1471>

represents  $y_i^{base}$ . Models  $f(\mathbf{x}_i^{base})$  that work well on some parts of the stream might become obsolete and outdated due to the aforementioned concept drift. The research community has developed a large number of machine learning algorithms capable of online modeling of general trends in stream data and providing accurate predictions for future observations.

### 11.1.1 Adapting batch classifiers to the data stream setting

Some batch classifiers can be adapted to a data stream setting. Examples are  $k$  nearest neighbor (Beringer and Hüllermeier, 2007; Zhang et al., 2011), stochastic gradient descent (SGD) (Bottou, 2004), and SPegasos (Stochastic Primal Estimated sub-Gradient SOLver for SVMs) (Shalev-Shwartz et al., 2011). Both stochastic gradient descent and SPegasos are gradient descent methods capable of learning a variety of linear models, such as support vector machines and logistic regression, depending on the chosen loss function.

Other classifiers have been created specifically to operate on data streams. Most notably, Domingos and Hulten (2000) introduced the *Hoeffding tree* induction algorithm, which inspects every example only once, and stores per-leaf statistics to calculate the *information gain* on which the split criterion is determined. The *Hoeffding bound* states that the true mean of a random variable of a given range will not differ from the estimated mean by more than a certain value. This provides statistical evidence that a certain split is superior to others. As Hoeffding trees seem to work very well in practice, many variants have been proposed, such as Hoeffding option trees (Pfahlinger et al., 2007), adaptive Hoeffding trees (Bifet and Gavaldà, 2009) and random Hoeffding trees (Bifet et al., 2012).

Moreover, a commonly used technique to adapt traditional batch classifiers to the data stream setting is training them on a window of  $w$  recent examples: after  $w$  new examples have been observed, a new model is built. This approach has the advantage that old examples are ignored, providing natural protection against concept drift. A disadvantage is that it does not operate directly on the most recently observed data, until  $w$  new observations are made and the model is retrained. Read et al. (2012) compare the performance of these *batch-incremental* classifiers with common data stream classifiers, and conclude that the overall performance is equivalent, although batch-incremental classifiers generally use more resources.

Finally, Finn et al. (2019) propose an online version of MAML. MAML is a meta-learning approach to optimizing the initial parameters (rather than hyperparameters) of gradient-based models and will be explained in detail in Chapter 13. The basic idea is to set the parameters for the next time step equal to the best parameters in hindsight.

### 11.1.2 Adapting ensembles to the data stream setting

As shown in Chapter 9, ensemble techniques train multiple classifiers, which are then used to produce a prediction. Various schemes exist regarding how this is done. More details can be found in Chapter 9. In this section we review how some techniques, namely bagging and boosting, have been extended to the data stream setting.

Bagging (Breiman, 1996) exploits the instability of classifiers by training them on different *bootstrap replicates*, which are resampling samples (with replacement) of the training set. Online bagging (Oza, 2005) operates on data streams by drawing the weight of each example from a *Poisson*(1) distribution, which converges to the behavior of the

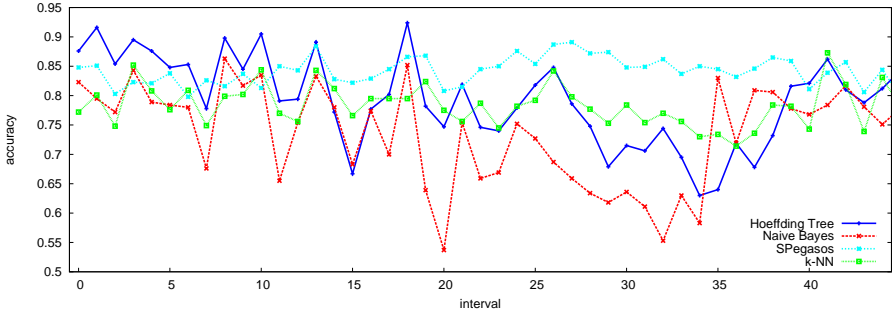


Fig. 11.1: Performance of four classifiers on intervals (size 1,000) of the electricity dataset. Each data point represents the accuracy of a classifier on the most recent interval. Figure adapted from van Rijn (2016)

classical bagging algorithm if the number of examples is large. As the Hoeffding bound gives statistical evidence that a certain split criteria is optimal, this makes them more stable and hence less suitable to use in a bagging scheme. However, in practice this yields good results. Boosting (Schapire, 1990) is a technique that sequentially trains multiple classifiers, in which more weight is given to examples that were misclassified by earlier classifiers. Online boosting (Oza, 2005) applies this technique on data streams by assigning more weight to training examples that were misclassified by previously trained classifiers in the ensemble.

Chapter 10 gives more details on how metalearning and AutoML techniques can be used to design good ensembles for the the current dataset.

### 11.1.3 Motivation

As data streams are constantly subject to change over time, the most accurate classifier for a given interval of observations also changes frequently, as illustrated by Figure 11.1. The horizontal axis represents a certain chronological point in the stream, while the vertical axis represents the performance of various online classifiers. Due to the changing behaviour of the underlying data stream, at various points in the stream, different classifiers perform best. For example, in the beginning the Hoeffding tree is superior, while later SPegasos performs best. It would be logical to dynamically adapt which learner or portfolio of learners should be applied to a given part of the stream. Hence, we are dealing with a repeating algorithm selection problem. As such, we want to minimize the overall loss expressed by  $\arg \min_{f_{meta}} \sum_i^n \mathcal{L}(f_{meta}(\mathbf{x}_i^{base}), y_i^{base})$ . Here,  $f_{meta}$  represents some dynamic procedure that determines, for each observation, which models to use for prediction. Metalearning has been successfully used to achieve this, as described further on in this chapter.

The remainder of this chapter is organized as follows: Section 11.2 demonstrates how traditional metafeatures can be used to solve this problem. Section 11.3 shows several techniques that have been developed to build ensembles that adapt to the current state of the stream. As data streams are often subject to seasonality, it is likely that some

concepts might re-occur. Section 11.4 describes several works that leverage this property. Finally, in Section 11.5, we discuss general trends and challenges for future research.

## 11.2 Metafeature-Based Approaches

As shown in Figure 11.1, not only the performance of classifiers changes over the data stream, but also the relative ranking of the classifiers. Note that this figure shows how accurate the classifiers are in a window of 1,000 independent observations. In this particular example, the Hoeffding tree performs best at the beginning of the stream, although it has several dips in performance (e.g., from interval 14 to 18 and from 46 to 41). After approximately interval 19 onwards, SPegasos is the best classifier until the end of the stream, where the Hoeffding tree,  $k$ -NN, and again SPegasos would lead to more accurate predictions.

One challenge for metalearning and algorithm selection in particular would be to dynamically predict which classifier would perform best for the next window of observations. In an ideal case, the algorithm selection system would always correctly identify which classifier to choose, leading to an accuracy that corresponds with the top boundary of the classifiers in Figure 11.1. This would yield performance gains in particular for data streams that are volatile, and for which at different intervals of observations different classifiers perform well.

Although it would be an interesting task to predict beforehand which classifier performs best on the metafeatures inferred over the complete data stream, this is not realistic; as was pointed out in Section 11.1, there is a requirement not to pass multiple times over the observations. One possibility is to take a small sample from the (beginning of the) data stream, calculate metafeatures based on this sample, and make a prediction on the basis of this (van Rijn et al., 2014). However, this is also not really practical. Due to possible concept drift, any conclusion based on the beginning of the stream might be outdated as time passes.

In order to dynamically select which classifier to use at any given point in time, we need the same components as in many other algorithm selection frameworks, that is: (i) a set of earlier observed data streams, (ii) performance values of the various classifiers (on intervals (portions) of) the data streams, and (iii) metafeatures that characterize (the intervals of) the data streams. There have been several works that attempt to do this. In this section we review some of these approaches.

### 11.2.1 Methods

Figure 11.2 shows the general framework for dynamic algorithm selection for data streams. The top of the image displays the current data stream. Each block here represents an observation in this data stream. The index of the current observation, for which a prediction is required, is indicated by  $c$ . We have seen all observations before index  $c$  and have not seen any of the observations from index  $c$  on. We are interested in selecting a classifier for the observations that are indicated by  $\alpha$ , starting with the current observation  $c$ . This can be done by extracting metafeatures based on the most recent part of the stream, indicated by the window  $w$ .

After using the selected classifier for making predictions for all the observations in interval  $\alpha$ , both the window  $w$  and interval  $\alpha$  shift by  $|\alpha|$  observations, and the algorithm selection system repeats this procedure. If the meta-model allows for incremental updates, it can be trained on the recently generated meta-observation.

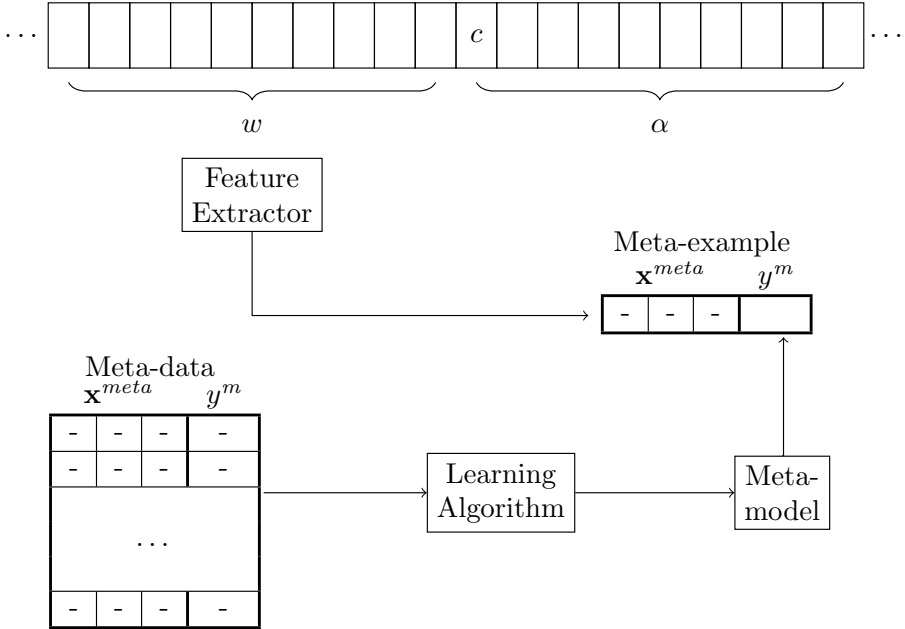


Fig. 11.2: Architecture of an algorithm selection system that dynamically selects a classifier for an interval  $\alpha$  of observations.  $y^{meta}$  is abbreviated as  $y^m$ . Figure adapted from Rossi et al. (2014)

### 11.2.2 Training the meta-model

The meta-model can be trained on metadata from the current stream (Rossi et al., 2014), metadata from previously seen streams (van Rijn et al., 2014), or metadata from both. In either case, the meta-model is trained on the same form of metadata. Generally, the meta-models are induced based on a dataset  $\mathcal{D}^{meta} = \{(\mathbf{x}_i^{meta}, y_i^{meta}) \mid i = 1, \dots, m\}$  (with  $|\mathcal{D}^{meta}| = m$ ) to map an input  $\mathbf{x}^{meta}$  to an output  $f(\mathbf{x}^{meta})$ , which closely represents  $y^{meta}$ . Here,  $\mathbf{x}^{meta}$  comprises the metafeatures calculated on some interval (section) of observations from the base data streams, and  $y^{meta}$  represents the classifier that performed best on that given interval.

The question regarding how to define the intervals (start position and length) needs to be considered. In some approaches, the size of the interval is fixed to  $w$  and the start of intervals is defined by either using a *sliding window* or *shifting window* scheme.

If the sliding window scheme is used, an interval can start at any given observation, splitting a data stream into  $n - w$  equal intervals (i.e., the intervals starting with observations  $\{1, 2, \dots, n - w\}$ ). Note that the data stream can be potentially of infinite length. As such, we consider  $n$  as the number of observations in the part of the data that we analyze. Each observation in the stream is involved in several intervals, yielding a large amount of metadata.

Alternatively, when the shifting window is used, an interval can only start any observation that is indexed by a divisor of  $w$  (i.e., the intervals starting with observations  $\{1, w + 1, 2 \times w + 1, \dots, \lceil \frac{n}{w} \rceil \times w + 1\}$ ). This way, each observation is used exactly in

one interval. This yields less training instances for the meta-model, and presumably decreases the training time and the amount of redundant information. Both Rossi et al. (2014) and van Rijn et al. (2014) opted for this setup.

### 11.2.3 Metafeatures

We can use most of the metafeatures defined in Chapter 4 on the intervals of the data streams. In the data stream literature, the following subset of metafeatures has been used: *simple* (number of examples, number of attributes), *statistical* (mean standard deviation of attributes, mean skewness of attributes), *information theoretic* (class entropy, mean mutual information), and *landmarkers* (Pfahring et al., 2000), representing performance evaluations of simple classifiers on the data stream. These are more expensive to measure but often yield good results.

Additionally, van Rijn et al. (2014) introduced the concept of *drift detection metafeatures*. These are produced by running a drift detector (in particular DDM (Gama et al., 2004) and ADWIN (Bifet and Gavaldà, 2007)) on each interval and recording the number of warnings and alarms raised. This informs the meta-algorithm that there was a change of concept and that some corrective action should be taken, such as retraining some classifier or substituting it by another one.

Finally, van Rijn et al. (2015) introduced the concept of *stream landmarks*. For each interval, the performance of all classifiers is measured and the best classifier identified. So the classifier that performed best on the previous interval can be used as an option to fall back on in the current interval.

### 11.2.4 Considerations regarding hyperparameters

The algorithm selection systems described in this chapter contain several considerations and hyperparameters that influence performance. We describe the most important ones.

**Set of base-classifiers:** The performance of any algorithm selection system is highly dependent on the algorithms that it can choose from. In general, it is good to have a diverse set of good-performing classifiers. Having more classifiers potentially makes the learning task harder, as the base classifiers might be less represented in the meta-dataset. Chapter 8 elaborates on a suitable choice of base-classifiers; further in this chapter we will also address this issue specific to data stream classifiers.

**Set of metafeatures:** Like most metalearning systems, the performance heavily relies on metafeatures. Many metafeatures that have been discussed in Chapter 4 can be used on the interval of the data streams. However, metafeatures such as *number of features* and *number of observations* in the interval will obviously be constant throughout the stream and will therefore not be able to trigger a dynamic switch of classifiers.

**Meta-model:** The meta-model that is trained on the meta-dataset that includes metafeatures as attributes. The quality of the model will depend on the setup, and hence also metafeatures. An accurate model will make the right choices of base-classifiers, whereas an inaccurate model might do the opposite. Both stream-based algorithms (e.g., Hoeffding trees) and batch algorithms (e.g., random forest) have been applied in the past.

**Metafeature window size:** The size of the window on which metafeatures are calculated (denoted as  $\alpha$  in Fig. 11.2). Small windows will not be able to capture trends in the data, whereas large windows will not allow for a fast adaptation of classifiers.

Table 11.1: Comparisons between two works on metalearning with data streams

	Rossi et al. (2014)	van Rijn et al. (2014)
Metadata	Generated on current stream	Collected from other streams
Meta-model	Stream model	Batch learner
Learning paradigm	Regression	Classification
$ A $ (algorithms)	2	5

**Prediction window size:** This determines the number of observations for which the same base-classifier will be used (denoted as  $w$  in Fig. 11.2). Setting this number low will result in a potentially too frequent switching of the active classifier, while setting this number high will have the opposite effect. van Rijn et al. (2014) suggested that the window size  $w$  should be set to be a multiple of the  $\alpha$  size. For example, when  $\alpha$  is set to 100, then  $w$  can be set to either of the values  $\{100, 200, 300, \dots\}$ . This way, all previously seen observations will have an influence on an equal number of prediction windows.

### 11.2.5 Meta-model

The most important component of metalearning systems is a meta-model. The online algorithm recommendation framework, as outlined in this chapter, has been studied in two works referred to in Table 11.1. This table outlines the main differences between both experimental setups.

The work of van Rijn et al. (2014) requires a meta-dataset built over other data streams, following the usual metalearning framework. The work of Rossi et al. (2014) did not require such a meta-database. The metadata was generated from earlier intervals on the current stream, following the style that is common to AutoML. Although it requires some time for the system to build up a sufficient amount of metadata to be able to construct a good model, this alleviates two problems: (i) the burden of collecting a set of representative metadata on other datasets, and (ii) statistical metafeatures, such as *standard deviation*, can be generated on a per column basis.

Another prominent difference is regarding the meta-model: Rossi et al. (2014) used a stream-based meta-model, which is updated whenever new metadata is gathered; van Rijn et al. (2014) used a batch version of random forests, which is trained once on the metadata and does not require updates. Other differences are the learning paradigm used in the evaluation (regression streams versus classification data streams) and the number of algorithms that have been considered.

### 11.2.6 Evaluation of metalearning systems for data streams

Evaluation of metalearning and AutoML systems is discussed in Chapter 3. Many of the concepts and methods can be reused for the task of evaluating metalearning and AutoML systems oriented to data streams.

As pointed out in Chapter 3, it is important to distinguish between *base-level performance* and *meta-level performance*. Base-level performance represents the performance



that would be obtained per dataset (or data stream) if the recommendation of the meta-algorithm was followed. Meta-level performance is the performance of the meta-model at the task of selecting a good base-level algorithm (e.g., a classifier if the task is classification).

When dealing with data-streaming tasks which involve intervals (sections of the data stream), it is necessary to introduce the concepts of base-level and meta-level performance. The *base-level performance per interval* determines the performance of the system relative to a specific interval. The *base-level performance across  $n$  intervals* (or simply *base-level performance*) returns the average of the above measures across the larger portion of the data stream of interest, given the performance of the base-classifiers that were selected per interval.

The value of the *meta-level performance per interval* is 1 if the correct base-level algorithm was selected, and 0 otherwise. The *meta-level performance across  $n$  intervals* (or simply *meta-level performance*) returns the average of the above measure across  $n$  intervals.

### 11.2.7 Baselines

As in other areas of machine learning and metalearning, we need good baselines against which the proposed systems can be compared. In the following we describe some baselines that have been proposed in the past:

**Average best classifier (AvBest)** is the classifier that obtained the highest base-level accuracy on all the data streams in the training set.

**Most frequent best classifier (FreqBest)** is the classifier that was the best classifier on most intervals in the meta-dataset.

**Best classifier on last interval (Blast)** selects, for each interval in the stream, the classifier that performed best on the previous interval. It will be explained formally in Section 11.3.

**Oracle** is the classifier that always selects the best classifier for each interval, i.e., its meta-performance is 1 (it is not an existing metalearning system). The oracle shows what the base-level performance would be if the meta-model always performed perfectly.

According to some experiments carried out by van Rijn et al. (2015), the variants AvBest and FreqBest have very similar performance. In general, the AvBest variant has a higher base-level accuracy, whereas FreqBest has a better meta-level performance. The Blast baseline basically embodies the no change classifier on the meta-level (Bifet et al., 2013) and is generally a very simple but yet rather strong baseline. Later work shows, somewhat surprisingly, that metalearning systems do not necessarily outperform the Blast baseline (van Rijn et al., 2015).

The oracle is a very useful concept for analyzing the performance of the meta-algorithm. If the gap between the algorithm selection system and the oracle is small, this means that the algorithm selection system performs well.

Besides, the oracle can be used to assess the adequacy of the pre-selected set of base-classifiers. If the oracle performs close to perfect prediction, this means that an adequate set of classifiers has been chosen. If this is not the case, an improvement can be made by adding other appropriate base-level classifiers to the base-level set.

### 11.2.8 Discussion

Metafeatures have been successfully applied for algorithm selection in data streams. Although there has been a lot of work on metafeatures and on data streams, to the best of our knowledge, only the aforementioned works have studied the interaction between the two.

There is a lot of progress to be made. As pointed out by the authors, not all of the defined baselines have been convincingly beaten by other variants. In particular, the Blast baseline turns out to be a very competitive baseline (more in the next section).

It would be interesting to see how metafeature approaches could be improved by taking advantage of recent developments in *batch metalearning*.

## 11.3 Data Stream Ensembles

As is well known, ensembles of classifiers have, in general, superior performance to individual classifiers. It thus makes sense to see how ensemble methods can be adapted to data stream settings.

Various types of ensemble methods were discussed in Chapter 9. In this section we will consider only some of those (e.g., bagging) and describe how this architecture can be adapted to the data stream setting. In this setting the most relevant decision is which base-classifiers (ensemble members) should be involved and how to weight their individual votes. However, due to the possible occurrence of concept drift, it is likely that more recent examples will be more relevant than older ones. Moreover, due to the fact that there is a temporal component in the data, we can measure how ensemble members have performed on recent examples and adjust their weight accordingly.

In this section we review several ensemble-based methods adapted for data streams that utilize these observations.

### 11.3.1 Best classifier on last interval (Blast)

In Section 11.2.7 we already briefly introduced the Blast baseline. It trains an array of diverse classifiers, and measures which of these performed best in the last interval. van Rijn et al. (2015) formalized this as follows: At every new observation in the stream, each ensemble member is assigned a certain score based on its performance on previous observations:

$$P_{win}(f_j, c, w, \mathcal{L}) = 1 - \sum_{i=\max(1, c-w)}^{c-1} \frac{\mathcal{L}(f_j(\mathbf{x}_i^{base}), y_i)}{\min(w, c-1)}, \quad (11.1)$$

where  $f_j$  is the learned labeling function of this specific ensemble member,  $c$  is the index of the last seen training example, and  $w$  is the number of training examples used to estimate the performance. This method has a certain start-up time (i.e., when  $w$  is larger than or equal to  $c$ ) during which we have fewer observations than we would like to have (i.e., fewer than  $w$  observations). Also note that it can only be performed after several labels have been observed (i.e., the index of the current observation  $c > 1$ ).

Finally,  $\mathcal{L}$  is a loss function that compares the labels predicted by the ensemble member with the true labels. The simplest version is a zero/one loss function, which returns 0 when the predicted label is correct and 1 otherwise. More complicated loss functions can

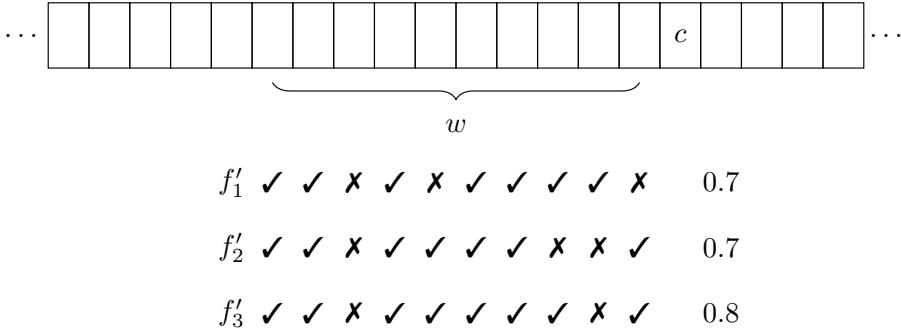


Fig. 11.3: Schematic view of window-based performance estimation. For all classifiers,  $w$  flags are stored, each indicating whether it predicted a recent observation correctly

also be incorporated. The outcome of  $P_{win}$  is in the range  $[0, 1]$ , with better-performing classifiers obtaining a higher score.

The classifier that performed best over the last interval of training examples is selected as the *active classifier* (i.e., it gets 100% of the weight). This is illustrated in Figure 11.3, where  $f'_3$  would be selected as the active classifier ( $AC_c$ ) with respect to example  $c$ . Formally, this can be expressed by

$$AC_c = \arg \max_{j \in J} P_{win}(f_j, c - 1, \alpha, \mathcal{L}), \tag{11.2}$$

where  $J$  is the set of models generated by the ensemble members,  $c$  is the index of the current example,  $\alpha$  is a parameter to be set by the user (fading factor (discussed in the next subsection)), and  $\mathcal{L}$  is a zero/one loss function, giving a penalty of 1 to all misclassified examples. As the authors have shown, this simple baseline outperforms the metafeature-based approaches.

### 11.3.2 Fading factors

Considering the above observation, a question arises as to whether metalearning approaches can be improved. It is clear that Blast has several drawbacks: (i) it requires the ensemble to store the  $w \times |J|$  additional values (recall that  $J$  is the set of classifiers that are being considered), which is inconvenient in a data stream setting, where both time and memory are important factors; (ii) it requires the user to tune parameter  $\alpha$ , which highly influences performance; and (iii) there is a hard cut-off point, i.e., an observation is either in or out of the window.

These problems can be alleviated using so-called *fading factors*, described by Gama et al. (2013). Fading factors give a high importance to recent predictions, but their importance decreases when they become older. This is illustrated in Figure 11.4. The red (solid) line corresponds to a relatively fast fading factor, and consequently the effect of a given prediction has faded away almost completely after 500 predictions. On the other hand, the blue (dashed) line corresponds to a relatively slow fading factor, where the

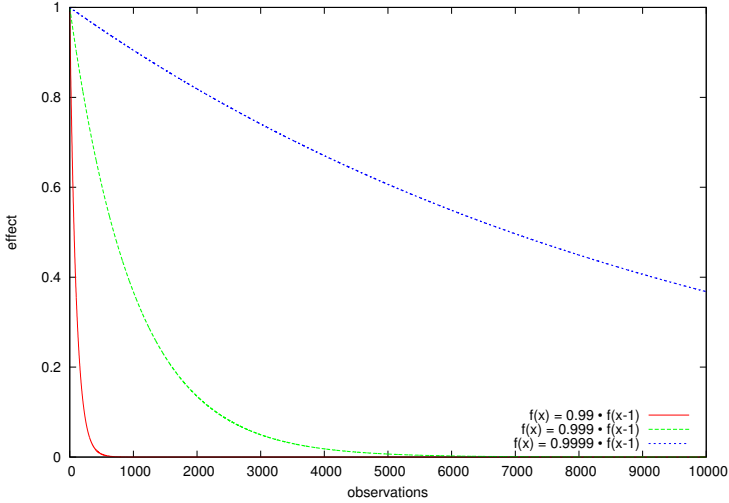


Fig. 11.4: The effect of a prediction after a number of observations, relative to when it was first observed (for various values of  $\alpha$ )

effect is still considerably high, even when 10,000 observations have passed. Note that, even though all these functions start at 1, in practice we need to scale this down to  $1 - \alpha$  in order to constrain the function within the range  $[0, 1]$ . Formally, the weighting function becomes

$$P_{ff}(f_j, c, \alpha, \mathcal{L}) = \begin{cases} 1 & \text{iff } c = 1 \\ P_{ff}(f_j, c - 2, \alpha, \mathcal{L}) \cdot \alpha + (1 - \mathcal{L}(f_j(\mathbf{x}_{c-1}^{base}), y_{c-1}^{base})) \cdot (1 - \alpha) & \text{otherwise} \end{cases} \tag{11.3}$$

where the meaning of the symbols is similar to Eq. 11.1. The fading factor  $\alpha$  (range  $[0, 1]$ ) determines at what rate the historical performance becomes irrelevant, and its value can be set (or tuned) by the user. A value close to 0 will result in rapid changes in estimated performance, whereas a value close to 1 will keep them more stable. The outcome of  $P_{ff}$  is in the range  $[0, 1]$ , with better-performing classifiers obtaining a higher score.

This notion has been used in the following works: Kolter and Maloof (2007) describe this weighting strategy and use it to build *dynamic ensembles*, that is, ensembles that grow in size whenever the current set of ensemble members make a mistake. They also introduce methods to prune the ensemble again.

van Rijn et al. (2018) used this weighting strategy to create heterogeneous ensembles of fixed size that includes different modeltypes. However, their experiments have shown that Blast achieved better performance.

Cerqueira et al. (2019) developed a method that operates in the regression setting. The method uses a meta-model that predicts the performance of each base-level model for the next observation. Based on these predictions, the weights of base-level models are determined and suitably normalized. It would be interesting to see whether the system could be improved by incorporating metafeatures.

### 11.3.3 Heterogeneous ensembles for feature drift

Nguyen et al. (2012) introduce the notion of feature drift. This arises when the set of most important features used for predicting the class changes. Besides, they show: (i) concept drift may give rise to feature drift, (ii) concept drift does not necessarily lead to feature drift, and (iii) feature drift occurs at a slower rate than concept drift.

So one could argue that systems that utilize stream ensembles should incorporate drift detection and feature selection. The method proposed by the authors employs a method for feature selection (i.e., the fast correlation-based filter algorithm by Yu and Liu (2003)). If a new set of features is detected, the model that performs badly is dropped from the ensemble and a new model is trained.

### 11.3.4 Considerations regarding the choice of base-classifiers

In the previous section we focused on how to weight the votes of individual classifiers based on recent data. Another important question is which base-level models should be considered in the first place. This issue was addressed in Chapter 8 (Section 8.3), where we discussed which base-level algorithms should be considered for inclusion in a given portfolio. Many of the concepts presented there are relevant also for the constitution of ensembles.

One such concept was *classifier output difference (COD)* (Peterson and Martinez, 2005), which can be used to detect whether two classifiers generate similar predictions. Lee and Giraud-Carrier (2011) used this function to build a hierarchical clustering of classifiers. Classifiers that generate similar predictions appear to be clustered together, and vice versa.

This idea was followed by van Rijn et al. (2018) to create a clustering of various stream classifiers from the MOA framework. The resulting dendrogram is shown in Figure 11.5. The assumption is that classifiers that are clustered far away from each other are *diverse* and hence would work well together in an ensemble.

### 11.3.5 Discussion

A popular approach to the repeated algorithm selection problem is the use of ensembles that either change (i) their set of base-learners, or (ii) the way the votes of the individual base-learners are weighted. These decisions are often based on how the classifiers perform on recent examples. The performance is often better than competitive ensembles that do not utilize this property. Unfortunately, there has been no general comparison between the various ensemble methods discussed in this chapter.

## 11.4 Recurring Meta-level Models

Another way to leverage properties of data streams is to store the meta-level models generated on old parts of the stream and learn when to reuse them. Gama and Kosina (2014) used this scheme in a sensor network consisting of several geographically distributed sensors, each producing a high-speed data stream. They measure some quantity of interest, for example, the electricity demand in a particular geographic region. Companies are interested in predicting the demand of electricity for a certain time horizon,

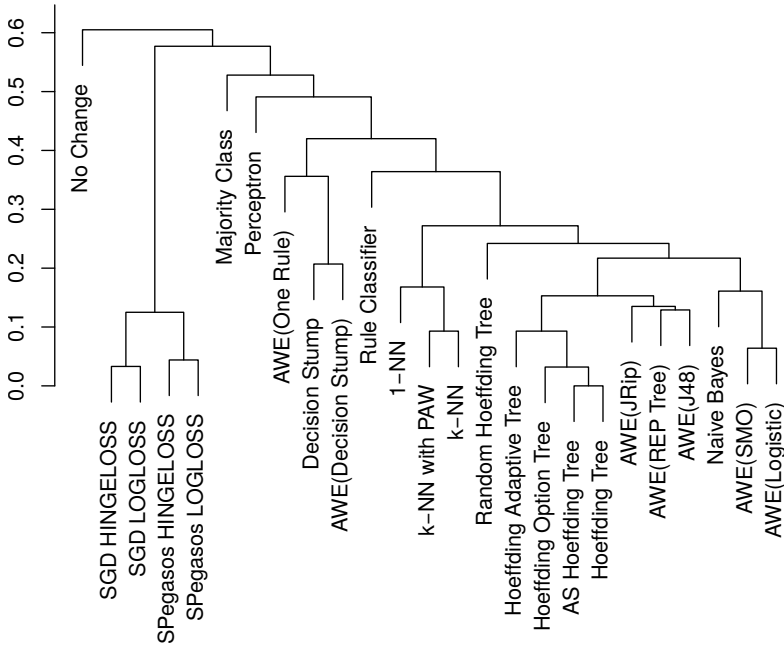


Fig. 11.5: Hierarchical clustering of stream classifiers

e.g., an hour later. At a given point in time, the model makes a prediction for the electricity demand an hour later. When this time point occurs, the sensor measures the actual electricity demand and can thus determine the loss obtained by the model. Consumption patterns change (e.g., from winter to summer) and, due to seasonality, might reoccur.

Several authors have used ideas of metalearning for this scenario. We will review two of these works here.

### 11.4.1 Accuracy-weighted ensemble

The approach proposed by Wang et al. (2003) is to separate the stream into windows of a given fixed size, and train a new classifier on each of these windows. During prediction, each classifier votes for a certain class label. The votes are weighted by their accuracy.

In addition to convincing empirical results, they provide a formal proof regarding the error rate of an ensemble of  $k$  classifiers, each trained on the data points from different windows, compared with the classifier that is trained on all data points from the last  $k$  windows. Their proof shows that the error rate of the ensemble will always be equal to or lower than the error rate of the individual classifier.

One of the most important advantages of this approach is that it does not come at the expense of additional training time, as each observation is used as training input to only one model, which is a cheap way of turning a single classifier into an ensemble. Read et al. (2012) noted that this is an opportunity to gain additional performance from batch classifiers, which must be trained on windows of data.

There are several practical considerations that need to be taken into account when employing this method. First, there is a question of the ensemble size, that is, how many members should be used. This is obviously a hyperparameter under the control of the user. Normally, higher values would be preferred, as these often lead to better performance. However, memory usage is another criterion to take into consideration when setting this hyperparameter.

Second, it is important to determine which members should be kept in memory. Once the ensemble has generated more members than it is possible to keep in memory, it is necessary to determine which members should be dropped. These can, for example, be the oldest members in memory, or the members with the worst performance on the current window (or on all windows).

Finally, the window size should be set appropriately. When the individual ensemble members are trained on a window that is too small, the models will not be able to accurately classify the data. When, on the other hand, the window is too large, the individual ensemble members will not be able to respond well to the dynamics of changing concepts.

#### 11.4.2 Two-layered architecture

Gama and Kosina (2014) proposed a two-layered architecture to handle the recurrence of concepts in the data. The first layer (*induction layer*) is responsible for doing the actual classification, whereas the second layer (*control layer*) is responsible for checking the applicability of the old, stored models. Let us see the role of each layer in more detail:

**Induction layer** This layer consists of a classifier that is specialized in the actual classification task. Whenever a new measurement is done, the classifier is trained on the most recent data.

**Control layer** This layer consists of a classifier that is specialized in determining whether the classifier from the induction layer will make the correct prediction. As such, it is involved in a binary classification task.

The classifier in the control layer can determine how well the corresponding classifier in the induction layer will work on a given window of data points.

Besides, a change detector is also used to estimate whether a concept drift occurred. The authors state that, although any change detector could be used, they suggest SPC, as it is also able to give a warning signal (Gama et al., 2004). Whenever a change in concept is detected, the meta-model has to choose between the following options: (i) train a new model, or (ii) activate one of the previously learned models.

In order to distinguish between the two options, all previously trained models will make predictions for the recent set of data points. This set can be of fixed size or include every data point since the drift detector signaled a warning. Depending on the prediction horizon, some of these data points might not yet have the associated label.<sup>4</sup> For these data points, the classifier in the control layer can be used to estimate whether they would have been classified correctly.

The classifier that performed best on this recent set of data points will be used to classify the new observations, provided its performance is better than a given threshold. So one of the stored models can be reused here. If none of the previously trained models performs better than this threshold, a new model is trained.

<sup>4</sup>In the most optimistic case, this is only one data point; in the most pessimistic case, this can be all the data points.

## 11.5 Challenges for Future Research

There remain many avenues for further research, and many questions to be answered. First, as discussed before and summarized in Table 11.1, we can generate and leverage metadata from earlier in the current stream alone, or we can do so from a larger set of previously seen data streams, when these are available. These two methods are complementary, but currently there does not yet exist clear analyses or comparisons between these approaches that explain when and how to use which approach, or whether both can be combined.

Such an analysis could elucidate several key questions: (i) which properties of a given setup contribute to its success (e.g., which types of learners to use), (ii) when a given setup works well (i.e., on what types of data), and (iii) why a given setup works well on a certain type of data.

Indeed, one of the most surprising results of the metafeature-based approaches is that they did not seem to be able to consistently outperform a simple baseline Blast. Although it is clear that this baseline would not work in highly volatile streams, there is currently no convincing argument for using metafeature-based approaches instead of Blast. Still, an interesting question is whether the metafeatures could still enhance the performance in some way, or whether better metafeatures could be included.

A second key question is whether the meta-problem should be modeled by stream learning or batch learning. In other words: should we adapt existing meta-learning and AutoML methods that use batch learners to work in the stream setting, or should we focus on techniques that do meta-learning and AutoML on stream learning algorithms directly? This should also include the tuning of hyperparameters and the inclusion of preprocessing in pipelines, which was not yet done in the studies discussed earlier in this chapter.

While, to the best of our knowledge, there do not yet exist AutoML tools that operate over stream learning algorithms, Celik and Vanschoren (2020) provide an analysis of how existing AutoML methods can be adapted to the stream setting, for instance, by restarting the AutoML process after concept drift is detected (with or without a warm start from the previous best configurations), or simply by retraining the best model(s) on the latest batches of data. This analysis shows that both Bayesian optimization and evolutionary approaches can handle concept drift well, given an appropriate adaptation strategy and forgetting mechanism. It also finds that different drift characteristics (e.g., gradual versus sudden drift) affect learning algorithms in different ways, and that different adaptation strategies may be needed to optimally deal with them. This shows that there is ample room to improve existing AutoML systems, and even to design entirely new AutoML systems that naturally adapt to concept drift.

Finally, many tools have been developed to facilitate the task of conducting meta-learning and AutoML experiments, such as OpenML (Vanschoren et al., 2014), discussed in Chapter 16. OpenML also contains several stream datasets, with and without concept drift, and it is integrated with the MOA stream mining library. As such, it provides a great starting point for novel research into metalearning and AutoML on data streams, which would certainly have a positive impact on the field, especially if more stream mining datasets can be made publicly available and further stream mining libraries are integrated to facilitate experimentation.



## References

- Beringer, J. and Hüllermeier, E. (2007). Efficient instance-based learning on data streams. *Intelligent Data Analysis*, 11(6):627–650.
- Bifet, A., Frank, E., Holmes, G., and Pfahringer, B. (2012). Ensembles of restricted Hoffding trees. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 3(2):30.
- Bifet, A. and Gavaldà, R. (2007). Learning from Time-Changing Data with Adaptive Windowing. In *SDM*, volume 7, pages 139–148. SIAM.
- Bifet, A. and Gavaldà, R. (2009). Adaptive learning from evolving data streams. In *Advances in Intelligent Data Analysis VIII*, pages 249–260. Springer.
- Bifet, A., Holmes, G., Kirkby, R., and Pfahringer, B. (2010). MOA: Massive Online Analysis. *J. Mach. Learn. Res.*, 11:1601–1604.
- Bifet, A., Read, J., Žliobaitė, I., Pfahringer, B., and Holmes, G. (2013). Pitfalls in benchmarking data stream classification and how to avoid them. In *Machine Learning and Knowledge Discovery in Databases*, pages 465–479. Springer.
- Bottou, L. (2004). Stochastic learning. In *Advanced Lectures on Machine Learning*, pages 146–168. Springer.
- Breiman, L. (1996). Bagging predictors. *Machine Learning*, 24(2):123–140.
- Celik, B. and Vanschoren, J. (2020). Adaptation strategies for automated machine learning on evolving data. *arXiv preprint arXiv:2006.06480*.
- Cerqueira, V., Torgo, L., Pinto, F., and Soares, C. (2019). Arbitrage of forecasting experts. *Machine Learning*, 108(6):913–944.
- Domingos, P. and Hulten, G. (2000). Mining High-Speed Data Streams. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 71–80.
- Domingos, P. and Hulten, G. (2003). A general framework for mining massive data streams. *Journal of Computational and Graphical Statistics*, 12(4):945–949.
- Finn, C., Rajeswaran, A., Kakade, S., and Levine, S. (2019). Online meta-learning. In Chaudhuri, K. and Salakhutdinov, R., editors, *Proceedings of the 36th International Conference on Machine Learning, ICML’19*, pages 1920–1930. JMLR.org.
- Gama, J. and Kosina, P. (2014). Recurrent concepts in data streams classification. *Knowledge and Information Systems*, 40(3):489–507.
- Gama, J., Medas, P., Castillo, G., and Rodrigues, P. (2004). Learning with drift detection. In *SBLA Brazilian Symposium on Artificial Intelligence*, volume 3171 of *Lecture Notes in Computer Science*, pages 286–295. Springer.
- Gama, J., Sebastião, R., and Rodrigues, P. P. (2009). Issues in evaluation of stream learning algorithms. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 329–338. ACM.
- Gama, J., Sebastião, R., and Rodrigues, P. P. (2013). On evaluating stream learning algorithms. *Machine Learning*, 90(3):317–346.
- Kolter, J. Z. and Maloof, M. A. (2007). Dynamic weighted majority: An ensemble method for drifting concepts. *Journal of Machine Learning Research*, 8:2755–2790.
- Lee, J. W. and Giraud-Carrier, C. (2011). A metric for unsupervised metalearning. *Intelligent Data Analysis*, 15(6):827–841.
- Nguyen, H.-L., Woon, Y.-K., Ng, W.-K., and Wan, L. (2012). Heterogeneous Ensemble for Feature Drifts in Data Streams. In *Advances in Knowledge Discovery and Data Mining*, pages 1–12. Springer.
- Oza, N. C. (2005). Online bagging and boosting. In *Systems, Man and Cybernetics, 2005 IEEE International Conference*, volume 3, pages 2340–2345. IEEE.

- Peterson, A. H. and Martinez, T. (2005). Estimating the potential for combining learning models. In *Proc. of the ICML Workshop on Meta-Learning*, pages 68–75.
- Pfahring, B., Bensusan, H., and Giraud-Carrier, C. (2000). Meta-learning by landmarking various learning algorithms. In Langley, P., editor, *Proceedings of the 17th International Conference on Machine Learning, ICML'00*, pages 743–750.
- Pfahring, B., Holmes, G., and Kirkby, R. (2007). New options for Hoeffding trees. In *AI 2007: Advances in Artificial Intelligence*, pages 90–99. Springer.
- Read, J., Bifet, A., Pfahring, B., and Holmes, G. (2012). Batch-Incremental versus Instance-Incremental Learning in Dynamic and Evolving Data. In *Advances in Intelligent Data Analysis XI*, pages 313–323. Springer.
- Rossi, A. L. D., de Leon Ferreira, A. C. P., Soares, C., and De Souza, B. F. (2014). MetaStream: A meta-learning based method for periodic algorithm selection in time-changing data. *Neurocomputing*, 127:52–64.
- Schapire, R. (1990). The strength of weak learnability. *Machine Learning*, 5(2):197–227.
- Shalev-Shwartz, S., Singer, Y., Srebro, N., and Cotter, A. (2011). Pegasos: primal estimated sub-gradient solver for SVM. *Mathematical Programming*, 127(1):3–30.
- van Rijn, J. N. (2016). *Massively collaborative machine learning*. PhD thesis, Leiden University.
- van Rijn, J. N., Holmes, G., Pfahring, B., and Vanschoren, J. (2014). Algorithm Selection on Data Streams. In *Discovery Science*, volume 8777 of *LNCS*, pages 325–336. Springer.
- van Rijn, J. N., Holmes, G., Pfahring, B., and Vanschoren, J. (2015). Having a Blast: Meta-Learning and Heterogeneous Ensembles for Data Streams. In *2015 IEEE International Conference on Data Mining (ICDM)*, pages 1003–1008. IEEE.
- van Rijn, J. N., Holmes, G., Pfahring, B., and Vanschoren, J. (2018). The online performance estimation framework: heterogeneous ensemble learning for data streams. *Machine Learning*, 107(1):149–167.
- Vanschoren, J., van Rijn, J. N., Bischl, B., and Torgo, L. (2014). OpenML: networked science in machine learning. *ACM SIGKDD Explorations Newsletter*, 15(2):49–60.
- Wang, H., Fan, W., Yu, P. S., and Han, J. (2003). Mining Concept-Drifting Data Streams using Ensemble Classifiers. In *KDD*, pages 226–235.
- Yu, L. and Liu, H. (2003). Feature selection for high-dimensional data: A fast correlation-based filter solution. In *Proceedings of the 20th International Conference on Machine Learning, ICML'03*, pages 856–863.
- Zhang, P., Gao, B. J., Zhu, X., and Guo, L. (2011). Enabling fast lazy learning for data streams. In *2011 IEEE 11th International Conference on Data Mining (ICDM)*, pages 932–941. IEEE.

## Transfer of Knowledge Across Tasks

Ricardo Vilalta and Mikhail M. Meskhi

**Summary.** This area is often referred to as *transfer of knowledge across tasks*, or simply *transfer learning*; it aims at developing learning algorithms that leverage the results of previous learning tasks. This chapter discusses different approaches in transfer learning, such as *representational transfer*, where transfer takes place after one or more source models have been trained. There is an explicit form of knowledge transferred directly to the target model or to the meta-model. The chapter also discusses *functional transfer*, where two or more models are trained simultaneously. This situation is sometimes referred to as *multi-task learning*. In this approach, the models share their internal structure (or possibly some parts) during learning. Other topics include instance-, feature-, and parameter-based transfer learning, often used to initialize the search on the target domain. A distinct topic is transfer learning in neural networks, which includes, for instance, the transfer of a part of the network structure. The chapter also presents the *double loop architecture*, where the base-learner iterates over the training set in an inner loop, while the metalearner iterates over different tasks to learn metaparameters in an outer loop. Details are given on transfer learning within kernel methods and parametric Bayesian models.

### 12.1 Introduction

Learning should not be viewed as an isolated task that starts from scratch with every new problem. Instead, a learning algorithm should exhibit the ability to adapt through a mechanism dedicated to the transfer of knowledge gathered from previous experience (Thrun and Mitchell, 1995; Thrun, 1998). The problem of how to transfer knowledge across tasks is central to the field of metalearning, and is also referred to as *learning to learn* or *transfer learning*. Here, knowledge can be understood as a collection of patterns observed across tasks. As an example, one view of the nature of patterns across tasks is that of invariant transformations. For example, image recognition of a target object is simplified if the object is invariant under rotation, translation, scaling, etc. A learning system should be able to recognize a target object in an image even if previous images show the object at different sizes or from different angles. We view transfer learning as the study of how to improve learning by detecting, extracting, and exploiting knowledge across tasks.

In this chapter, we take a look at various approaches to implement learning systems armed with the ability to transfer knowledge across tasks. We focus our description by responding to two questions: What can be transferred across tasks? What learning architectures have been commonly used for transfer learning? We also present developments in the theoretical aspects of learning to learn. Our focus is on supervised learning; other work can be found in fields such as unsupervised learning (Bengio, 2012) and reinforcement learning (Taylor and Stone, 2009).

## 12.2 Background, Terminology, and Notation

We focus on the task of supervised learning or classification where we are given the task of inducing a model from a sample  $\{(x, y)\}$ , where the vector  $x$  is an instance (feature vector) of the input space  $\mathcal{X}$ , and  $y$  is an instance of the output space  $\mathcal{Y}$ . The sample contains independently and identically distributed (i.i.d.) examples that come from a fixed but unknown joint probability distribution,  $P(X = \mathbf{x}, Y = y)$ , in the input–output space  $\mathcal{X} \times \mathcal{Y}$ . The output of the learning algorithm is a hypothesis (i.e., model, function)  $h(X)$  mapping the input space to the output space,  $h : \mathcal{X} \rightarrow \mathcal{Y}$ . The function  $h$  comes from a space of hypotheses  $\mathcal{H}$ . The idea is to search for the hypothesis that minimizes the expectation of a loss function  $L(Y, h(X))$ , a.k.a. the risk:  $R(h) = E_{P(X, Y)}[L(Y, h(X))]$ .

### 12.2.1 When is transfer learning applicable?

In transfer learning, we assume the existence of a *source domain*  $\mathcal{D}_S$  from which we can leverage experience to generate an accurate model on the *target domain*  $\mathcal{D}_T$ . Ultimately, the main goal is to induce an accurate model  $h_T(X)$  on the target domain. The need to transfer knowledge across domains is prompted by the change of at least one of the following elements between domains:  $\{\mathcal{X}, P(X), \mathcal{Y}, P(Y|X)\}$  (each element will normally be labeled with a subscript to differentiate between source and target domains, e.g.,  $\mathcal{X}_S$  and  $\mathcal{X}_T$ ). Let us follow a concrete case study to understand these elements. If we assume the learning task of inducing a model to predict disease from laboratory tests in a medical facility, the first element refers to the case where the feature space differs,  $\mathcal{X}_S \neq \mathcal{X}_T$ , as would happen if two medical centers rely on different laboratory tests. The second element refers to the marginal distribution  $P(X) = \int_{\mathcal{Y}} P(X, Y) d_Y$ ; it can be illustrated as two medical centers having populations of patients exhibiting differences in demographics,  $P_S(X) \neq P_T(X)$ . The third element refers to the output or class-label space; this would correspond to the case where two medical centers aim at predicting different diseases,  $\mathcal{Y}_S \neq \mathcal{Y}_T$ . The last element, the class posterior probability, refers to the scenario where, due to environmental, genetic, or other factors, disease is manifested differently across two medical centers,  $P_S(Y|X) \neq P_T(Y|X)$ . Transfer learning is justified when one or more of these elements differ across the source and target domains.

Remember that the emphasis is always placed on the target domain  $\mathcal{D}_T$ , corresponding to the task at hand. The main objective is to induce a model  $h_T(X)$  for the target domain; when building the model, one can exploit knowledge from the source domain  $\mathcal{D}_S$ . A cautionary note applies when the similarity between the source and target domains is poor; it may occur that an attempt to leverage information from the source domain leads to a loss of generalization performance on the target domain. This effect, also known as *negative transfer* (Torrey and Shavlik, 2010), places a boundary on the potential benefits of adapting models to new domains.

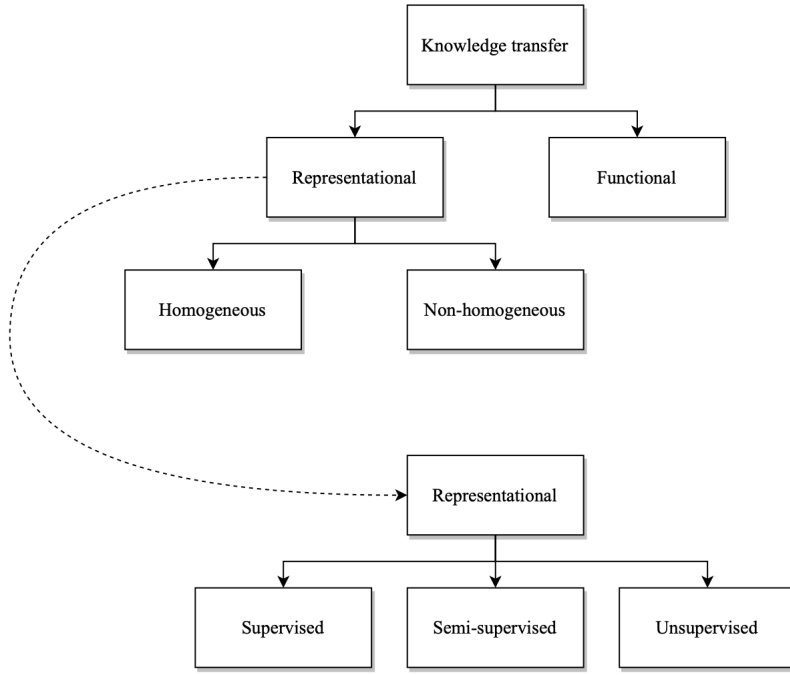


Fig. 12.1: A taxonomy of different approaches to knowledge transfer

### 12.2.2 Types of transfer learning

Different approaches are available to transfer knowledge across tasks (Weiss et al., 2016). A proposed taxonomy is shown in Figure 12.1. We use the term *representational transfer* to denote the case where the target and source models are trained at different times and the transfer takes place after the source model has already been trained; here, there is an explicit form of knowledge transferred into the target model. In contrast, we use the term *functional transfer* to denote the case where two or more models are trained simultaneously; in this case, the models share (part of) their internal structure during learning (e.g., multi-task learning). When the transfer of knowledge is explicit, as is the case with representational transfer, further distinctions can be made. First, in terms of the input or feature space, we can have source and target domains sharing the same input space, also known as *homogeneous transfer* (Weiss et al., 2016), or conversely, we can have source and target domains not sharing the same input space, also known as *non-homogeneous transfer*. In terms of the availability of class labels, we denote as *unsupervised transfer* the case where both source and target datasets contain no class labels. We denote as *semi-supervised transfer* the case where the source dataset contains labels, but the target dataset contains no class labels (or very few) (e.g., domain adaptation). Finally, we denote as *supervised transfer* the case where both the source and target datasets contain class labels. The need for transfer learning often points to target

datasets with few or no class labels from which it is difficult to build accurate models. But it is important to note that transfer learning is also applicable to datasets with abundant class labels, where the goal is to improve over previous mistakes, further restricting the size of the hypothesis space.

### 12.2.3 What can be transferred?

While many different types of knowledge can be transferred across domains, popular techniques can be divided into three categories: *instance-based transfer learning*, *feature-based transfer learning*, and *parameter-based transfer learning*. We briefly review each technique in turn.

**Instance-based transfer learning.** The first type of knowledge transfer, instance-based transfer learning, aims at identifying instances on the source domain that seem to be *closer* to the distribution on the target domain. The idea in instance-based methods is to assign high weights to source examples occupying regions of high density in the target domain. A popular approach is known as the covariate shift (Quionero-Candela et al., 2009; Shimodaira, 2000; Kanamori et al., 2009; Sugiyama et al., 2008; Bickel et al., 2009). The covariance-shift assumption is that one can build a model on the newly weighted source sample and apply it directly to the target domain (Gretton et al., 2009). Specifically, we adopt the assumption that the difference in the source  $P_S(X, Y)$  and target  $P_T(X, Y)$  distributions is due to a covariate shift, i.e.,  $P_S(X) \neq P_T(X)$ , whereas the conditional probabilities remain the same  $P_S(Y|X) = P_T(Y|X)$ . In this case, we can redefine the risk as  $R(h) = E_{\sim P_T(X, Y)}[L(Y, h(X))]$ ,  $R(h) = E_{\sim P_T(X, Y)}[\frac{P_T(X, Y)}{P_S(X, Y)} L(Y, h(X))]$ ,  $R(h) = E_{\sim P_T(X, Y)}[\beta(X, Y)L(Y, h(X))]$ . By obtaining the value of  $\beta(X, Y)$  on every source instance  $X$ , we can minimize the risk on the target domain. A stringent requirement, however, is that the source and target distributions must be close to each other.

**Feature-based transfer learning.** The second type of knowledge transfer, feature-based transfer learning, aims at finding a common representation where both the source and target distributions overlap. Feature-based methods attempt to project the source and target datasets into a latent feature space where the covariate-shift assumption holds. A model is then built on the transformed space and used as the classifier on the target. Examples are structural corresponding learning (Blitzer et al., 2006) and subspace alignment methods (Basura et al., 2013), among others.

**Parameter-based transfer learning.** The third type of knowledge transfer, parameter-based transfer learning, aims at generating a good set of initial parameters to expedite the model building phase on the target domain. As an illustration, we may perform an exhaustive search for the right model parameters on a source domain, where we can generate a set of prior distributions. Upon the arrival of a new target task, transfer learning obviates such exhaustive search; instead, we can generate a posterior distribution on the target (using the source to obtain the priors) that would lead to finding a near-optimal set of target model parameters.

## 12.3 Learning Architectures in Transfer Learning

Many experiments in supervised learning have been reported within the neural network community, but other architectures have also played an important role. Besides neural networks, this section includes kernel methods and parametric Bayesian methods.

### 12.3.1 Transfer in neural networks

A learning paradigm amenable to testing the feasibility of knowledge transfer is that of neural networks. A neural network is capable of expressing flexible decision boundaries over the input space (Goodfellow et al., 2016); it is a nonlinear statistical model that applies to both regression and classification. In particular, for a neural network with one hidden layer, each output node computes the following function:

$$g_k(X = \mathbf{x}) = f\left(\sum_l w_{kl} f\left(\sum_i w_{li}x_i + w_{l0}\right) + w_{k0}\right), \quad (12.1)$$

where  $\mathbf{x}$  is the input feature vector,  $f(\cdot)$  is a nonlinear (e.g., sigmoid, ReLU) function, and  $x_i$  is a component of the vector  $\mathbf{x}$ . The index  $i$  runs along the components of the vector  $\mathbf{x}$ , the index  $l$  runs along the number of intermediate functions (i.e., nonlinear transformations of the input features), and the index  $k$  refers to the  $k$ th output node. The output is a nonlinear transformation of the intermediate functions. The learning process is limited to finding appropriate values for all the weights  $\{w\}$ . The concepts described below are equally valid for *deep neural networks* (Goodfellow et al., 2016), where there is more than just one hidden layer between the input and output nodes.

Neural networks have received much attention in the context of knowledge transfer because one can exploit the final set of weights of the *source* network (i.e., of the network obtained on a previous task) to initialize the set of weights corresponding to the *target* network (i.e., to the network corresponding to the current task). We describe different strategies to transfer knowledge between neural network models.

**Functional transfer in neural networks.** Most approaches to transfer learning in neural networks follow a representational approach, where some knowledge is explicitly transferred from the source network to the target network. But a functional approach is also popular, where several networks are combined into a single network architecture enabling different tasks to share the same hidden representation; this field is also known as *multi-task learning* (Argyriou et al., 2007). As an illustration, Figure 12.2 shows two networks, one intended to classify stars and the other galaxies, that can be combined into one single architecture where hidden nodes now capture patterns that are common across both domains.

**Sharing part of the neural network structure.** In general, many hybrid variations have been tried around the central idea of sharing a neural network structure, often by combining different forms of knowledge transfer. Examples include dividing the neural network into two parts: a common structure at the bottom of the network (i.e., a set of adjacent layers next to the input layer) capturing a common task representation, and a set of upper structures (i.e., a set of adjacent layers next to the output layer), each focused on learning a specific task (Yosinski et al., 2014). Specifically, a source domain with an abundance of labeled examples can be exploited to generate a network model with high generalization performance. New target domains with limited training samples

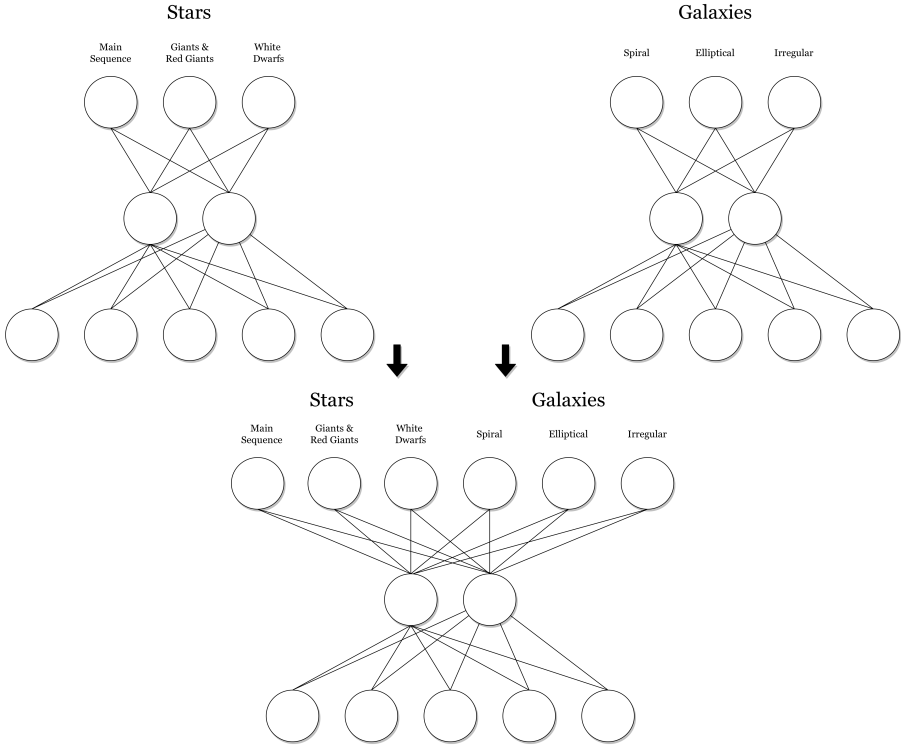


Fig. 12.2: One can combine tasks together into a single parallel multi-task problem; here, multiple luminous objects are identified in parallel using a common hidden layer

can reuse the bottom layers of the source network, while simply adjusting the weights on the upper part of the target network (Heskes, 2000; Yosinski et al., 2014).

**Searching for invariant transformations.** An interesting example of an application of knowledge transfer in neural networks is the search for certain forms of invariant transformations. We mentioned before the importance of finding such transformations in the context of image recognition. As an illustration, suppose we have gathered images of a set of objects under different angles, brightness, location, etc. Let us assume our goal is to automatically learn to recognize an object in an image, using images containing the same object (albeit captured in different conditions) as experience.

One way to proceed is to train a neural network to learn an invariant function  $\sigma$ . Function  $\sigma$  is trained with pairs of images generated under different conditions to identify when the images contain the same object. If the function is approximated with no error, one could perfectly predict the type of object contained in one image by simply applying  $\sigma$  over the current image and previous images containing several prototype objects. In practice, however, finding  $\sigma$  can be intractable, and information about the shape



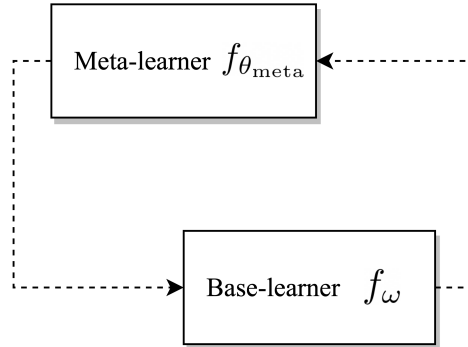


Fig. 12.3: The double-loop architecture

of the invariant function (e.g., function slopes) can be used to improve the accuracy of the learner (Thrun and Mitchell, 1995; Zaheer et al., 2017).

**Nested learning and  $k$ -shot learning.** A general way to depict a metalearning algorithm is to divide its internal architecture into two main components: a base learner and a metalearner. The base learner works as is traditional in supervised learning, by inducing a model from a set of labeled examples by searching for near-optimal model parameters on a specific task (or episode). The metalearner instead takes on the role of learning patterns (i.e., knowledge) across tasks to simplify the task of each base learner. This can be visualized as a double-loop architecture (Vilalta and Drissi, 2002; Bertinetto et al., 2019), where the base learner iterates over a training set to learn model parameters under a fixed hypothesis space, in what is described as the *inner loop*, while concurrently, the metalearner iterates over different tasks to learn metaparameters under a family of hypothesis spaces, in what is described as the *outer loop* (see Figure 12.3).

This double-loop architecture has seen an upsurge of different techniques and settings (Finn et al., 2017), particularly in the neural network community. A typical application is the  $n$ -way  $k$ -shot learning task, where the challenge is to train a (deep) neural network with very few examples, specifically, to induce an accurate model with only  $k$  examples for each of the  $n$  possible classes. This is only possible if the metalearner has captured relevant patterns across multiple tasks. We briefly illustrate some instances of these ideas.

- **Learning similarity functions.** One form of metalearning uses the source task to learn a *similarity function* that can accurately predict if two objects belong to the same class (Koch et al., 2015; Chopra et al., 2005). This is different from traditional supervised learning, where the classifier receives two examples (i.e, two feature vectors) as input and predicts if they belong to the same class or not. This verification problem can be exploited by transferring such a similarity function to the target domain. In one-shot learning, for example, the single labeled example on the target task can replace one matching element of the similarity function, while the other element corresponds to a target testing example. The nested learning framework can be effected by minimizing a loss over each task or episode corresponding to a

specific target sample (inner loop), while improving on the similarity function across many learning tasks (outer loop) (Vinyals et al., 2016).

- **Learning with recurrent neural networks.** One advantage of the double-loop view of metalearning is that fixed update routines can be transformed into adaptable modules, amenable to learning. A typical framework for learning update rules is that of recurrent neural networks, particularly long short-term memory (LSTM), where the ability to remember past events provides feedback to improve the update mechanism itself (Hochreiter et al., 2001). As an illustration, a recurrent neural network can be designed with a double-loop architecture, where a search for model parameters on the base learner (optimizee) for a specific task is guided by a metalearner (optimizer) in charge of learning the update rule itself after seeing several tasks (Andrychowicz et al., 2016; Ravi and Larochelle, 2017).
- **Bidirectional feedback between learner and metalearner.** One prominent line of research is to increase the interdependence between the base learner and the metalearner by adjusting the optimization process to ensure feedback is sent in both directions (Maclaurin et al., 2015; Finn et al., 2017, 2018; Bertinetto et al., 2019). Specifically, a base learner can update its parameters  $\theta'$  by relying on a global meta-parameter  $\theta$  (controlled by the metalearner) for parameter initialization. In the context of stochastic gradient descent (SGD), a single update step can be defined as

$$\theta'_i = \theta - \alpha \nabla_{\theta} \mathcal{L}_{\mathcal{T}_i}(f_{\theta}), \quad (12.2)$$

where the second term on the right hand side of the equation is the gradient over the loss function on task  $\mathcal{T}_i$ . The update step above defines the inner loop (see the previous discussion), but notice the dependence on the global parameter  $\theta$ . The outer loop is effected when  $\theta$  is updated after seeing several tasks:

$$\theta = \theta - \beta \sum_{\mathcal{T}_i} \nabla_{\theta} \mathcal{L}_{\mathcal{T}_i}(f_{\theta'_i}), \quad (12.3)$$

where the metaparameter  $\theta$  is based on the sum of local gradients. In effect, the metalearner provides an initial set of parameters on each task  $\mathcal{T}_i$ , to update  $\theta_i$  in few steps (Finn et al., 2017, 2018).

- **Memory-augmented neural networks.** Another interesting direction is to enhance neural networks to remember past events by adding memory components (Graves et al., 2014). In transfer learning, this leads to models that remember past events and can generalize to new tasks by leveraging past experience (Santoro et al., 2016; Munkhdalai and Yu, 2017), overcoming the *catastrophic forgetting* typical of deep networks. Memory becomes then an additional component of the neural network, with the capacity to store and retrieve representations relatively fast. This is critical in a  $k$ -shot learning scenario, where generalizing with few examples is difficult, requiring the storage of newly observed events. Here, the inner loop of metalearning is achieved by quickly retrieving instances for which proper generalization has not been reached, with an outer loop where the slow acquisition of patterns across tasks or episodes leads to robust and stable models.

### 12.3.2 Transfer in kernel methods

Kernel methods such as support vector machines (SVMs) have been extended to work on multi-task learning. Kernel methods look for a solution to the classification (or regression) problem using a discriminant function  $g(\cdot)$  of the form

$$g(X = \mathbf{x}) = \sum_j c_j k(\mathbf{x}_j, \mathbf{x}), \quad (12.4)$$

where  $\{c_j\}$  is a set of real parameters, the index  $j$  runs along the number of training examples, and  $k$  is a kernel function in a reproducing kernel Hilbert space (Shawe-Taylor and Cristianini, 2004).

Knowledge transfer can be effected using kernel methods by forcing the different hypotheses (corresponding to the different tasks) to share a common structure. As an illustration, consider the space of hypotheses made of hyperplanes, where every hypothesis is represented as  $\mathbf{w} \cdot \mathbf{x}$  (i.e., as the inner product of  $\mathbf{w}$  and  $\mathbf{x}$ ). To employ the idea of having multiple tasks, we assume we have several datasets  $\mathbf{T} = \{T_p\}_{p=1}^n$ . Our goal is to produce hypotheses  $\{h_p\}_{p=1}^n$  from  $\mathbf{T}$  under the assumption that the tasks are related. The idea of task relatedness can be incorporated by modifying the space of hypotheses so that the weight vector is made of two components:

$$\mathbf{w}_p = \mathbf{w}_0 + \mathbf{v}_p, \quad 1 \leq p \leq n, \quad (12.5)$$

where we assume all models share a common model  $\mathbf{w}_0$ , and the vectors  $\mathbf{v}_j$  serve to model each particular task. In this case, we are in effect forcing all hypotheses to share a common component while also allowing for deviations from the common model (Evgeniou and Pontil, 2004).

### 12.3.3 Transfer in parametric Bayesian models

One type of knowledge transfer uses a Bayesian model by computing the posterior probability of each class  $y$  given an input vector  $\mathbf{x}$ ,  $P(Y = y|X = \mathbf{x})$ . For a fixed class  $y$ , Bayes theorem results in the following formula:

$$g(\mathbf{x}) = P(Y = y|X = \mathbf{x}) = \frac{P(\mathbf{x}|y)P(y)}{P(\mathbf{x})} \quad (12.6)$$

where  $P(y)$  is the prior probability of class  $y$ ,  $P(\mathbf{x}|y)$  is the likelihood of  $y$  with respect to  $\mathbf{x}$  or the class-conditional probability, and  $P(\mathbf{x})$  is the evidence (Duda et al., 2001). Under this framework, a parameter-based transfer learning approach is to train a Bayesian learning algorithm on source domain  $D_S$ , resulting in a predictive model with a parameter vector  $\theta_S$  that embeds the set of probabilities required to compute the posterior probabilities. For a new target domain  $D_T$ , we require that the new probability vector  $\theta_T$  be similar to the previous one (i.e.,  $\theta_S \sim \theta_T$ ). To accomplish this, we assume that each component parameter of  $\theta_S$  and  $\theta_T$  stems from a hyper-prior distribution. The degree of similarity between parameter components can be controlled by forcing the hyper-prior distribution to have small variance (corresponding to similar tasks) or large variance (corresponding to dissimilar tasks) (Rosenstein et al., 2005; Cao et al., 2013).

**Transfer by clustering.** One approach to learning to learn consists of designing a learning algorithm that groups similar tasks into clusters. A new task is assigned to the most

related cluster; knowledge transfer takes place when generalization exploits information about the cluster to which each task belongs. This idea of clustering similar tasks has also been pursued under a Bayesian approach. Essentially, each vector of hidden to output weights can be modeled as a mixture of Gaussians, where each Gaussian is in fact describing a cluster of tasks (Bakker and Heskes, 2003; Thrun and O’Sullivan, 1998).

## 12.4 A Theoretical Framework

Several studies have provided a theoretical analysis of the learning-to-learn paradigm. The aim is to understand the conditions under which a metalearner can provide good generalizations when embedded in an environment made of related tasks. Although the idea of knowledge transfer is normally made implicit in the analysis, it is clear that the metalearner extracts and exploits knowledge from every task to perform well on future tasks. Theoretical studies fall within a Bayesian model (Baxter, 1998; Heskes, 2000) and a probably approximately correct (PAC) model (Baxter, 2000; Maurer, 2005). The idea is to find not only the right hypothesis  $h$  in a hypothesis space  $\mathcal{H}$ ,  $h \in \mathcal{H}$ , but also to find the right hypothesis space  $\mathcal{H}$  in a family of hypothesis spaces  $\mathbb{H}$ ,  $\mathcal{H} \in \mathbb{H}$ .

Let us look at these studies more closely. We focus on the problem of bounding the number of examples needed to produce good generalizations when the learner faces a stream of tasks. Consider first that the goal of traditional learning is to find a hypothesis  $h^* \in \mathcal{H}$  that minimizes a functional risk,  $h^* = \arg \min_{h \in \mathcal{H}} R_\phi(h)$ , where

$$R_\phi(h) = \int_{(\mathbf{x}, y) \in \mathcal{X} \times \mathcal{Y}} L(h(\mathbf{x}), y) d\phi(\mathbf{x}, y), \quad (12.7)$$

The risk corresponds to the expected loss incurred by hypothesis  $h$ ;  $L(h(\mathbf{x}), y)$  is a particular loss function (e.g., zero–one loss), and the integral runs across the input–output space. We introduce a new notation,  $\phi$ , to denote the probability distribution over  $\mathcal{X} \times \mathcal{Y}$  that indicates which examples are more likely to be seen for that particular task. Since we do not have access to all possible examples in the input–output space, we may choose to approximate the true risk with an empirical risk ( $\hat{R}_\phi(h)$ ). We do this by randomly sampling  $m$  examples according to  $\phi$  to generate a training sample  $T = \{(\mathbf{x}_j, y_j)\}_{j=1}^m$ , where

$$\hat{R}_\phi(h, T) = \frac{1}{m} \sum_{j=1}^m L(h(\mathbf{x}_j), y_j). \quad (12.8)$$

It has been formally shown that one can bound the true risk  $R_\phi(h)$  as a function of the empirical risk  $\hat{R}_\phi(h, T)$  if there exists a uniform bound for all  $h \in \mathcal{H}$  on the probability of deviation between  $R_\phi(h)$  and  $\hat{R}_\phi(h, T)$  (Vapnik, 1995; Blumer et al., 1989). Such bounds can be represented as a function of the Vapnik–Chervonenkis dimension of the hypothesis space  $\mathcal{H}$ ,  $\text{VC}(\mathcal{H})$ . The VC dimension captures the degree of expressiveness or richness in delimiting flexible decision boundaries by the set of functions in  $\mathcal{H}$ ; it provides an objective characterization of  $\mathcal{H}$  (Vapnik, 1995). Bounds for the deviation between  $R_\phi(h)$  and  $\hat{R}_\phi(h, T)$  take on the form

$$R_\phi(h) \leq \hat{R}_\phi(h, T) + g(m, \delta, \text{VC}(\mathcal{H})), \quad (12.9)$$

where the function  $g(\cdot)$  explicitly indicates an upper bound on the distance between the true risk and the empirical risk; the inequality is satisfied for all  $h \in \mathcal{H}$  with probability  $1 - \delta$ .

### 12.4.1 The learning-to-learn scenario

Let us now consider the novelty brought about by the learning-to-learn scenario (Baxter, 2000)). Here we assume the learner is embedded in a set of related tasks that share certain commonalities. In traditional learning, we assume a probability distribution  $\phi$  that indicates which examples are more likely to be seen in such a task. Now we assume there is a metadistribution  $\Phi$  over the space of all possible distributions  $\{\phi\}$ . In essence,  $\Phi$  indicates which tasks are more likely to be found within the sequence of tasks faced by the metalearner (just as  $\phi$  indicates which examples are more likely to be seen in such a task). As an example, if we were interested in classifying luminous objects in astronomical surveys,  $\Phi$  may stand for a probability distribution that peaks over tasks that identify classes of astronomical objects. Given a family of hypothesis spaces  $\mathbb{H}$ , the goal of the metalearner is to find a hypothesis space  $\mathcal{H}^* \in \mathbb{H}$  that minimizes a new functional risk,  $\mathcal{H}^* = \arg \min_{\mathcal{H} \in \mathbb{H}} R_{\Phi}(\mathcal{H})$ , where

$$R_{\Phi}(\mathcal{H}) = \int_{\phi \in \Phi} \inf_{h \in \mathcal{H}} R_{\phi}(h) d\Phi(\phi). \quad (12.10)$$

An expansion of the above formula gives

$$R_{\Phi}(\mathcal{H}) = \int_{\phi \in \Phi} \inf_{h \in \mathcal{H}} \int_{(\mathbf{x}, y) \in \mathcal{X} \times \mathcal{Y}} L(h(\mathbf{x}), y) d\phi(\mathbf{x}, y) d\Phi(\phi). \quad (12.11)$$

The new functional risk,  $R_{\Phi}(\mathcal{H})$ , represents the expected loss of the best possible hypothesis in each hypothesis space. The integral runs across all task distributions  $\{\phi\}$ , which are themselves distributed according to a metadistribution  $\Phi$ . In practice, since we do not know the form of  $\Phi$ , we need to draw samples  $T_1, T_2, \dots, T_n$  to infer how tasks are distributed in our environment.

The advantage of working in a learning-to-learn scenario is that the learner accumulates experience after each new task. Such experience, here referred to as metaknowledge, is expected to result in more accurate models when the tasks share commonalities or patterns. The expectation is that, as more tasks are observed, the number of examples required to attain accurate models (with high probability) decreases over time.

### 12.4.2 Bounds on generalization error for metalearners

Finding bounds on the generalization error for metalearners follows the same logic as the one adopted in conventional learning theory. The idea is to formally show that it is possible to bound the new functional risk  $R_{\Phi}(\mathcal{H})$  as a function of the empirical risk  $\hat{R}_{\Phi}(\mathcal{H})$ . Given a set of  $n$  samples  $\mathbf{T} = \{T_p\}$ , the empirical risk is defined as the average of the best possible empirical error for each training sample  $T_p$ :

$$\hat{R}_{\Phi}(\mathcal{H}) = \frac{1}{n} \sum_{p=1}^n \inf_{h \in \mathcal{H}} \hat{R}_{\phi}(h, T_p). \quad (12.12)$$

The bound can be found if there exists a uniform bound for all  $\mathcal{H} \in \mathbb{H}$  on the probability of deviation between  $R_{\Phi}(\mathcal{H})$  and  $\hat{R}_{\Phi}(\mathcal{H})$ . In conventional learning theory, these bounds are governed by the expressiveness of the family of hypotheses  $\mathcal{H}$ . Similarly, in the learning-to-learn scenario, bounds on the generalization error are governed by the size of the function classes associated with the family space  $\mathbb{H}$ . Specifically, one can guarantee that with probability  $1 - \delta$  (according to the choice of samples  $\mathbf{T}$ ), all  $\mathcal{H} \in \mathbb{H}$  will satisfy the following inequality:

$$R_{\Phi}(\mathcal{H}) \leq \hat{R}_{\Phi}(\mathcal{H}) + \epsilon. \quad (12.13)$$

This holds if the number of tasks  $n$  is such that

$$n \geq \max \left\{ \frac{256}{\epsilon^2} \log \frac{8\mathcal{C}(\frac{\epsilon}{32}, \Lambda_{\mathbb{H}})}{\delta}, \frac{64}{\epsilon^2} \right\} \quad (12.14)$$

and the number of examples  $m$  for each task is such that

$$m \geq \max \left\{ \frac{256}{n\epsilon^2} \log \frac{8\mathcal{C}(\frac{\epsilon}{32}, \Lambda_{\mathbb{H}}^n)}{\delta}, \frac{64}{\epsilon^2} \right\}. \quad (12.15)$$

The theorem (Baxter, 2000) introduces two new properties characterizing the family of hypothesis spaces  $\mathbb{H}$ ,  $\mathcal{C}(\epsilon, \Lambda_{\mathbb{H}})$  and  $\mathcal{C}(\epsilon, \Lambda_{\mathbb{H}}^n)$ . These functions measure the capacity of  $\mathbb{H}$  in a way similar to how the VC dimension measures the capacity of  $\mathcal{H}$ . To provide continuity to our chapter, we defer explanation of these properties to Appendix A. The bounds stated above simply show that, to learn both a good hypothesis space  $\mathcal{H} \in \mathbb{H}$  and a good hypothesis  $h \in \mathcal{H}$ , one needs a minimum number of both the number of tasks and the number of examples on each task. It is known that, if  $\epsilon$  and  $\delta$  are fixed (Baxter, 2000), the number of examples  $m$  needed on each task to attain an accurate model is such that

$$m = O \left( \frac{1}{n} \log \mathcal{C}(\epsilon, \Lambda_{\mathbb{H}}^n) \right). \quad (12.16)$$

This indicates that the required number of examples on each task decreases as the number of tasks increases, in accordance with our expectations of the benefits gained when the learning algorithm has the capability of exploiting previous experience.

### 12.4.3 Other theoretical work

#### Bounds using algorithmic stability

The results described above can be improved if one makes certain assumptions (Maurer, 2005). To understand this, we need to review the concept of algorithmic stability (Bousquet and Elisseeff, 2002). A learning algorithm is said to be uniformly  $\beta$ -stable if taking away one example from the training set does not modify the loss of the output hypothesis by more than  $\beta$  (for a fixed loss function). We update our definition of a metalearning algorithm as a function  $\mathcal{A}(\mathbf{T})$  that outputs a hypothesis after looking at a sequence of samples  $\mathbf{T} = \{T_p\}_{p=1}^n$ . That is, we no longer talk about a hypothesis space but of a single hypothesis that does well on all previous tasks. In that case, one can also think of a metalearning algorithm as being  $\beta'$ -stable if removing one sample from the set of samples  $\mathbf{T}$  does not modify the loss of the output hypothesis by more than  $\beta'$ . Notice that the parameter  $\beta'$  corresponds to the concept of stability across tasks, whereas the parameter  $\beta$  is used to refer to stability across examples drawn from one task.

Given that  $\mathcal{A}(\mathbf{T}) = h$  for a given set of samples  $\mathbf{T}$ , the new results show that, for every environment  $\Phi$ , with probability greater than  $1 - \delta$  according to the selection of  $\mathbf{T}$ , the following inequality holds:

$$\forall \phi \ R_{\Phi}(h) \leq \frac{1}{n} \sum_{p=1}^n \hat{R}_{\phi_p}(h, T_p) + 2\beta' + (4n\beta' + m) \sqrt{\frac{\ln(1/\delta)}{2n}} + 2\beta, \quad (12.17)$$

where  $\phi_p \in \Phi$  and  $\hat{R}_{\phi_p}(h, T_p)$  is an estimation of the empirical loss of hypothesis  $h$  when the examples are drawn from the sample  $T_p$ . The first term on the right-hand side of

the inequality is then the average empirical loss of  $h$  on the set of tasks  $\mathbf{T}$ . It can be shown that the new bound is tighter than that of Section 12.4.2 (of course, under the assumption of stability parameterized by  $\beta$  and  $\beta'$  on  $\mathcal{A}(\mathbf{T}) = h$ ).

### Bounds for domain adaptation

The context of domain adaptation leads to another set of interesting learning bounds (Ben-David et al., 2010). Assume a source domain  $\mathcal{D}_S$  where class labels abound, and a target domain  $\mathcal{D}_T$  with few or no class labels. It is implicitly assumed that the source and target domains must be *related*, with no mechanism to quantify the degree of *relatedness*. This can be helpful to understand how to bound the error of a model trained on the source domain but applied to the target domain, where we assume the distribution over  $\mathcal{X}$  has changed, i.e.,  $P_S(X) \neq P_T(X)$ .

We begin by defining the error of a hypothesis  $h$  under a zero–one loss function as  $R_\phi(h) = E_{(\mathbf{x}, y) \sim \phi} [|h(\mathbf{x}) - y|]$ , where we assume  $\mathcal{Y} = \{-1, 1\}$ . We refer to the source and target distributions as  $\phi_S$  and  $\phi_T$ , with the understanding that the only difference is in the marginal distributions  $P_S(X) \neq P_T(X)$ . It has been formally shown that the generalization error on the target domain can be bound as a function of three terms:

$$R_{\phi_T}(h) \leq R_{\phi_S}(h) + \frac{1}{2}d_{\mathcal{H}\Delta\mathcal{H}}(\phi_S, \phi_T) + \lambda, \quad (12.18)$$

where the first term on the right hand side of the inequality simply refers to the generalization error on the source domain. The second term is a measure of the *relatedness* of the two distributions. Formally,

$$d_{\mathcal{H}\Delta\mathcal{H}}(\phi_S, \phi_T) = 2 \sup_{h, h' \in \mathcal{H}} |P_{\mathbf{x} \sim \phi_S}[h(\mathbf{x}) \neq h'(\mathbf{x})] - P_{\mathbf{x} \sim \phi_T}[h(\mathbf{x}) \neq h'(\mathbf{x})]|. \quad (12.19)$$

The goal is simply to capture the difference in the probability of disagreement between two hypotheses in the space of hypothesis  $\mathcal{H}$ . The last term  $\lambda$  refers to the combined error of an *ideal* hypothesis:

$$\lambda = R_{\phi_S}(h^*) + R_{\phi_T}(h^*), \quad (12.20)$$

where  $h^* = \operatorname{argmin}_{h \in \mathcal{H}} R_{\phi_S}(h) + R_{\phi_T}(h)$ . The bound depends, then, on the *distance* between the source and target distributions, and on the existence of a hypothesis that can attain low generalization error on both the source and target domains.

### 12.4.4 Bias versus variance in metalearning

As part of our theoretical study, we conclude by looking into the nature of the bias–variance dilemma in classification when immersed in a learning-to-learn framework. Let us first recall what the bias–variance dilemma states in traditional learning (Hastie et al., 2009; Geman et al., 1992). The dilemma is based on the fact that the expected prediction error, or risk, can be decomposed into bias and variance components.<sup>1</sup> Ideally we would like to have classifiers with both low bias and low variance but these components are inversely related. On the one hand, simple classifiers encompass a small hypothesis space  $\mathcal{H}$ . Their small repertoire of functions produces high bias (since the hypothesis with lowest prediction error may lie far from the true target function) but low variance (since there are

<sup>1</sup>A third component, the *irreducible error* or *Bayes error*, cannot be eliminated or traded (Hastie et al., 2009).

few hypotheses to choose from). On the other hand, increasing the size of  $\mathcal{H}$  reduces the bias, but increases the variance. The large size of  $\mathcal{H}$  normally allows for flexible decision boundaries (low bias), but the learning algorithm inevitably becomes sensitive to small variations in the data (high variance).

In the learning-to-learn framework, there is an equal need to find a balance in the size of the family of hypothesis spaces  $\mathbb{H}$ . A small  $\mathbb{H}$  will exhibit low variance and high bias; here, unless we can find a good hypothesis space  $\mathcal{H} \in \mathbb{H}$  with a small risk  $R_\Phi(\mathcal{H})$ , the best  $\mathcal{H}$  may be far from the true hypothesis space. And just as in traditional learning, a large  $\mathbb{H}$  will exhibit low bias but high variance, since a large number of available hypothesis spaces increases the chances of selecting one that simply accommodates to the idiosyncrasies of the training data. One main goal is to understand if learning the right family of hypothesis spaces  $\mathbb{H}$  is inherently easier (or not) than learning the right hypothesis space  $\mathcal{H}$ .

## Appendix A

Section 12.4.2 makes use of two properties characterizing the space of a family of hypothesis spaces  $\mathbb{H}$ ,  $\mathcal{C}(\epsilon, \Lambda_{\mathbb{H}})$  and  $\mathcal{C}(\epsilon, \mathbf{\Lambda}_{\mathbb{H}}^n)$ . These functions quantify the capacity of the space of a family of hypothesis spaces  $\mathbb{H}$ . We now explain the nature of these properties in more detail:<sup>2</sup>

**Definition 1.** For each  $\mathcal{H} \in \mathbb{H}$ , define a new function  $\lambda_{\mathcal{H}}(\phi_i)$  by

$$\lambda_{\mathcal{H}}(\phi) = \inf_{h \in \mathcal{H}} R_\Phi(h), \quad (12.21)$$

where  $\lambda : \Phi \rightarrow [0, 1]$ . In other words, the function  $\lambda$  specifies the minimum error loss achieved after looking at every  $h \in \mathcal{H}$  under the distribution  $\phi$ .

**Definition 2.** For the family of hypothesis spaces  $\mathbb{H}$ , define a new set  $\Lambda_{\mathbb{H}}$  by

$$\Lambda_{\mathbb{H}} = \{\lambda_{\mathcal{H}} : \mathcal{H} \in \mathbb{H}\}. \quad (12.22)$$

According to Definition 1, the set  $\Lambda_{\mathbb{H}}$  contains all *different* functions within the space of a family of hypotheses  $\mathbb{H}$ . We can compute the expected difference in the minimum error loss for any two functions  $\lambda_1, \lambda_2 \in \Lambda_{\mathbb{H}}$  as follows:

**Definition 3.** For any two functions  $\lambda_1, \lambda_2 \in \Lambda_{\mathbb{H}}$  and a distribution  $\Phi$  on the space of possible input–output distributions, define

$$D_\Phi(\lambda_1, \lambda_2) = \int_{\phi} |\lambda_1(\phi) - \lambda_2(\phi)| d\Phi(\phi). \quad (12.23)$$

The function  $D$  can be seen as the expected distance between two functions  $\lambda_1, \lambda_2$ . We now define the concept of an  $\epsilon$ -cover as follows:

**Definition 4.** An  $\epsilon$ -cover of  $(\Lambda_{\mathbb{H}}, D_\Phi)$  is a set  $\{\lambda_1, \lambda_2, \dots, \lambda_n\}$  such that, for all  $\lambda \in \Lambda_{\mathbb{H}}$ ,  $D_\Phi(\lambda, \lambda_p) \leq \epsilon$  ( $1 \leq p \leq n$ ). Let  $\mathcal{N}(\epsilon, \Lambda_{\mathbb{H}}, D_\Phi)$  represent the size of the smallest  $\epsilon$ -cover. We now define the capacity of  $\Lambda_{\mathbb{H}}$  by

---

<sup>2</sup>We follow Baxter's work (Baxter, 2000) in different order and notation to simplify the explanation of the two properties characterizing  $\mathbb{H}$ .



$$\mathcal{C}(\epsilon, \Lambda_{\mathbb{H}}) = \sup_{\phi} \mathcal{N}(\epsilon, \Lambda_{\mathbb{H}}, D_{\phi}), \quad (12.24)$$

where the supremum runs over all probability distributions over  $\mathcal{X} \times \mathcal{Y}$ .

We can similarly define the second capacity  $\mathcal{C}(\epsilon, \Lambda_{\mathbb{H}}^n)$ . To begin, consider a sequence of  $n$  tasks that has been modeled with  $n$  hypotheses  $\mathbf{h} = (h_1, h_2, \dots, h_n)$ . We can compute the expected error loss across  $n$  tasks as follows:

$$\lambda_{\mathbf{h}}^n(\{\mathbf{x}, y\}) = \frac{1}{n} \sum_{p=1}^n L(h_p(\mathbf{x}), y). \quad (12.25)$$

**Definition 5.** For the space of a family of hypotheses  $\mathbb{H}$ , define a new set  $\Lambda_{\mathbf{h}}^n$  by

$$\Lambda_{\mathbf{h}}^n = \{\lambda_{\mathbf{h}}^n : h_1, h_2, \dots, h_n \in \mathcal{H}\}. \quad (12.26)$$

The set  $\Lambda_{\mathbf{h}}^n$  is a loss function class and, as before, it indicates how many *different* classes of functions (capturing the average error loss for a sequence of  $n$  hypotheses) are contained within the hypothesis space  $\mathcal{H}$ ; the difference is that now we are comparing sets of  $n$  loss functions.

**Definition 6.** For the space of a family of hypotheses  $\mathbb{H}$ , define

$$\Lambda_{\mathbb{H}}^n = \bigcup_{\mathcal{H} \in \mathbb{H}} \Lambda_{\mathbf{h}}^n, \quad (12.27)$$

where  $\mathbf{h} \subseteq \mathcal{H}$ . The second capacity  $\mathcal{C}(\epsilon, \Lambda_{\mathbb{H}}^n)$  is defined similarly to the first one but using a new distance function:

$$D_{\phi}^n(\mathbf{h}, \mathbf{h}') = \int_{(\mathcal{X} \times \mathcal{Y})^n} |\lambda_{\mathbf{h}}^n(\{\mathbf{x}_i, y_i\}) - \lambda_{\mathbf{h}'}^n(\{\mathbf{x}_i, y_i\})| d\phi_1, d\phi_2, \dots, d\phi_n. \quad (12.28)$$

This brings us to the second capacity function:

$$\mathcal{C}(\epsilon, \Lambda_{\mathbb{H}}^n) = \sup_{\phi} \mathcal{N}(\epsilon, \Lambda_{\mathbb{H}}^n, D_{\phi}^n), \quad (12.29)$$

where the supremum runs over all sequences of  $n$  probability distributions over  $\mathcal{X} \times \mathcal{Y}$ .

## References

- Andrychowicz, M., Denil, M., Colmenarejo, S. G., Hoffman, M. W., Pfau, D., Schaul, T., Shillingford, B., and de Freitas, N. (2016). Learning to learn by gradient descent by gradient descent. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, pages 3988–3996, USA. Curran Associates Inc.
- Argyriou, A., Evgeniou, T., and Pontil, M. (2007). Multi-task feature learning. In *Advances in neural information processing systems 20*, NIPS'07, pages 41–48.
- Bakker, B. and Heskes, T. (2003). Task clustering and gating for Bayesian multitask learning. *Journal of Machine Learning Research*, 4:83–99.
- Basura, F., Habrard, A., Sebban, M., and Tuytelaars, T. (2013). Unsupervised visual domain adaptation using subspace alignment. In *Proceedings of the IEEE International Conference on Computer Vision, ICCV*, pages 2960–2967.

- Baxter, J. (1998). Theoretical models of learning to learn. In Thrun, S. and Pratt, L., editors, *Learning to Learn*, chapter 4, pages 71–94. Springer-Verlag.
- Baxter, J. (2000). A model of inductive learning bias. *Journal of Artificial Intelligence Research*, 12:149–198.
- Ben-David, S., Blitzer, J., Crammer, K., Kulesza, A., Pereira, F., and Vaughan, J. W. (2010). A theory of learning from different domains. *Mach. Learn.*, 79(1-2):151–175.
- Bengio, Y. (2012). Deep learning of representations for unsupervised and transfer learning. In *Proceedings of ICML Workshop on Unsupervised and Transfer Learning*, pages 17–36.
- Bertinetto, L., Henriques, J. F., Torr, P. H. S., and Vedaldi, A. (2019). Meta-learning with differentiable closed-form solvers. In *International Conference on Learning Representations*, ICLR’19.
- Bickel, S., Brückner, M., and Scheffer, T. (2009). Discriminative learning under covariate shift. *J. Mach. Learn. Res.*, 10:2137–2155.
- Blitzer, J., McDonald, R., and Pereira, F. (2006). Domain adaptation with structural correspondence learning. In *Proceedings of the 2006 Conference on Empirical Methods in Natural Language Processing, ACL*, pages 120–128.
- Blumer, A., Haussler, D., and Warmuth, M. K. (1989). Learnability and the Vapnik-Chervonenkis dimension. *Journal of the ACM*, 36(1):929–965.
- Bousquet, O. and Elisseeff, A. (2002). Stability and generalization. *Journal of Machine Learning Research*, 2:499–526.
- Cao, X., Wipf, D., Wen, F., and Duan, G. (2013). A practical transfer learning algorithm for face verification. In *International Conference on Computer Vision (ICCV)*.
- Chopra, S., Hadsell, R., and LeCun, Y. (2005). Learning a similarity metric discriminatively, with application to face verification. In *Proceedings of Computer Vision and Pattern Recognition (CVPR’05) - Volume 1, CVPR ’05*, pages 539–546, Washington, DC, USA. IEEE Computer Society.
- Duda, R. O., Hart, P. E., and Stork, D. G. (2001). *Pattern Classification (2 ed.)*. John Wiley & Sons, New York.
- Evgeniou, T. and Pontil, M. (2004). Regularized multi-task learning. In *Tenth Conference on Knowledge Discovery and Data Mining*.
- Finn, C., Abbeel, P., and Levine, S. (2017). Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning, ICML’17*, pages 1126–1135. JMLR.org.
- Finn, C., Xu, K., and Levine, S. (2018). Probabilistic model-agnostic meta-learning. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems, NIPS’18*, pages 9537–9548, USA. Curran Associates Inc.
- Geman, S., Bienenstock, E., and Doursat, R. (1992). Neural networks and the bias/variance dilemma. *Neural Computation*, pages 1–58.
- Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press.
- Graves, A., Wayne, G., and Danihelka, I. (2014). Neural Turing Machines. *arXiv preprint arXiv:1410.5401*.
- Gretton, A., Smola, A., Huang, J., Schmittfull, M., Borgwardt, K., and Schölkopf, B. (2009). Covariate shift by kernel mean matching. In Quiñero-Candela, J., Sugiyama, M., Schwaighofer, A., and Lawrence, N. D., editors, *Dataset Shift in Machine Learning*, pages 131–160. MIT Press, Cambridge, MA.
- Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, 2nd edition*. Springer.

- Heskes, T. (2000). Empirical Bayes for Learning to Learn. In *Proceedings of the 17th International Conference on Machine Learning, ICML'00*, pages 367–374. Morgan Kaufmann, San Francisco, CA.
- Hochreiter, S., Younger, A. S., and Conwell, P. R. (2001). Learning to learn using gradient descent. In Dorffner, G., Bischof, H., and Hornik, K., editors, *Lecture Notes on Comp. Sci. 2130, Proc. Intl. Conf. on Artificial Neural Networks (ICANN-2001)*, pages 87–94. Springer.
- Kanamori, T., Hido, S., and Sugiyama, M. (2009). A least-squares approach to direct importance estimation. *J. Mach. Learn. Res.*, 10:1391–1445.
- Koch, G., Zemel, R., and Salakhutdinov, R. (2015). Siamese Neural Networks for One-shot Image Recognition. In *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *ICML'15*. JMLR.org.
- Maclaurin, D., Duvenaud, D., and Adams, R. P. (2015). Gradient-based hyperparameter optimization through reversible learning. In *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *ICML'15*, pages 2113–2122.
- Maurer, A. (2005). Algorithmic Stability and Meta-Learning. *Journal of Machine Learning Research*, 6:967–994.
- Munkhdalai, T. and Yu, H. (2017). Meta networks. In Precup, D. and Teh, Y. W., editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *ICML'34*, pages 2554–2563, International Convention Centre, Sydney, Australia. JMLR.org.
- Quionero-Candela, J., Sugiyama, M., Schwaighofer, A., and Lawrence, N. D. (2009). *Dataset shift in machine learning*. The MIT Press.
- Ravi, S. and Larochelle, H. (2017). Optimization as a model for few-shot learning. In *International Conference on Learning Representations, ICLR'17*.
- Rosenstein, M. T., Marx, Z., and Kaelbling, L. P. (2005). To transfer or not to transfer. In *Workshop at NIPS (Neural Information Processing Systems)*.
- Santoro, A., Bartunov, S., Botvinick, M., Wierstra, D., and Lillicrap, T. (2016). Meta-learning with memory-augmented neural networks. In *Proceedings of the 33rd International Conference on Machine Learning, ICML'16*, pages 1842–1850. JMLR.org.
- Shawe-Taylor, J. and Cristianini, N. (2004). *Kernel Methods for Pattern Analysis*. Cambridge University Press.
- Shimodaira, H. (2000). Improving predictive inference under covariate shift by weighting the log-likelihood function. *Journal of Statistical Planning and Inference*, 90(2):227–244.
- Sugiyama, M., Nakajima, S., Kashima, H., Buenau, P. V., and Kawanabe, M. (2008). Direct importance estimation with model selection and its application to covariate shift adaptation. In *Advances in Neural Information Processing Systems 21, NIPS'08*, pages 1433–1440.
- Taylor, M. E. and Stone, P. (2009). Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10:1633–1685.
- Thrun, S. (1998). Lifelong Learning Algorithms. In Thrun, S. and Pratt, L., editors, *Learning to Learn*, pages 181–209. Kluwer Academic Publishers, MA.
- Thrun, S. and Mitchell, T. (1995). Learning One More Thing. In *Proceedings of the International Joint Conference of Artificial Intelligence*, pages 1217–1223.
- Thrun, S. and O'Sullivan, J. (1998). Clustering Learning Tasks and the Selective Cross-Task Transfer of Knowledge. In Thrun, S. and Pratt, L., editors, *Learning to Learn*, pages 235–257. Kluwer Academic Publishers, MA.

- Torrey, L. and Shavlik, J. (2010). Transfer learning. In *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques*, pages 242–264. IGI Global.
- Vapnik, V. (1995). *The Nature of Statistical Learning Theory*. Springer Verlag, New York.
- Vilalta, R. and Drissi, Y. (2002). A perspective view and survey of meta-learning. *Artificial Intelligence Review*, 18(2):77–95.
- Vinyals, O., Blundell, C., Lillicrap, T., Kavukcuoglu, K., and Wierstra, D. (2016). Matching networks for one shot learning. In *Proceedings of the 30th International Conference on Neural Information Processing Systems, NIPS'16*, pages 3637–3645, USA. Curran Associates Inc.
- Weiss, K., Khoshgoftaar, T. M., and Wang, D. (2016). A survey of transfer learning. *Journal of Big Data*, 3(1).
- Yosinski, J., Clune, J., Bengio, Y., and Lipson, H. (2014). How transferable are features in deep neural networks? *arXiv e-prints*, page arXiv:1411.1792.
- Zaheer, M., Kottur, S., Ravanbakhsh, S., Poczos, B., Salakhutdinov, R., and Smola, A. (2017). Deep sets. *arXiv e-prints*, page arXiv:1703.06114.

---

## Metalearning for Deep Neural Networks

Mike Huisman, Jan N. van Rijn, and Aske Plaat

**Summary.** Deep neural networks have enabled large breakthroughs in various domains ranging from image and speech recognition to automated medical diagnosis. However, these networks are notorious for requiring large amounts of data to learn from, limiting their applicability in domains where data is scarce. Through metalearning, the networks can learn how to learn, allowing them to learn from fewer data. In this chapter, we provide a detailed overview of metalearning for knowledge transfer in deep neural networks. We categorize the techniques into (i) metric-based, (ii) model-based, and (iii) optimization-based techniques, cover the key techniques per category, discuss open challenges, and provide directions for future research such as performance evaluation on heterogeneous benchmarks.

### 13.1 Introduction

Although current deep learning methods obtain great successes, training is generally time consuming and large datasets are required to achieve good performance. Metalearning offers a solution to these issues. That is, metalearning deep neural networks can improve their learning ability over time, or equivalently, “learn to learn”: this allows them to learn new concepts from less data.

Most metalearning techniques, in the context of deep learning, learn at two levels. At the *inner level*, the agent is presented with a new task (dataset) and tries to quickly learn the associated concepts. This quick adaptation is facilitated by the knowledge that the agent has accumulated from other tasks, at the *outer level*. Thus, the inner level involves a single task, whereas the outer level involves a multitude of tasks. The accumulated knowledge is often directly embedded into the parameters of the agent (neural network). Note that this is in contrast to some other methods in this book (see, e.g., Chapter 6), where metalearning is used to optimize the hyperparameters of algorithms.

In this chapter, we provide a detailed overview of metalearning for knowledge transfer in deep neural networks, which was briefly introduced in the previous chapter. Following Vinyals (2017), we categorize this field into three groups: (i) metric-, (ii) model-, and (iii) optimization-based techniques. After introducing our notation and providing background information, we summarize key techniques of each category, identify the main challenges, and formulate open questions.

## 13.2 Background and Notation

In this section, we introduce and contrast base-level learning and metalearning in the context of deep learning. Additionally, we briefly discuss common training and evaluation procedures.

### 13.2.1 The meta-abstraction for deep neural networks

In *supervised learning*, we wish to learn a function  $f_\theta : X \rightarrow Y$  that maps inputs  $x_i \in X$  to their corresponding outputs  $y_i \in Y$ . In the context of deep learning,  $f$  is a neural network with parameters  $\theta$ . Note that this is in contrast to previous chapters, where  $\theta$  was used to denote hyperparameters of algorithms. Given a dataset  $D = \{(x_i, y_i)\}_{i=1}^m$  of  $m$  examples, learning boils down to finding parameters  $\theta$  that minimize an empirical loss function  $\mathcal{L}_D$ . This loss function captures how well the model is performing by computing the difference between the model predictions  $\hat{y}_i$  and correct dataset outputs  $y_i$ . In short, we wish to find the optimal parameters

$$\theta^* := \arg \min_{\theta} \mathcal{L}_D(\theta). \quad (13.1)$$

Finding theoretically optimal parameters  $\theta^*$  is often infeasible. We can, however, approximate them, guided by *pre-defined* meta-knowledge  $\omega$  (including, e.g., initial model parameters  $\theta$ , choice of optimizer, and learning rate schedule) (Hospedales et al., 2020). As such, we approximate

$$\theta^* \approx g_\omega(D, \mathcal{L}_D), \quad (13.2)$$

where  $g_\omega$  is an optimization procedure that takes meta-knowledge  $\omega$ , a dataset  $D$ , and loss function  $\mathcal{L}_D$  to produce updated weights  $g_\omega(D, \mathcal{L}_D)$  that presumably perform well on  $D$ . In practice, the optimizer  $g_\omega$  often uses gradients of the loss function to update the model parameters  $\theta$ . To measure the generalization performance of the found parameters  $\theta^*$ , we can split the dataset into training and test sets  $D^{tr}$  and  $D^{test}$ , and use, e.g., cross-validation techniques (see Chapter 3).

In contrast, *supervised metalearning* for deep neural networks does not assume that some meta-knowledge  $\omega$  is given, or pre-defined. Instead, the goal is to find the best  $\omega$  such that new tasks  $\mathcal{T}_j$  (datasets) can be learned as quickly as possible. Metalearning techniques often learn  $\omega$  from a multitude of tasks  $\mathcal{T}_j$ . Note that this is in contrast to regular supervised learning, where only one task (dataset) is used.

More formally, we have a probability distribution of tasks  $p(\mathcal{T})$  and wish to find optimal meta-knowledge

$$\omega^* := \arg \min_{\omega} \underbrace{\mathbb{E}_{\mathcal{T}_j \sim p(\mathcal{T})}}_{\text{Outer level}} \underbrace{[\mathcal{L}_{\mathcal{T}_j}(g_\omega(\mathcal{T}_j, \mathcal{L}_{\mathcal{T}_j}))]}_{\text{Inner level}}. \quad (13.3)$$

Here, the inner level concerns task-specific learning, while the outer level concerns multiple tasks. One can now easily see why this is metalearning: we learn  $\omega$ , which allows for quick learning of tasks  $\mathcal{T}_j$  at the inner level. Hence, we are learning to learn.

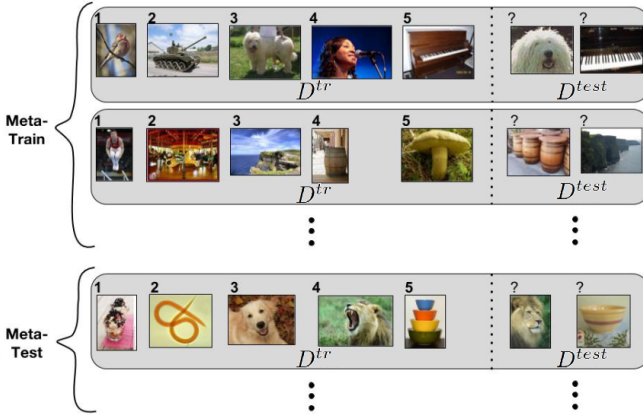


Fig. 13.1: Illustration of  $N$ -way  $k$ -shot classification, where  $N = 5$ , and  $k = 1$ . Meta-validation tasks are not displayed. Adapted from Ravi and Larochelle (2017)

### 13.2.2 Common training and evaluation procedures

In the previous subsection, we described the learning objectives for supervised learning and metalearning for deep neural networks. However, we remained agnostic with respect to the setup used in practice to achieve these objectives. In general, one optimizes a meta-objective by using various tasks. In practice, this is done in three stages: the (i) *meta-train*, (ii) *meta-validation*, and (iii) *meta-test* stages, each of which is associated with a set of tasks from a homogeneous source. Importantly, note that this is in contrast to some of the previous chapters, where heterogeneous data sources were used for knowledge transfer.

Now, in the meta-train stage, the metalearning algorithm is deployed on the meta-train tasks. The meta-validation tasks can then be used to evaluate the performance on unseen tasks which were not used for training. Effectively, this measures the *meta-generalization* ability of the trained network, which serves as feedback to tune, e.g., hyperparameters of the metalearning algorithm. Lastly, the meta-test tasks are used to give a final performance estimate of the metalearning technique.

#### $N$ -way $k$ -shot learning

A frequently used instantiation of this general meta-setup is called  $N$ -way  $k$ -shot classification (see Figure 13.1). This setup is also divided into the three stages — meta-train, meta-validation, and meta-test — which are used for metalearning, metalearner hyperparameter optimization, and evaluation, respectively. Each stage has a corresponding set of disjoint labels, i.e.,  $L^{tr}, L^{val}, L^{test} \subset Y$ , such that  $L^{tr} \cap L^{val} = \emptyset$ ,  $L^{tr} \cap L^{test} = \emptyset$ , and  $L^{val} \cap L^{test} = \emptyset$ . In a given stage  $s$ , *tasks/episodes*  $\mathcal{T}_j = (D_{\mathcal{T}_j}^{tr}, D_{\mathcal{T}_j}^{test})$  are obtained by sampling examples  $(x_i, y_i)$  from the full dataset  $D$ , such that every  $y_i \in L^s$ . Note that this requires access to a dataset  $D$ . Now, the sampling process is guided by the  $N$ -way  $k$ -shot principle, which states that every train dataset  $D_{\mathcal{T}_j}^{tr}$  should contain exactly  $N$  classes and  $k$  examples per class, implying that  $|D_{\mathcal{T}_j}^{tr}| = N \cdot k$ . Furthermore, the true labels of

examples in the test set  $D_{T_j}^{test}$  must be present in the training set  $D_{T_j}^{tr}$  of a given task  $T_j$ .  $D_{T_j}^{tr}$  acts as a *support set*, literally supporting classification decisions on the *query set*  $D_{T_j}^{test}$ . Throughout this chapter, we will use the terms training/support and test/query sets interchangeably. Importantly, note that with this terminology, the test set (or query set) of a task is actually used during the meta-training phase. Furthermore, the fact that the labels across stages are disjoint ensures that we test the ability of a model to learn *new* concepts.

The metalearning objective in the training phase is to minimize the loss function of the model predictions on the query sets, conditioned on the support sets. As such, for a given task, the model “sees” the support set and extracts information from the support set to guide its predictions on the query set. By applying this procedure for different episodes/tasks, with different support and query sets, the model will slowly accumulate meta-knowledge  $\omega$ , which can ultimately speed up learning on new tasks.

At the meta-validation and meta-test stages, or evaluation phases, the learned meta-information in  $\omega$  is fixed. The model is, however, still allowed to make task-specific updates to its parameters  $\theta$  (which implies that it is learning). After task-specific updates, we can evaluate the performance on the test sets. This way we test how well a technique performs at metalearning.

$N$ -way  $k$ -shot classification is often performed for small values of  $k$  (since we want our models to learn new concepts quickly, i.e., from few examples). In that case, one can refer to it as *few-shot learning*.

### 13.2.3 Overview of the rest of this chapter

In the remainder of this chapter we will look in more detail at individual metalearning methods. As indicated before, the methods can be grouped into three main categories (Vinyals, 2017), which we will discuss in sequence:

- (i) Metric-based
- (ii) Model-based
- (iii) Optimization-based techniques

To help give an overview of the methods, we draw your attention to the following tables. Table 13.1 summarizes the three categories and provides the key ideas, strengths, and weaknesses of the approaches. The technical details are explained in the remainder of this chapter. Table 13.2 contains an overview of all the techniques that are discussed further on.

## 13.3 Metric-Based Metalearning

At a high level, the goal of metric-based techniques is to acquire — among others — meta-knowledge  $\omega$  in the form of a good feature space that can be used for various new tasks. In the context of neural networks, this feature space coincides with the weights  $\theta$  of the networks. Then, new tasks can be learned by comparing new inputs with example inputs (of which we know the labels) in the metalearned feature space. The greater the similarity between a new input and an example, the more likely it is that the new input will have the same label as the example input.

Metric-based techniques are a form of metalearning as they leverage their prior learning experience (metalearned feature space) to “learn” new tasks more quickly. Here,



	<b>Metric</b>	<b>Model</b>	<b>Optimization</b>
<b>Key idea</b>	Input similarity	Internal task representation	Bilevel optimization
<b>Predictions</b>	$\sum_{x_i, y_i \in D_{\mathcal{T}_j}^{tr}} k_{\theta}(x, x_i) y_i$	$f_{\theta}(x, D_{\mathcal{T}_j}^{tr})$	$f_{g_{\varphi}(\theta, D_{\mathcal{T}_j}^{tr}, \mathcal{L}_{\mathcal{T}_j})}(x)$
<b>Strength</b>	Simple and effective	Flexible	More robust generalizability
<b>Weakness</b>	Limited to supervised learning	Weak generalization	Computationally expensive

Table 13.1: High-level overview of the categorization of metalearning techniques for deep neural networks. See Section 13.6 for more details. Table extended from Vinyals (2017)

Name	Key idea
<b>Metric-based</b>	<b>Input similarity</b>
Siamese neural networks	Two-input, shared-weight, class identity network
Matching networks	Learn input embeddings for cosine-similarity weighted predictions
Graph neural networks	Propagate label information to unlabeled inputs in a graph
Attentive recurrent comparators	LSTM-based input fusion through interleaved glimpses
<b>Model-based</b>	<b>Internal task representations</b>
Memory-augmented neural networks	External short-term memory module for fast learning
Meta networks	Fast reparameterization of base-learner by distinct metalearner
Simple neural attentive meta-learner	Attention mechanism coupled with temporal convolutions
Conditional neural processes	Condition predictive model on embedded contextual task data
<b>Optimization-based</b>	<b>Bilevel optimization</b>
LSTM optimizer	RNN proposing weight updates for base-learner
Reinforcement learning optimizer	View optimization as reinforcement learning problem
Model-agnostic meta-learning	Learn initialization weights $\theta$ for fast adaptation
Reptile	Move initialization towards task-specific updated weights

Table 13.2: Overview of the metalearning techniques discussed in this chapter

“learn” is used in a non-standard way since metric-based techniques do not make any network changes when presented with new tasks, as they rely solely on input comparisons in the already metalearned feature space. These input comparisons are a form of *non-parametric learning*; i.e., new task information is not absorbed into the network parameters.

More formally, metric-based learning techniques aim to learn a similarity kernel, or equivalently, *attention mechanism*  $k_\theta$  (parameterized by  $\theta$ ), that takes two inputs  $x_1$  and  $x_2$  and outputs their similarity score. Larger scores indicate greater similarity. Class predictions for new inputs  $x$  can then be made by comparing  $x$  with example inputs  $x_i$ , of which we know the true label  $y_i$ , the underlying idea being that the greater the similarity between  $x$  and  $x_i$ , the more likely it becomes that  $x$  also has label  $y_i$ .

Given a task  $\mathcal{T}_j = (D_{\mathcal{T}_j}^{tr}, D_{\mathcal{T}_j}^{test})$  and an unseen input vector  $x \in D_{\mathcal{T}_j}^{test}$ , a probability distribution over classes  $Y$  is computed/predicted as a weighted combination of labels from the support set  $D_{\mathcal{T}_j}^{tr}$ , using the similarity kernel  $k_\theta$ :

$$P_\theta(Y|x, D_{\mathcal{T}_j}^{tr}) = \sum_{(x_i, y_i) \in D_{\mathcal{T}_j}^{tr}} k_\theta(x, x_i) y_i. \tag{13.4}$$

Importantly, the labels  $y_i$  are assumed to be *one-hot encoded*, meaning that they are represented by zero vectors with a “1” at the position of the true class. For example, suppose there are five classes in total, and our example  $x_1$  has true class 4. Then, the one-hot encoded label is  $y_1 = [0, 0, 0, 1, 0]$ . Note that the probability distribution  $P_\theta(Y|x, D_{\mathcal{T}_j}^{tr})$  over classes is a vector of size  $|Y|$ , in which the  $i$ th entry corresponds to the probability that input  $x$  has class  $Y_i$  (given the support set). The predicted class is thus  $\hat{y} = \operatorname{argmax}_{i=1,2,\dots,|Y|} P_\theta(Y|x, S)_i$ , where  $P_\theta(Y|x, S)_i$  is the computed probability that input  $x$  has class  $Y_i$ .

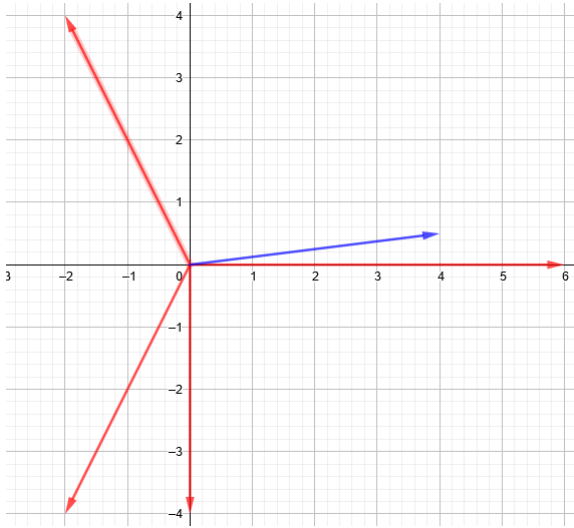


Fig. 13.2: Illustration of our metric-based example

### Example

Suppose that we are given a task  $\mathcal{T}_j = (D_{\mathcal{T}_j}^{tr}, D_{\mathcal{T}_j}^{test})$ . Furthermore, suppose that  $D_{\mathcal{T}_j}^{tr} = \{([0, -4], 1), ([-2, -4], 2), ([-2, 4], 3), ([6, 0], 4)\}$ , where a tuple denotes a pair  $(\mathbf{x}_i, y_i)$ . For simplicity, the example will not use an embedding function, which maps example inputs onto an (more informative) embedding space. Now, our test set only contains one example  $D_{\mathcal{T}_j}^{test} = \{([4, 0.5], y)\}$ . Then, the goal is to predict the correct label for the new input  $[4, 0.5]$  using only examples in  $D_{\mathcal{T}_j}^{tr}$ . The problem is visualized in Figure 13.2, where red vectors correspond to example inputs from our training set. The blue vector is the new input that needs to be classified. Intuitively, this new input is most similar to the vector  $[6, 0]$ , which means that we expect the label for the new input to be the same as that for  $[6, 0]$ , i.e., 4.

Now, suppose we use a fixed similarity kernel, namely the cosine similarity, i.e.,  $k(\mathbf{x}, \mathbf{x}_i) = \frac{\mathbf{x} \cdot \mathbf{x}_i^T}{\|\mathbf{x}\| \cdot \|\mathbf{x}_i\|}$ , where  $\|v\|$  denotes the length of vector  $v$ , i.e.,  $\|v\| = \sqrt{(\sum_n v_n^2)}$ . Here,  $v_n$  denotes the  $n$ th element of placeholder vector  $v$  (substitute  $v$  by  $\mathbf{x}$  or  $\mathbf{x}_i$ ). We can now compute the cosine similarity between the new input  $[4, 0.5]$  and every example input  $\mathbf{x}_i$ , as done in Table 13.3, where we used the facts that  $\|\mathbf{x}\| = \|[4, 0.5]\| = \sqrt{4^2 + 0.5^2} \approx 4.03$ , and  $\frac{\mathbf{x}}{\|\mathbf{x}\|} \approx \frac{[4, 0.5]}{4.03} = [0.99, 0.12]$ .

From this table and Equation 13.4, it follows that the predicted probability distribution  $P_\theta(Y|\mathbf{x}, D_{\mathcal{T}_j}^{tr}) = -0.12y_1 - 0.58y_2 - 0.37y_3 + 0.99y_4 = -0.12[1, 0, 0, 0] - 0.58[0, 1, 0, 0] - 0.37[0, 0, 1, 0] + 0.99[0, 0, 0, 1] = [-0.12, -0.58, -0.37, 0.99]$ . Note that this is not really a probability distribution. That would require normalization such that every element is at least 0 and the sum of all elements is 1. For the sake of this example, we do not perform this normalization, as it is clear that class 4 (the class of the most similar example input  $[6, 0]$ ) will be predicted.

$\mathbf{x}_i$	$y_i$	$\ \mathbf{x}_i\ $	$\frac{\mathbf{x}_i}{\ \mathbf{x}_i\ }$	$\frac{\mathbf{x}_i}{\ \mathbf{x}_i\ } \cdot \frac{\mathbf{x}}{\ \mathbf{x}\ }$
$[0, -4]$	$[1, 0, 0, 0]$	4	$[0, -1]$	-0.12
$[-2, -4]$	$[0, 1, 0, 0]$	4.47	$[-0.48, -0.89]$	-0.58
$[-2, 4]$	$[0, 0, 1, 0]$	4.47	$[-0.48, 0.89]$	-0.37
$[6, 0]$	$[0, 0, 0, 1]$	6	$[1, 0]$	0.99

Table 13.3: Example showing pairwise input comparisons. Numbers were rounded to two decimals

One may wonder why such techniques are metalearners, for we could take any single dataset  $D$  and use pairwise comparisons to compute predictions. Now, at the outer level, metric-based metalearners are trained on a distribution of different tasks, in order to learn (among others) a good input embedding function. This embedding function facilitates inner level learning, which is achieved through pairwise comparisons. As such, one learns an embedding function across tasks to facilitate task-specific learning, which is equivalent to “learning to learn”, or metalearning.

In the remainder of this section we will discuss various key metric-based techniques, including

- Siamese networks (Koch et al., 2015)
- Matching networks Vinyals et al. (2016)

- Graph neural networks (Garcia and Bruna, 2017)
- Attentive recurrent comparators (Shyam et al., 2017)

### 13.3.1 Siamese neural networks

A Siamese neural network (Koch et al., 2015) consists of two neural networks  $f_\theta$  that share the same weights  $\theta$ . Siamese neural networks take two inputs  $x_1, x_2$ , and compute two hidden states  $f_\theta(x_1), f_\theta(x_2)$ , corresponding to the activation patterns in the final hidden layers. These hidden states are fed into a distance layer, which computes a distance vector  $d = |f_\theta(x_1) - f_\theta(x_2)|$ , where  $d_i$  is the absolute distance between the  $i$ th elements of  $f_\theta(x_1)$  and  $f_\theta(x_2)$ . From this distance vector, the similarity between  $x_1, x_2$  is computed as  $\sigma(\alpha^T d)$ , where  $\sigma$  is the sigmoid function (with output range  $[0,1]$ ), and  $\alpha$  is a vector of free weighting parameters, determining the importance of each  $d_i$ . This network structure can be seen in Figure 13.3.

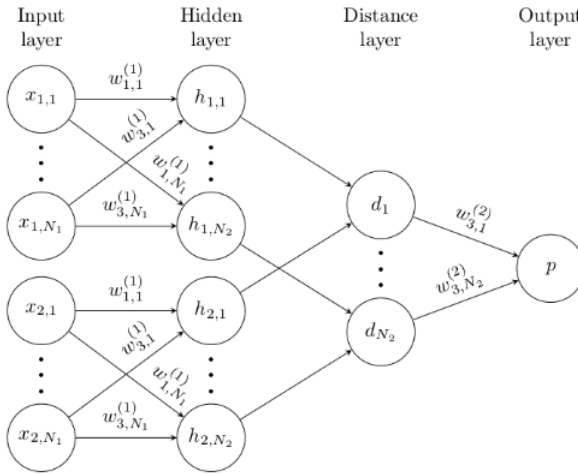


Fig. 13.3: Example of a Siamese neural network. Source: Koch et al. (2015)

Koch et al. (2015) applied this technique to few-shot image recognition in two stages. In the first stage, they train the twin network on an *image verification* task, where the goal is to output whether two input images  $x_1$  and  $x_2$  have the same class. The network is thus stimulated to learn discriminative features. In the second stage, where the model is confronted with a new task, the network leverages its prior learning experience. That is, given a task  $\mathcal{T}_j = (D_{\mathcal{T}_j}^{tr}, D_{\mathcal{T}_j}^{test})$  and previously unseen input  $x \in D_{\mathcal{T}_j}^{test}$ , the predicted class  $\hat{y}$  is equal to the label  $y_i$  of the example  $(x_i, y_i) \in D_{\mathcal{T}_j}^{tr}$  which yields the highest similarity score to  $x$ . In contrast to other techniques mentioned further in this chapter, Siamese neural networks do not optimize directly for good performance across tasks. However, they do leverage learned knowledge from the verification task to learn new tasks quicker.

In summary, Siamese neural networks are a simple and elegant approach to perform few-shot learning. However, they are not readily applicable outside the supervised learning setting.

### 13.3.2 Matching networks

Matching networks (Vinyals et al., 2016) build upon the idea that underlies Siamese neural networks (Koch et al., 2015). That is, they leverage pairwise comparisons between the given support set  $D_{\mathcal{T}_j}^{tr} = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$  (for a task  $\mathcal{T}_j$ ) and new inputs  $\mathbf{x} \in D_{\mathcal{T}_j}^{test}$  from the query/test set which we want to classify. However, instead of assigning the class  $y_i$  of the most similar example input  $\mathbf{x}_i$ , matching networks use a weighted combination of *all* the example labels  $y_i$  in the support set, based on the similarity of inputs  $\mathbf{x}_i$  to new input  $\mathbf{x}$ . More specifically, predictions are computed as follows:  $\hat{y} = \sum_{i=1}^m a(\mathbf{x}, \mathbf{x}_i) y_i$ , where  $a$  is a non-parametric (non-trainable) attention mechanism, or similarity kernel. This classification process is shown in Figure 13.4. In this figure, the input to  $f_\theta$  has to be classified, using the support set  $D_{\mathcal{T}_j}^{tr}$  (input to  $g_\theta$ ).

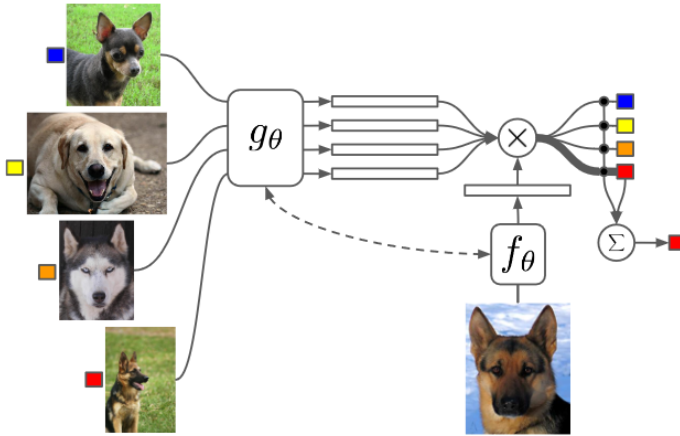


Fig. 13.4: Architecture of matching networks. Source: Vinyals et al. (2016)

The attention that is used consists of a softmax over the cosine similarity  $c$  between the inputs, i.e.,

$$a(\mathbf{x}, \mathbf{x}_i) = \frac{e^{c(f_\phi(\mathbf{x}), g_\varphi(\mathbf{x}_i))}}{\sum_{j=1}^m e^{c(f_\phi(\mathbf{x}), g_\varphi(\mathbf{x}_j))}}, \quad (13.5)$$

where  $f_\phi$  and  $g_\varphi$  are neural networks, parameterized by  $\phi$  and  $\varphi$ , that map raw inputs to a (lower-dimensional) latent vector (corresponding to the activation state in the final hidden layer of a neural network). As such, neural networks act as embedding functions. Now, the larger the cosine similarity between the embeddings of  $\mathbf{x}$  and  $\mathbf{x}_i$ , the larger  $a(\mathbf{x}, \mathbf{x}_i)$  and thus the influence of label  $y_i$  on the predicted label  $\hat{y}$  for input  $\mathbf{x}$ .

Vinyals et al. (2016) propose two main choices for the embedding functions. The first is to use a single neural network, granting us  $\theta = \phi = \varphi$  and thus  $f_\phi = g_\varphi$ . This setup

is the default form of matching networks, as shown in Figure 13.4. The second choice is to make  $f_\phi$  and  $g_\varphi$  dependent on the support set  $D_{\mathcal{T}_j}^{tr}$  using long short-term memory networks (LSTMs). In that case,  $f_\phi$  is represented by an attention LSTM, and  $g_\varphi$  by a bidirectional one. This choice for embedding functions is called *full context embeddings* (FCE), and yielded an accuracy improvement of roughly 2% on miniImageNet compared with the regular matching networks, indicating that task-specific embeddings can aid the classification of new data points from the same distribution.

Matching networks learn a good feature space across tasks for making pairwise comparisons between inputs. In contrast to Siamese neural networks (Koch et al., 2015), this feature space is learned across tasks instead of on a distinct verification task.

In summary, matching networks are an elegant and simple approach to metric-based metalearning. However, these networks are not readily applicable outside of supervised learning settings, and suffer in performance when label distributions are biased (Vinyals et al., 2016).

Some slight variations on matching networks have given rise to new metric-based techniques. Prototypical networks (Snell et al., 2017) use Euclidean distance as the basis for a similarity function, and reduce the required number of pairwise comparisons by comparing the new input  $x$  with class prototypes (mean vectors of inputs  $x_i$  from a class) in the support set. Instead of using fixed similarity metrics, relation networks (Sung et al., 2018) learn a similarity metric represented by a neural network, which allows for greater expressive power.

### 13.3.3 Graph neural networks

Graph neural networks (Garcia and Bruna, 2017) use a more general and flexible approach than previously discussed techniques for  $N$ -way  $k$ -shot classification. As such, graph neural networks subsume Siamese (Koch et al., 2015) and prototypical networks (Snell et al., 2017). The graph neural network approach represents each task as a fully connected graph  $G = (V, E)$ , where  $V$  is a set of nodes/vertices and  $E$  a set of edges connecting nodes. In this graph, nodes  $v_i$  correspond to input embeddings  $f_\theta(x_i)$ , concatenated with their one-hot encoded labels  $y_i$ , i.e.,  $v_i = [f_\theta(x_i), y_i]$ . For inputs  $x$  from the query/test set (for which we do not have the labels), a uniform prior over all  $N$  possible labels is used:  $y = [\frac{1}{N}, \dots, \frac{1}{N}]$ . Thus, each node contains an input and label section. Edges are weighted links that connect these nodes.

The graph neural network then propagates information in the graph using a number of local operators. The underlying idea is that label information can be transmitted from nodes for which we do have the labels to nodes for which we have to predict labels. The used local operators are out of scope for this chapter, and the reader is referred to Garcia and Bruna (2017) for details.

By exposing the graph neural network to various tasks  $\mathcal{T}_j$ , the propagation mechanism can be altered to improve the flow of label information in such a way that predictions become more accurate. As such, in addition to learning a good input representation function  $f_\theta$ , graph neural networks also learn to propagate label information from labeled examples to unlabeled inputs.

Graph neural networks achieve good performance in few-shot settings (Garcia and Bruna, 2017) and are not restricted to supervised learning settings.

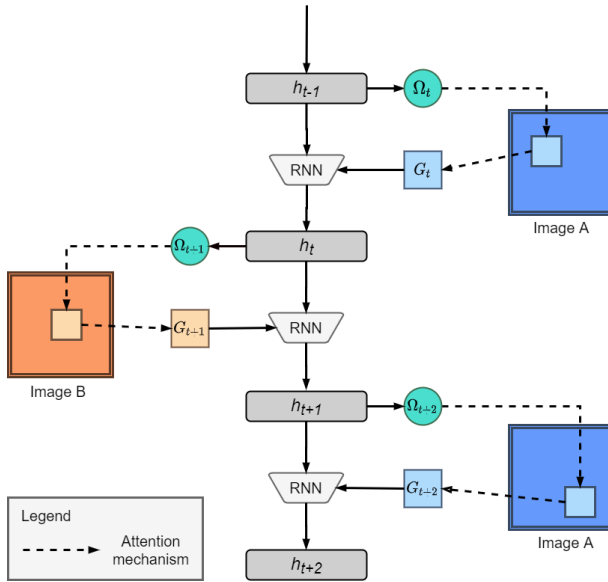


Fig. 13.5: Processing in an attentive recurrent comparator. Source: Shyam et al. (2017)

### 13.3.4 Attentive recurrent comparators

Attentive recurrent comparators (Shyam et al., 2017) differ from previously discussed techniques as they do not compare inputs as a whole, but by parts. This approach is inspired by how humans would make a decision concerning the similarity of objects. That is, we shift our attention from one object to the other, and move back and forth to take glimpses of different parts of both objects. In this way information of two objects is fused from the beginning, whereas other techniques (e.g., matching networks (Vinyals et al., 2016) and graph neural networks (Garcia and Bruna, 2017)) only combine information at the end (after embedding both images) (Shyam et al., 2017).

Given two inputs  $x_i$  and  $x$ , we feed them in interleaved fashion repeatedly into a recurrent neural network (controller):  $x_i, x, \dots, x_i, x$ . Thus, the image at time step  $t$  is given by  $I_t = x_i$  if  $t$  is even, else  $x$ . Then, at each time step  $t$ , the attention mechanism focuses on a square region of the current image:  $G_t = \text{attend}(I_t, \Omega_t)$ , where  $\Omega_t = W_g h_{t-1}$  are attention parameters, which are computed from the previous hidden state  $h_{t-1}$ . The next hidden state  $h_{t+1} = \text{RNN}(G_t, h_{t-1})$  is given by the glimpse at time  $t$ , i.e.,  $G_t$ , and the previous hidden state  $h_{t-1}$ . The entire sequence consists of  $g$  glimpses per image. After this sequence is fed into the recurrent neural network (indicated by  $\text{RNN}(\dots)$ ), the final hidden state  $h_{2g}$  is used as a combined representation of  $x_i$  relative to  $x$ . This process is summarized in Figure 13.5. Classification decisions can then be made by feeding the combined representations into a classifier. Optionally, the combined representations can be processed by bi-directional LSTMs before passing them to the classifier.

The attention approach is biologically inspired and biologically plausible. A downside of attentive recurrent comparators is the higher computational cost, while the per-

formance is often not better than less biologically plausible techniques such as graph neural networks (Garcia and Bruna, 2017).

## Metric-based techniques, in conclusion

In this section, we have seen various metric-based techniques. The metric-based techniques metalearn an informative feature space that can be used to compute class predictions based on input similarity scores.

Key advantages of these techniques are that (i) the underlying idea of similarity-based predictions is conceptually simple, and (ii) they can be fast at test time when tasks are small, as the networks do not need to make task-specific adjustments. However, when tasks at meta-test time become more distant from the tasks that were used at meta-training time, metric-learning techniques are unable to absorb new task information into the network weights. Consequently, performance may suffer.

Furthermore, when tasks become larger, pairwise comparisons may become computationally expensive. Lastly, most metric-based techniques rely on the presence of labeled examples, which makes them inapplicable outside supervised learning settings.

## 13.4 Model-Based Metalearning

A different approach to metalearning for deep neural networks is the model-based approach. On a high level, model-based techniques rely upon an adaptive, internal state, in contrast to metric-based techniques, which generally use a fixed neural network at test time.

More specifically, model-based techniques maintain a stateful, internal representation of a task. When presented with a task, a model-based neural network processes the support/training set in sequential fashion. At every time step, an input enters and alters the internal state of the model. Thus, the internal state can capture relevant task-specific information, which can be used to make predictions for new inputs.

Because the predictions are based on internal dynamics that are hidden from the outside, model-based techniques are also called *black boxes*. Information from previous inputs must be remembered, which is why model-based techniques have a memory component, either in- or externally.

Recall that the mechanics of metric-based techniques were limited to pairwise input comparisons. This is not the case for model-based techniques, where the human designer has the freedom to choose the internal dynamics of the algorithm. As a result, model-based techniques are not restricted to metalearning good feature spaces, as they can also learn internal dynamics, used to process and predict the input data of tasks.

More formally, given a support set  $D_{\mathcal{T}_j}^{tr}$  corresponding to task  $\mathcal{T}_j$ , model-based techniques compute a class probability distribution for a new input  $x$  as follows:

$$P_{\theta}(Y|x, D_{\mathcal{T}_j}^{tr}) = f_{\theta}(x, D_{\mathcal{T}_j}^{tr}), \quad (13.6)$$

where  $f$  represents the black-box neural network model and  $\theta$  its parameters.

### Example

Using the same example as in Section 13.3, suppose we are given a task training set  $D_{\mathcal{T}_j}^{tr} = \{([0, -4], 1), ([-2, -4], 2), ([-2, 4], 3), ([6, 0], 4)\}$ , where a tuple denotes a



pair  $(x_i, y_i)$ . Furthermore, suppose our test set only contains one example  $D_{\mathcal{T}_j}^{test} = \{([4, 0.5], 4)\}$ . This problem has been visualized in Figure 13.2 (in Section 13.3). Now, for the sake of the example, we do not use an input embedding function: our model will operate on the raw inputs of  $D_{\mathcal{T}_j}^{tr}$  and  $D_{\mathcal{T}_j}^{test}$ . As an internal state, our model uses an external *memory matrix*  $M \in \mathbb{R}^{4 \times (2+1)}$ , with four rows (one for each example in our support set) and three columns (the dimensionality of input vectors, plus one dimension for the correct label). Our model proceeds to process the support set in sequential fashion, reading the examples from  $D_{\mathcal{T}_j}^{tr}$  one by one, and by storing the  $i$ th example in the  $i$ th row of the memory module. After processing the support set, the memory matrix contains all examples and, as such, serves as an internal task representation.

Now, given the new input  $[4, 0.5]$ , our model could use many different techniques to make a prediction based on this representation. For simplicity, assume that it computes the dot product between  $x$  and every memory  $M(i)$  (the 2-D vector in the  $i$ th row of  $M$ , ignoring the correct label) and predicts the class of the input which yields the largest dot product. This would produce scores of  $-2$ ,  $-10$ ,  $-6$ , and  $24$  for the examples in  $D_{\mathcal{T}_j}^{tr}$ , respectively. Since the last example  $[6, 0]$  yields the largest dot product, we predict that class, i.e., 4.

This example was a bit simplistic for illustrative purposes. More advanced and successful techniques have been proposed, including

- Memory-augmented neural networks (Santoro et al., 2016)
- Meta networks (Munkhdalai and Yu, 2017)
- Simple neural attentive meta-learner (SNAIL) (Mishra et al., 2018)
- Conditional neural processes (Garnelo et al., 2018)

We discuss these techniques in order.

### 13.4.1 Memory-augmented neural networks

The key idea of memory-augmented neural networks (Santoro et al., 2016) is to enable neural networks to learn quickly with the help of an *external memory*. The main *controller* (the recurrent neural network interacting with the memory) then gradually accumulates knowledge across tasks, while the external memory allows for quick task-specific adaptation. For this, Santoro et al. (2016) used neural Turing machines (Graves et al., 2014). Here, the controller is parameterized by  $\theta$  and acts as the long-term memory of the memory-augmented neural network, while the external memory module is the short-term memory.

The workflow of memory-augmented neural networks is displayed in Figure 13.6. Note that the data from a task is processed as a sequence; i.e., data are fed into the network one by one. The support/training set is fed into the memory-augmented neural network first. Afterwards, the query/test set is processed. At time step  $t$ , the model receives input  $x_t$  with the label of the previous input, i.e.,  $y_{t-1}$ . This was done to prevent the network from mapping class labels directly to the output (Santoro et al., 2016).

The interaction between the controller and memory is visualized in Figure 13.7. Given an input  $x_t$  at time  $t$ , the controller generates a key  $k_t$ , which is used to either retrieve or store a memory from/in memory matrix  $M$ , i.e.,  $M_t(i)$ . To read from memory using a key  $k_t$ , a (column) read vector  $w_t^r$  is created with the same number of rows as the memory matrix  $M$ , where each entry  $i$  denotes the cosine similarity between the key  $k_t$ , and the memory stored in row  $i$ , i.e.,  $M_t(i)$ . Then, the memory  $r_t = \sum_i w_t^r(i)M(i)$  is retrieved, which is simply a linear combination of rows/memories in the memory matrix  $M$ .

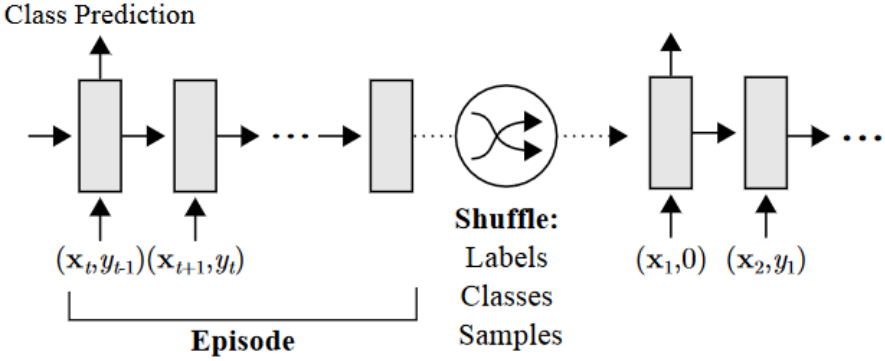


Fig. 13.6: Workflow of memory-augmented neural networks. Source: Santoro et al. (2016)

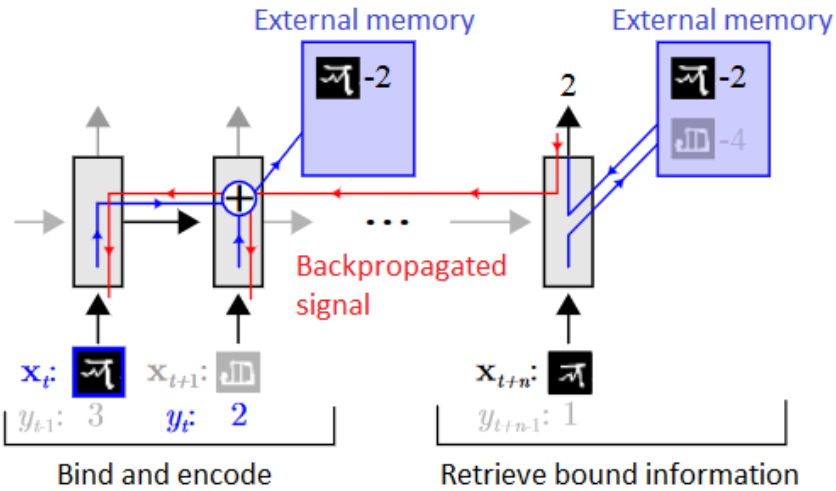


Fig. 13.7: Controller-memory interaction in memory-augmented neural networks. Source: Santoro et al. (2016)

To write to memory, Santoro et al. (2016) propose a new mechanism called least recently used access (LRUA). LRUA writes to either the least, or most recently used memory location. In the former case, it preserves recent memories, and in the latter it updates recently obtained information. The writing mechanism works by keeping track of how often every memory location is accessed in a usage vector  $w_t^u$ , which is updated at every time step according to the following update rule:  $w_t^u := \gamma w_{t-1}^u + w_t^r + w_t^w$ , where the superscripts  $u$ ,  $w$ , and  $r$  refer to usage, write, and read vectors, respectively. In words, the previous usage vector is decayed (using parameter  $\gamma$ ), while current reads

$(w_t^r)$  and writes  $(w_t^w)$  are added to the usage. Now, let  $n$  be the total number of reads to memory, and  $\ell u(n)$  ( $\ell u$  for 'least used') be the  $n$ th smallest value in the usage vector  $w_t^u$ . Then, the least-used weights are defined as follows:

$$w_t^{\ell u}(i) = \begin{cases} 0 & \text{if } w_t^u(i) > \ell u(n) \\ 1 & \text{else} \end{cases}.$$

Then, the write vector  $w_t^w$  is computed as  $w_t^w = \sigma(\alpha)w_{t-1}^r + (1 - \sigma(\alpha))w_{t-1}^{\ell u}$ , where  $\alpha$  is a parameter that interpolates between the two weight vectors. As such, if  $\sigma(\alpha) = 1$ , we write to the most recently used memory, whereas when  $\sigma(\alpha) = 0$ , we write to the least recently used memory locations. Finally, writing is performed as follows:  $M_t(i) := M_{t-1}(i) + w_t^w(i)k_t$ , for all  $i$ .

Predictions are made as follows: given an input  $x_t$ , memory-augmented neural networks compute the corresponding memory  $r_t$  and feed that memory into a classifier. Across tasks, memory-augmented neural networks learn a good input embedding function  $f_\theta$  and classifier weights, which can be exploited when presented with new tasks.

In summary, memory-augmented neural networks (Santoro et al., 2016) combine an external memory and a neural network to achieve metalearning. The interaction between a controller, with long-term memory parameters  $\theta$ , and the memory  $M$  may also be interesting for studying human metalearning (Santoro et al., 2016). In contrast to many metric-based techniques, this model-based technique is applicable to both classification and regression problems. A downside of this approach is the architectural complexity.

### 13.4.2 Meta networks

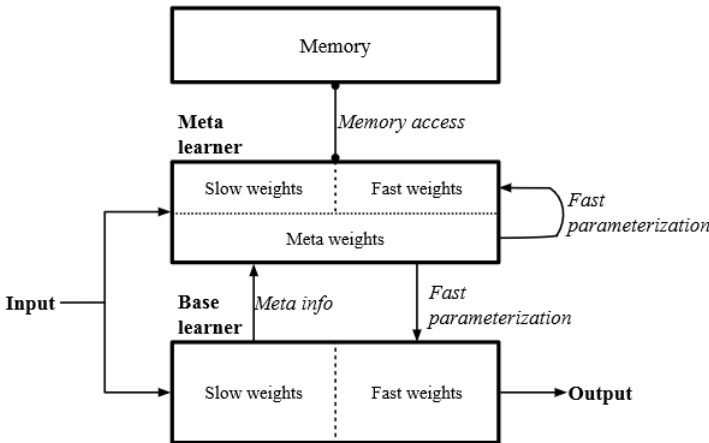


Fig. 13.8: Architecture of a meta network. Source: Munkhdalai and Yu (2017)

In a similar fashion to memory-augmented neural networks (Santoro et al., 2016), meta networks (Munkhdalai and Yu, 2017) also leverage the idea of an external mem-

ory module. However, meta networks use the memory for a different purpose. Meta networks are divided into two distinct subsystems (consisting of neural networks), i.e., the base- and metalearner (whereas in memory-augmented neural networks the base- and meta-components are intertwined). Now, the base-learner is responsible for performing tasks, and for providing the metalearner with meta-information, such as loss gradients. The metalearner can then compute fast task-specific weights for itself and the base-learner, such that it can perform better on the given task  $\mathcal{T}_j = (D_{\mathcal{T}_j}^{tr}, D_{\mathcal{T}_j}^{test})$ . This workflow is depicted in Figure 13.8.

The metalearner consists of neural networks  $u_\phi$ ,  $m_\varphi$ , and  $d_\psi$ . The network  $u_\phi$  is used as an input representation function. The networks  $d_\psi$  and  $m_\varphi$  are used to compute task-specific weights  $\phi^*$  and example-level fast weights  $\theta^*$ . Lastly,  $b_\theta$  is the base-learner which performs input predictions. Note that we used the term “fast weights” throughout, which refers to task- or input-specific versions of slow (initial) weights.

```

1: Sample  $S = \{(x_i, y_i) \sim D_{\mathcal{T}_j}^{tr}\}_{i=1}^T$  from the support set
2: for  $(x_i, y_i) \in S$  do
3:    $\mathcal{L}_i = \text{error}(u_\phi(x_i), y_i)$ 
4: end for
5:  $\phi^* = d_\psi(\{\nabla_\phi \mathcal{L}_i\}_{i=1}^T)$ 
6: for  $(x_i, y_i) \in D_{\mathcal{T}_j}^{tr}$  do
7:    $\mathcal{L}_i = \text{error}(b_\theta(x_i), y_i)$ 
8:    $\theta_i^* = m_\varphi(\nabla_\theta \mathcal{L}_i)$ 
9:   Store  $\theta_i^*$  in  $i$ th position of example-level weight memory  $M$ 
10:   $r_i = u_{\phi, \phi^*}(x_i)$ 
11:  Store  $r_i$  in  $i$ th position of representation memory  $R$ 
12: end for
13:  $\mathcal{L}_{task} = 0$ 
14: for  $(x, y) \in D_{\mathcal{T}_j}^{test}$  do
15:   $r = u_{\phi, \phi^*}(x)$ 
16:   $a = \text{attention}(R, r)$   $\{a_k$  is the cosine similarity between  $r$  and  $R(k)\}$ 
17:   $\theta^* = \text{softmax}(a)^T M$ 
18:   $\mathcal{L}_{task} = \mathcal{L}_{task} + \text{error}(b_{\theta, \theta^*}(x), y)$ 
19: end for
20: Update  $\Theta = \{\theta, \phi, \psi, \varphi\}$  using  $\nabla_\Theta \mathcal{L}_{task}$ 

```

**Algorithm 13.1:** Meta networks, by Munkhdalai and Yu (2017)

The pseudocode for meta networks is displayed in Algorithm 13.1. First, a sample of the support set is created (line 1), which is then used to compute task-specific weights  $\phi^*$  for the representation network  $u$  (lines 2–5).

Then, meta networks iterate over every example  $(x_i, y_i)$  in the support set  $D_{\mathcal{T}_j}^{tr}$ . The base-learner  $b_\theta$  attempts to make class predictions for these examples, resulting in loss values  $\mathcal{L}_i$  (lines 7–8). The gradients of these losses are used to compute fast weights  $\theta^*$  for example  $i$  (line 8), which are then stored in the  $i$ th row of memory matrix  $M$  (line 9). Additionally, input representations  $r_i$  are computed and stored in the memory matrix  $R$  (lines 10–11).

Now, meta networks are ready to address the query set  $D_{\mathcal{T}_j}^{test}$ . They iterate over every example  $(x, y)$  and compute a representation  $r$  of it (line 15). This representation is matched against the representations of the support set, which are stored in the memory

matrix  $R$ . This matching gives us a similarity vector  $\alpha$ , where every entry  $k$  denotes the similarity between the input representation  $r$  and the  $k$ th row in the memory matrix  $R$ , i.e.,  $R(k)$  (line 16). A softmax over this similarity vector is performed to normalize the entries. The resulting vector is used to compute a linear combination of weights that were generated for inputs in the support set (line 17). These weights  $\theta^*$  are specific for the input  $x$  in the query set, and can be used by the base-learner  $b$  to make predictions for that input (line 18). The observed error is added to the task loss. After the entire query set is processed, all the involved parameters can be updated using backpropagation (line 20).

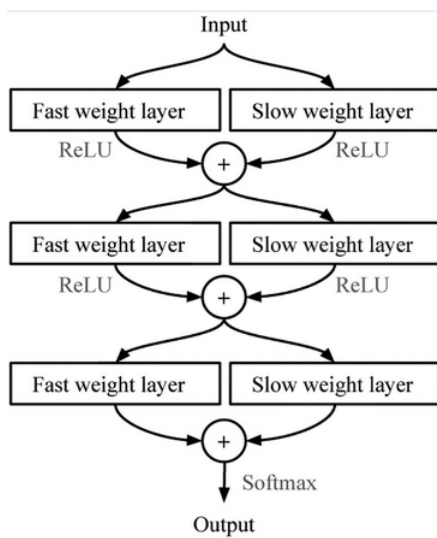


Fig. 13.9: Layer augmentation setup used to combine slow and fast weights. Source: Munkhdalai and Yu (2017)

Note that some neural networks use both slow and fast weights at the same time. Munkhdalai and Yu (2017) use an augmentation setup for this, as depicted in Figure 13.9.

In short, meta networks rely on a reparameterization of the meta- and base-learner for every task. Despite the flexibility and applicability to both supervised and reinforcement learning settings, the approach is quite complex. It consists of many components, each with its own set of parameters, which can be a burden on memory usage and computation time. Additionally, finding the correct architecture for all the involved components can be time consuming.

### 13.4.3 Simple neural attentive meta-learner (SNAIL)

Instead of an external memory matrix, SNAIL (Mishra et al., 2018) relies on a special model architecture to serve as the memory. Mishra et al. (2018) argue that it is not possible to use recurrent neural networks for this, as they have limited memory capacity and cannot pinpoint specific prior experiences (Mishra et al., 2018). Hence, SNAIL uses

a different architecture, consisting of 1D *temporal convolutions* (Oord et al., 2016) and a *soft attention* mechanism (Vaswani et al., 2017). The temporal convolutions allow for ‘high-bandwidth’ memory access, and the attention mechanism allows us to pinpoint specific experiences. Figure 13.10 visualizes the architecture and workflow of SNAIL for supervised learning problems. From this figure, it becomes clear why this technique is model based. That is, model outputs are based upon the internal state, computed from earlier inputs.

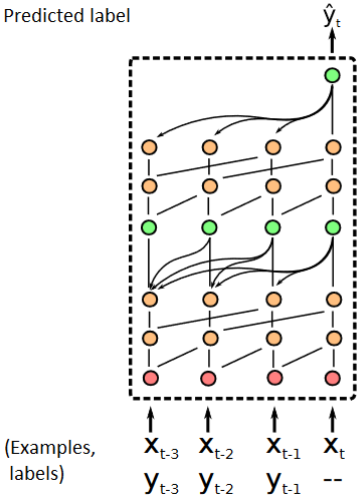


Fig. 13.10: Architecture and workflow of SNAIL. Temporal convolution blocks are orange; attention blocks are green. Source: Mishra et al. (2018)

SNAIL consists of three building blocks. The first is the *DenseBlock*, which applies a single 1D convolution to the input and concatenates (in the feature/horizontal direction) the result. The second is a *TCBlock*, which is simply a series of *DenseBlocks* with exponentially increasing dilation rate of the temporal convolutions (Mishra et al., 2018). Note that the dilation is nothing but the temporal distance between two nodes in a network. For example, if we use a dilation of 2, a node at position  $p$  in layer  $L$  will receive the activation from node  $p - 2$  from layer  $L - 1$ . The third block is the *AttentionBlock*, which learns to focus on the important parts of prior experience.

In a similar fashion to memory-augmented neural networks (Santoro et al., 2016) (Subsection 13.4.1), SNAIL also processes task data in sequence, as shown in Figure 13.10. However, the input at time  $t$  is accompanied with the label at time  $t$ , instead of  $t - 1$  (as was the case for memory-augmented neural networks). SNAIL learns internal dynamics from seeing various tasks, so that it can make good predictions on the query set, conditioned upon the support set.

A key advantage of SNAIL is that it can be applied to both supervised and reinforcement learning tasks. In addition, it achieves good performance compared with previously discussed techniques. A downside of SNAIL is that finding the correct architecture of TCBlocks and DenseBlocks can be time consuming.

### 13.4.4 Conditional neural processes

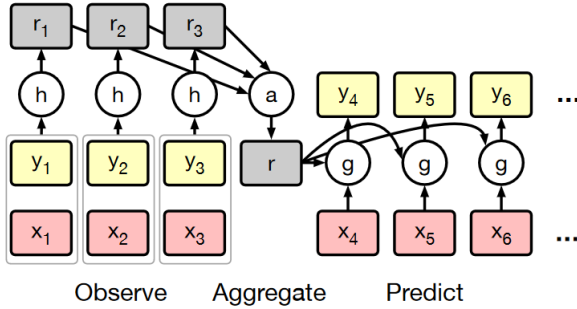


Fig. 13.11: Conditional neural process. Source: Garnelo et al. (2018)

In contrast to previous techniques, a conditional neural process (CNP) (Garnelo et al., 2018) does not rely on an external memory module. Instead, it aggregates the support set into a single aggregated latent representation. The general architecture is shown in Figure 13.11. As we can see, the conditional neural process operates in three phases on task  $\mathcal{T}_j$ . First, it observes the training set  $D_{\mathcal{T}_j}^{tr}$ , including the ground-truth outputs  $y_i$ . Examples  $(x_i, y_i) \in D_{\mathcal{T}_j}^{tr}$  are embedded using a neural network  $h_\theta$  into representations  $r_i$ . Second, these representations are aggregated using the operator  $a$  to produce a single representation  $r$  of  $D_{\mathcal{T}_j}^{tr}$  (hence it is model based). Third, a neural network  $g_\phi$  processes this single representation  $r$  and the new inputs  $x$  and produces the predictions  $\hat{y}$ .

Let the entire conditional neural process model be denoted by  $Q_\Theta$ , where  $\Theta$  is a set of all the involved parameters  $\{\theta, \phi\}$ . The training process is different compared with other techniques. Let  $\mathbf{x}_{\mathcal{T}_j}$  and  $\mathbf{y}_{\mathcal{T}_j}$  denote all inputs and corresponding outputs in  $D_{\mathcal{T}_j}^{tr}$ . Then, the first  $\ell \sim U(0, \dots, k \cdot N - 1)$  examples in  $D_{\mathcal{T}_j}^{tr}$  are used as a conditioning set  $D_{\mathcal{T}_j}^c$  (effectively splitting the training set in a true training set and a validation set). Given a value of  $\ell$ , the goal is to maximize the log likelihood (or minimize the negative log likelihood) of the labels  $\mathbf{y}_{\mathcal{T}_j}$  in the entire training set  $D_{\mathcal{T}_j}^{tr}$ :

$$\mathcal{L}(\Theta) = -\mathbb{E}_{\mathcal{T}_j \sim p(\mathcal{T})} \left[ \mathbb{E}_{\ell \sim U(0, \dots, k \cdot N - 1)} \left( Q_\Theta(\mathbf{y}_{\mathcal{T}_j} | D_{\mathcal{T}_j}^c, \mathbf{x}_{\mathcal{T}_j}) \right) \right]. \quad (13.7)$$

Conditional neural processes are trained by repeatedly sampling various tasks and values of  $\ell$ , and propagating the observed loss backwards.

In summary, conditional neural processes use compact representations of previously seen inputs to aid the classification of new observations. Despite its simplicity and elegance, a disadvantage of this technique is that it is often outperformed in few-shot settings by other techniques such as matching networks (Vinyals et al., 2016) (see Subsection 13.3.2).

### Model-based techniques, in conclusion

In this section, we have discussed various model-based techniques. Despite apparent differences, they all build on the notion of task internalization; that is, tasks are processed

and represented in the state of the model-based system. This state can then be used to make predictions.

Advantages of model-based approaches include the flexibility of the internal dynamics of the systems and their broader applicability compared with most metric-based techniques. However, model-based techniques are often outperformed by metric-based techniques in supervised settings (e.g., graph neural networks (Garcia and Bruna, 2017); Subsection 13.3.3), may not perform well when presented with larger datasets (Hospedales et al., 2020), and generalize less well to more distant tasks than optimization-based techniques (Finn and Levine, 2018). We discuss this optimization-based approach next.

## 13.5 Optimization-Based Metalearning

Optimization-based techniques adopt a different perspective on metalearning than the previous two approaches. They explicitly optimize for fast learning. Most optimization-based techniques do so by approaching metalearning as a bilevel optimization problem. At the inner level, a base-learner makes task-specific updates using some optimization strategy (such as gradient descent). At the outer level, the performance across tasks is optimized.

More formally, given a task  $\mathcal{T}_j = (D_{\mathcal{T}_j}^{tr}, D_{\mathcal{T}_j}^{test})$  with new a input  $\mathbf{x} \in D_{\mathcal{T}_j}^{test}$  and base-learner parameters  $\theta$ , optimization-based metalearners return

$$P(Y|\mathbf{x}, D_{\mathcal{T}_j}^{tr}) = f_{g_\varphi(\theta, D_{\mathcal{T}_j}^{tr}, \mathcal{L}_{\mathcal{T}_j})}(\mathbf{x}), \quad (13.8)$$

where  $f$  is the base-learner and  $g_\varphi$  is a (learned) optimizer that makes task-specific updates to the base-learner parameters  $\theta$  using the training data  $D_{\mathcal{T}_i}^{tr}$  and loss function  $\mathcal{L}_{\mathcal{T}_j}$ .

### Example

Suppose we are faced with a linear regression problem, where every task is associated with a different function  $f(x)$ . For this example, suppose our model only has two parameters:  $a$  and  $b$ , which together form the function  $\hat{f}(x) = ax + b$ . Suppose further that our meta-training set consists of four different tasks, i.e., A, B, C, and D. Then, according to the optimization-based view, we wish to find a single set of parameters  $\{a, b\}$  from which we can quickly learn the optimal parameters for each of the four tasks, as displayed in Figure 13.12. In fact, this is the intuition behind the popular optimization-based technique MAML (Finn et al., 2017).

We now discuss the following optimization-based techniques, in turn:

- LSTM optimizer (Andrychowicz et al., 2016)
- Reinforcement learning optimizer (Ravi and Larochelle, 2017)
- Model-agnostic meta-learning (MAML) (Finn et al., 2017)
- Reptile (Nichol and Schulman, 2018)

### 13.5.1 LSTM optimizer

Standard gradient update rules have the form



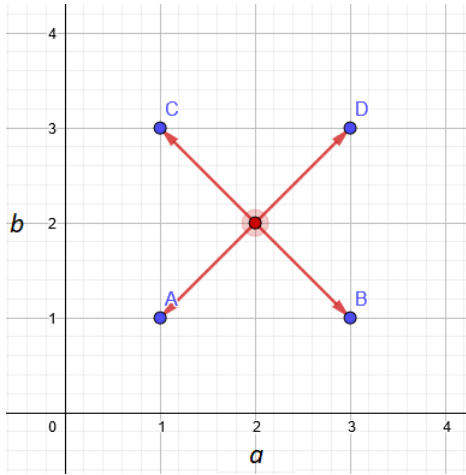


Fig. 13.12: Example of an optimization-based technique, inspired by Finn et al. (2017)

$$\theta_{t+1} := \theta_t - \alpha \nabla_{\theta_t} \mathcal{L}_{\mathcal{T}_j}(\theta_t), \tag{13.9}$$

where  $\alpha$  is the learning rate and  $\mathcal{L}_{\mathcal{T}_j}(\theta_t)$  is the loss function with respect to the task  $\mathcal{T}_j$  and network parameters at time  $t$ , i.e.,  $\theta_t$ . The key idea underlying LSTM optimizers (Andrychowicz et al., 2016) is to replace the update term  $(-\alpha \nabla \mathcal{L}_{\mathcal{T}_j}(\theta_t))$  by an update proposed by an LSTM  $g$  with parameters  $\varphi$ . Then, the new update becomes

$$\theta_{t+1} := \theta_t + g_{\varphi}(\nabla_{\theta_t} \mathcal{L}_{\mathcal{T}_j}(\theta_t)). \tag{13.10}$$

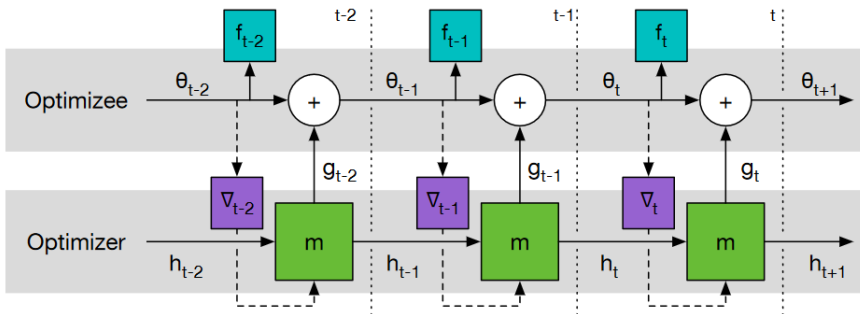


Fig. 13.13: Workflow of the LSTM optimizer. Gradients can only propagate backwards through solid edges.  $f_t$  denotes the observed loss at time step  $t$ . Source: Andrychowicz et al. (2016)

This new update allows the optimization strategy to be tailored to a specific family of tasks. Note that this is metalearning; i.e., the LSTM learns to learn.

The loss function used to train an LSTM optimizer is

$$\mathcal{L}(\varphi) = \mathbb{E}_{\mathcal{L}_{\mathcal{T}_j}} \left[ \sum_{t=1}^T w_t \mathcal{L}_{\mathcal{T}_j}(\theta_t) \right], \quad (13.11)$$

where  $T$  is the number of parameter updates made, and  $w_t$  are weights larger than zero (set to the constant 1 in the original paper (Andrychowicz et al., 2016)). As is often done, second-order derivatives (arising from the dependency between the updated weights and the LSTM optimizer) were ignored due to the computational expense associated with the computation thereof. This loss function is fully differentiable and thus allows for training an LSTM optimizer (see Figure 13.13). To prevent a parameter explosion, the same network is used for every *coordinate/weight* in the base-learner’s network, causing the update rule to be the same for every parameter. Of course, the updates depend on their prior values and gradients.

The key advantage of LSTM optimizers is that they can enable faster learning compared with hand-crafted optimizers, also on different datasets than those used to train the optimizer. However, Andrychowicz et al. (2016) did not apply this technique to few-shot learning. In fact, they did not apply it across tasks at all. Thus, it is unclear whether this technique can perform well in few-shot settings, where few data per class are available for training. Furthermore, the question remains of whether it can scale to larger base-learner architectures.

### 13.5.2 Reinforcement learning optimizer

Li and Malik (2018) proposed a framework which casts optimization as a reinforcement learning problem. Optimization can then be performed by existing reinforcement learning techniques. Now, at a high level, an optimization algorithm  $g$  takes as input an initial set of weights  $\theta_0$  and a task  $\mathcal{T}_j$  with a corresponding loss function  $\mathcal{L}_{\mathcal{T}_j}$  and produces a sequence of new weights  $\theta_1, \dots, \theta_T$ , where  $\theta_T$  is the final solution found. On this sequence of proposed new weights, we can define a loss function  $\mathcal{L}$  that captures unwanted properties (e.g., slow convergence, oscillations, etc.). The goal of learning an optimizer can then be formulated more precisely as follows: we wish to learn an optimal optimizer  $g^*$ :

$$g^* = \operatorname{argmin}_g \mathbb{E}_{\mathcal{T}_j \sim p(\mathcal{T}), \theta_0 \sim p(\theta_0)} [\mathcal{L}(g(\mathcal{L}_{\mathcal{T}_j}, \theta_0))]. \quad (13.12)$$

The key insight is that the optimization can be formulated as a partially observable Markov decision process (POMDP). Then, the state corresponds to the current set of weights  $\theta_t$ , the action to the proposed update at time step  $t$ , i.e.,  $\Delta\theta_t$ , and the policy to the function that computes the update. With this formulation, the optimizer  $g$  can be learned by existing reinforcement learning techniques. In their paper, they used a recurrent neural network as the optimizer. At each time step, they feed it observation features, which depend on the previous set of weights, loss gradients, and objective functions, and use guided policy search to train it.

In summary, Li and Malik (2018) made a first step towards general optimization through reinforcement learning optimizers, which were shown to be able to generalize across network architectures and datasets. However, the base-learner architecture that was used was quite small. The question remains of whether this approach can scale to larger architectures.

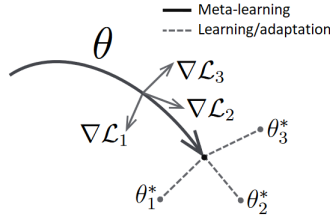


Fig. 13.14: MAML learns an initialization point from which it can perform well on various tasks. Source: Finn et al. (2017)

### 13.5.3 Model-agnostic meta-learning (MAML)

Model-agnostic meta-learning (MAML) (Finn et al., 2017) uses a simple gradient-based inner optimization procedure (e.g., stochastic gradient descent) instead of more complex LSTM procedures or procedures based on reinforcement learning. The key idea of MAML is to explicitly optimize for fast adaptation to new tasks by learning a good set of initialization parameters  $\theta$ . This is shown in Figure 13.14: from the learned initialization  $\theta$ , we can quickly move to the best set of parameters for task  $\mathcal{T}_j$ , i.e.,  $\theta_j^*$  for  $j = 1, 2, 3$ . The learned initialization can be seen as the *inductive bias* of the model, or simply the set of assumptions (encapsulated in  $\theta$ ) that the model makes with respect to the overall task structure.

More formally, let  $\theta$  denote the initial model parameters of a model. The goal is to quickly learn new concepts, which is equivalent to achieving a minimal loss in few gradient update steps. The amount of gradient steps  $s$  has to be specified upfront, such that MAML can explicitly optimize for achieving good performance within that number of steps. Suppose we pick only one gradient update step, i.e.,  $s = 1$ . Then, given a task  $\mathcal{T}_j = (D_{\mathcal{T}_j}^{tr}, D_{\mathcal{T}_j}^{test})$ , gradient descent would produce updated parameters (i.e., fast weights)

$$\theta'_j = \theta - \alpha \nabla_{\theta} \mathcal{L}_{D_{\mathcal{T}_j}^{tr}}(\theta), \tag{13.13}$$

specific to task  $i$ . The *meta-loss* of quick adaptation (using  $s = 1$  gradient steps) across tasks can then be formulated as

$$ML := \sum_{\mathcal{T}_j \sim p(\mathcal{T})} \mathcal{L}_{D_{\mathcal{T}_j}^{test}}(\theta'_j) = \sum_{\mathcal{T}_j \sim p(\mathcal{T})} \mathcal{L}_{D_{\mathcal{T}_j}^{test}}(\theta - \alpha \nabla_{\theta} \mathcal{L}_{D_{\mathcal{T}_j}^{tr}}(\theta)), \tag{13.14}$$

where  $p(\mathcal{T})$  is a probability distribution over tasks. This expression contains an inner gradient ( $\nabla_{\theta} \mathcal{L}_{\mathcal{T}_j}(\theta_j)$ ). As such, by optimizing this meta-loss using gradient-based techniques, we have to compute second-order gradients. One can easily see this in the computation below:

$$\begin{aligned}
\nabla_{\theta} ML &= \nabla_{\theta} \sum_{\mathcal{T}_j \sim p(\mathcal{T})} \mathcal{L}_{D_{\mathcal{T}_j}^{test}}(\theta'_j) \\
&= \sum_{\mathcal{T}_j \sim p(\mathcal{T})} \nabla_{\theta} \mathcal{L}_{D_{\mathcal{T}_j}^{test}}(\theta'_j) \\
&= \sum_{\mathcal{T}_j \sim p(\mathcal{T})} \mathcal{L}'_{D_{\mathcal{T}_j}^{test}}(\theta'_j) \nabla_{\theta}(\theta'_j) \\
&= \sum_{\mathcal{T}_j \sim p(\mathcal{T})} \mathcal{L}'_{D_{\mathcal{T}_j}^{test}}(\theta'_j) \nabla_{\theta}(\theta - \alpha \nabla_{\theta} \mathcal{L}_{D_{\mathcal{T}_j}^{tr}}(\theta)) \\
&= \underbrace{\sum_{\mathcal{T}_j \sim p(\mathcal{T})} \mathcal{L}'_{D_{\mathcal{T}_j}^{test}}(\theta'_j)}_{\text{FOMAML}} (\nabla_{\theta} \theta - \alpha \nabla_{\theta}^2 \mathcal{L}_{D_{\mathcal{T}_j}^{tr}}(\theta)), \tag{13.15}
\end{aligned}$$

where we used  $\mathcal{L}'_{D_{\mathcal{T}_j}^{test}}(\theta'_j)$  to denote the derivative of the loss function with respect to the test set, evaluated at the post-update parameters  $\theta'_j$ . The term  $\alpha \nabla_{\theta}^2 \mathcal{L}_{D_{\mathcal{T}_j}^{tr}}(\theta)$  contains the second-order gradients. The computation thereof is expensive in terms of time and memory costs, especially when the optimization trajectory is large (when using a larger number of gradient updates  $s$  per task). Finn et al. (2017) experimented with leaving out second-order gradients, by assuming  $\nabla_{\theta} \theta'_j = I$ , giving us first-order MAML (FOMAML, see Equation 13.15). They found that FOMAML performed reasonably similar implying that most of the benefit comes from the post-update gradients. This means that updating the initialization using only first-order gradients  $\sum_{\mathcal{T}_j \sim p(\mathcal{T})} \mathcal{L}'_{D_{\mathcal{T}_j}^{test}}(\theta'_j)$  is roughly equal to using the full gradient expression for the meta-loss in Equation 13.15. One can extend the meta-loss to incorporate multiple gradient steps by substituting  $\theta'_j$  by a multi-step variant.

Now, MAML is trained as follows: the initialization weights  $\theta$  are updated by continuously sampling a batch of  $m$  tasks  $B = \{\mathcal{T}_j \sim p(\mathcal{T})\}_{j=1}^m$ . Then, for every task  $\mathcal{T}_j \in B$ , an *inner update* is performed to obtain  $\theta'_j$ , in turn granting an observed loss  $\mathcal{L}_{D_{\mathcal{T}_j}^{test}}(\theta'_j)$ . These losses across a batch of tasks are used in the *outer update*:

$$\theta := \theta - \beta \nabla_{\theta} \sum_{\mathcal{T}_j \in B} \mathcal{L}_{D_{\mathcal{T}_j}^{test}}(\theta'_j). \tag{13.16}$$

The complete training procedure of MAML is displayed in Algorithm 13.2. At test time, when presented with a new task  $\mathcal{T}_j$ , the model is initialized with  $\theta$  and performs a number of gradient updates on the task data. Note that the algorithm for FOMAML is equivalent to Algorithm 13.2, except for the fact that the update on line 8 is done differently. That is, FOMAML updates the initialization with the rule  $\theta = \theta - \beta \sum_{\mathcal{T}_j \sim p(\mathcal{T})} \mathcal{L}'_{D_{\mathcal{T}_j}^{test}}(\theta'_j)$ .

Antoniou et al. (2019), in response to MAML, proposed many technical enhancements that can improve the training stability, performance, and generalization ability. Improvements include (i) updating the initialization  $\theta$  after every inner update step (instead of after all steps are done) to increase gradient propagation, (ii) using second-order gradients only after 50 epochs to increase the training speed, (iii) learning layer-wise learning rates to improve flexibility, (iv) annealing the metalearning rate  $\beta$  over time, and (v) some batch normalization tweaks (keep running statistics instead of batch-specific ones, and using per-step biases).

```

1: Randomly initialize  $\theta$ 
2: while not done do
3:   Sample batch of  $J$  tasks  $B = \mathcal{T}_1, \dots, \mathcal{T}_J \sim p(\mathcal{T})$ 
4:   for  $\mathcal{T}_j = (D_{\mathcal{T}_j}^{tr}, D_{\mathcal{T}_j}^{test}) \in B$  do
5:     Compute  $\nabla_{\theta} \mathcal{L}_{D_{\mathcal{T}_j}^{tr}}(\theta)$ 
6:     Compute  $\theta'_j = \theta - \alpha \nabla_{\theta} \mathcal{L}_{D_{\mathcal{T}_j}^{tr}}(\theta)$ 
7:   end for
8:   Update  $\theta = \theta - \beta \nabla_{\theta} \sum_{\mathcal{T}_j \in B} \mathcal{L}_{D_{\mathcal{T}_j}^{test}}(\theta'_j)$ 
9: end while

```

**Algorithm 13.2:** One-step MAML for supervised learning, by Finn et al. (2017)

MAML has attracted great attention within the field of metalearning for deep neural networks, perhaps due to its (i) simplicity (only requires two hyperparameters), (ii) general applicability, and (iii) strong performance. A downside of MAML, as mentioned above, is that it can be quite expensive in terms of running time and memory to optimize a base-learner for every task and compute higher-order derivatives from the optimization trajectories.

Several other optimization-based techniques build upon MAML. Here, we briefly mention some of them. *Meta-SGD* (Li et al., 2017) does not only learn an initialization point, but also appropriate learning rates for every single parameter in the network. *Latent embedding optimization* (LEO) (Rusu et al., 2018) attempts to find a good initialization in a lower-dimensional space, which can aid generalization performance. Several probabilistic versions of MAML have also been proposed, which learn a distribution over initializations (Grant et al., 2018; Finn et al., 2018) or multiple initializations which are jointly optimized (Yoon et al., 2018). Lastly, one can combine a representation module (e.g., a CNN) with closed-form base-learners, for which one can derive analytical solutions (Bertinetto et al., 2019; Lee et al., 2019).

### 13.5.4 Reptile

Reptile (Nichol and Schulman, 2018) is another optimization-based technique that, like MAML (Finn et al., 2017), solely attempts to find a good set of initialization parameters  $\theta$ . The way in which Reptile attempts to find this initialization is quite different from MAML. It repeatedly samples a task, trains on the task, and moves the model weights towards the trained weights (Nichol and Schulman, 2018). Algorithm 13.3 displays the pseudocode describing this simple process.

```

1: Initialize  $\theta$ 
2: for  $i = 1, 2, \dots$  do
3:   Sample task  $\mathcal{T}_j = (D_{\mathcal{T}_j}^{tr}, D_{\mathcal{T}_j}^{test})$  and corresponding loss function  $\mathcal{L}_{\mathcal{T}_j}$ 
4:    $\theta'_j = SGD(\mathcal{L}_{D_{\mathcal{T}_j}^{tr}}, \theta, k)$  {Perform  $k$  gradient update steps to get  $\theta'_j$ }
5:    $\theta := \theta + \epsilon(\theta'_j - \theta)$  {Move initialization point  $\theta$  towards  $\theta'_j$ }
6: end for

```

**Algorithm 13.3:** Reptile, by Nichol and Schulman (2018)

Nichol and Schulman (2018) note that it is possible to treat  $(\theta - \theta'_j)/\alpha$  as gradients, where  $\alpha$  is the learning rate of the inner stochastic gradient descent optimizer (line 4 in the pseudocode) and to feed that into a meta-optimizer (e.g., Adam). Moreover, instead of sampling one task at a time, one could sample a batch of  $n$  tasks and move the initialization  $\theta$  towards the average update direction  $\bar{\theta} = \frac{1}{n} \sum_{j=1}^n (\theta'_j - \theta)$ , granting the update rule  $\theta := \theta + \epsilon \bar{\theta}$ .

The intuition behind Reptile is that updating the initialization weights towards updated parameters will grant a good inductive bias for tasks from the same family. By performing Taylor expansions of the gradients of Reptile and MAML (both first order and second order), Nichol and Schulman (2018) show that the expected gradients differ in their direction. They argue, however, that in practice, the gradients of Reptile will also bring the model towards a point minimizing the expected loss over tasks.

A mathematical argument as to why Reptile works goes as follows: let  $\theta$  denote the initial parameters and  $\theta_j^*$  the optimal set of weights for task  $\mathcal{T}_j$ . Lastly, let  $d$  be the Euclidean distance function. Then, the goal is to minimize the distance between the initialization point  $\theta$  and the optimal point  $\theta_j^*$ :

$$\min_{\theta} \mathbb{E}_{\mathcal{T}_j \sim p(\mathcal{T})} \left[ \frac{1}{2} d(\theta, \theta_j^*)^2 \right]. \tag{13.17}$$

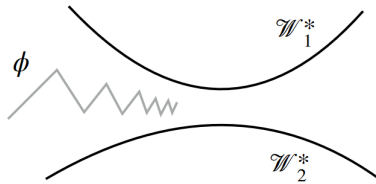


Fig. 13.15: Visualization of Reptile’s learning trajectory. Source: (Nichol and Schulman, 2018)

The gradient of this expected distance with respect to the initialization  $\theta$  is given by

$$\begin{aligned} \nabla_{\theta} \mathbb{E}_{\mathcal{T}_j \sim p(\mathcal{T})} \left[ \frac{1}{2} d(\theta, \theta_j^*)^2 \right] &= \mathbb{E}_{\mathcal{T}_j \sim p(\mathcal{T})} \left[ \frac{1}{2} \nabla_{\theta} d(\theta, \theta_j^*)^2 \right] \\ &= \mathbb{E}_{\mathcal{T}_j \sim p(\mathcal{T})} [\theta - \theta_j^*], \end{aligned} \tag{13.18}$$

where we used the fact that the gradient of the squared Euclidean distance between two points  $x_1$  and  $x_2$  is the vector  $2(x_1 - x_2)$ . Nichol and Schulman (2018) go on to argue that performing gradient descent on this objective would result in the following update rule:

$$\begin{aligned} \theta &= \theta - \epsilon \nabla_{\theta} \frac{1}{2} d(\theta, \theta_j^*)^2 \\ &= \theta - \epsilon (\theta_j^* - \theta). \end{aligned} \tag{13.19}$$

Since we do not know  $\theta_j^*$ , one can approximate this term by  $k$  steps of gradient descent  $SGD(\mathcal{L}_{\mathcal{T}_j}, \theta, k)$ . In short, Reptile can be seen as gradient descent on the distance minimization objective given in Equation 13.17. A visualization is shown in Figure 13.15.

The initialization  $\theta$  ( $\phi$  in the figure) is moving towards the optimal weights for tasks 1 and 2 in interleaved fashion (hence the oscillations).

In conclusion, Reptile is an extremely simple metalearning technique, which does not need to differentiate through the optimization trajectory like, e.g., MAML (Finn et al., 2017), saving time and memory costs. However, the theoretical foundation is a bit weaker, and performance may be a bit worse than that of MAML.

## Optimization-based techniques, in conclusion

Optimization-based techniques explicitly optimize for fast learning. A key advantage of optimization-based approaches is that they can achieve better performance on wider task distributions than, e.g., model-based approaches (Finn and Levine, 2018). However, optimization-based techniques optimize a base-learner for every task that they are presented with, which is computationally expensive (Hospedales et al., 2020).

## 13.6 Discussion and Outlook

In this section, we give a helicopter view of the methods discussed in this chapter, and the field of metalearning for deep neural networks in general.

In recent years, this field has gained much popularity. The idea of self-improving neural networks that can leverage prior learning experience to learn new tasks more quickly is quite appealing. Instead of training a new model from scratch for different tasks, we can use the same (metalearning) model across tasks. As such, metalearning can widen the applicability of powerful deep learning techniques to domains where less data is available and computational resources are limited.

Metalearning techniques for deep neural networks are characterized by their meta-objective, which allows them to maximize performance across various tasks, instead of a single one, as is the case in base-level learning objectives. This meta-objective is reflected in the training procedure of metalearning methods, as they learn on a set of different meta-training tasks. The few-shot setting lends itself nicely towards this end, as tasks consist of few data points. This makes it computationally feasible to train on many different tasks, and it allows us to evaluate whether a neural network can learn new concepts from few examples. Task construction for training and evaluation does require some special attention to prevent the *memorization problem* (meta-overfitting), where the neural network has memorized tasks seen at training time but fails to generalize to new tasks. Clever task design and meta-regularization may prove useful to avoid such problems (Yin et al., 2020). Furthermore, to improve test performance, it is beneficial to match training and test conditions (Vinyals et al., 2016), and perhaps train in a more difficult setting than the one that will be used for evaluation (Snell et al., 2017).

On a high level, there are three categories of metalearning in the context of deep learning, namely (i) metric-, (ii) model-, and (iii) optimization-based ones, which rely on computing input similarity, task embeddings with states, and task-specific updates, respectively. Each approach has strengths and weaknesses. Metric-learning techniques are simple and effective (Garcia and Bruna, 2017) but are not readily applicable outside of the supervised learning setting (Hospedales et al., 2020). Model-based techniques, on the other hand, can have very flexible internal dynamics but lack generalization ability to more distant tasks than the ones used at meta-training time (Finn and Levine, 2018). Optimization-based approaches have shown greater generalizability but are in general

computationally expensive, as they optimize a base-learner for every task (Finn and Levine, 2018; Hospedales et al., 2020).

Table 13.1 provides a concise, tabular overview of these approaches. Many techniques have been proposed for each one of the categories, and the underlying ideas may vary greatly, even within the same category. Table 13.2, therefore, provides an overview of all methods and key ideas that we have discussed in this chapter. Recall that  $\mathcal{T}_j$  is a task,  $D_{\mathcal{T}_j}^{tr}$  the corresponding training set,  $\mathcal{L}_{\mathcal{T}_j}$  the loss function,  $k_{\theta}(\mathbf{x}, \mathbf{x}_i)$  a kernel function returning the similarity between the two inputs  $\mathbf{x}$  and  $\mathbf{x}_i$ ,  $y_i$  are true labels for example inputs  $\mathbf{x}_i$ ,  $\theta$  are base-learner parameters, and  $g_{\varphi}$  is a (learned) optimizer with parameters  $\varphi$ .

### 13.6.1 Open challenges

Despite the great potential of metalearning for deep neural networks, there are still open challenges. In addition to the challenges mentioned above (computational costs and the memorization problem), there are two other major challenges. Firstly, most of the metalearning techniques discussed in this chapter are evaluated on narrow benchmark sets. This means that the data that the metalearner used for training are not too distant from the data used for evaluating its performance. As such, one may wonder how well these techniques are actually able to adapt to more distant tasks. Chen et al. (2019) showed that the ability to adapt to new tasks decreases as they become more distant from the tasks seen at training time. Increasing the degree of transferability of prior learning experience is an important and open challenge. As noted by Hospedales et al. (2020), the recently proposed meta-dataset (Triantafillou et al., 2019), could prove to be a great tool towards this end.

Secondly, Chen et al. (2019) have shown that a simple non-metalearning baseline yields competitive or better performance than some metalearning techniques on few-shot image classification. This begs the question of whether metalearning for deep neural networks is a good approach at all in the few-shot setting.

### 13.6.2 Future research

Future research is needed to address these challenges. Moreover, it is interesting to further investigate the applicability of metalearning for deep neural networks to online, active, and continual/lifelong learning settings (see, e.g., Finn et al. (2018); Yoon et al. (2018); Finn et al. (2019); Munkhdalai and Yu (2017); Vuorio et al. (2018)), as that could increase the applicability of deep learning tools in the real world.

Yet another direction for future research is the creation of *compositional* metalearning systems, which instead of learning flat and associative functions  $\mathbf{x} \rightarrow y$ , organize knowledge in a *compositional* manner. This would allow them to decompose an input  $\mathbf{x}$  into several (already learned) components  $c_1(\mathbf{x}), \dots, c_n(\mathbf{x})$ , which in turn could help the performance in low-data regimes (Tokmakov et al., 2019). Lastly, the question has been raised of whether some contemporary metalearning techniques for deep neural networks actually learn how to perform rapid learning, or simply learn a set of robust high-level features, which can be (re)used for many (new) tasks. Raghu et al. (2020) investigated this question for MAML (Finn and Levine, 2018) and found that it largely relies on feature reuse.

We end this chapter by formulating a list of open research questions concerning the challenges that we encounter when applying metalearning to deep neural networks:



- How can we design metalearners that are less susceptible to the memorization problem (Yin et al., 2020)?
- How well do current metalearning techniques generalize to more distant tasks, and how can we increase transferability (Finn and Levine, 2018)?
- How well do metalearning techniques perform in other domains such as active, online, and lifelong learning (Finn et al., 2019; Munkhdalai and Yu, 2017)?
- Can we reduce the computational costs of metalearning systems (Rajeswaran et al., 2019)?
- Can the principle of compositionality be successfully applied to metalearners (Barrett et al., 2018; Lake, 2019)?
- Can we understand why the non-metalearning baseline of Chen et al. (2019) outperforms some state-of-the-art metalearning techniques?
- Is it feasible and helpful to add more meta-abstraction levels, e.g. meta-metalearning, meta-meta-metalearning, etc. (Hospedales et al., 2020)?
- Can we design new metalearning techniques that rely more on rapid learning instead of reusing learned features (Raghu et al., 2020)?

Metalearning for deep neural networks is a vibrant field. Despite great advances, ample challenges remain to be solved, which makes for exciting future work.

## References

- Andrychowicz, M., Denil, M., Colmenarejo, S. G., Hoffman, M. W., Pfau, D., Schaul, T., Shillingford, B., and de Freitas, N. (2016). Learning to learn by gradient descent by gradient descent. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, pages 3988–3996, USA. Curran Associates Inc.
- Antoniou, A., Edwards, H., and Storkey, A. (2019). How to train your MAML. In *International Conference on Learning Representations*, ICLR'19.
- Barrett, D. G., Hill, F., Santoro, A., Morcos, A. S., and Lillicrap, T. (2018). Measuring abstract reasoning in neural networks. In *Proceedings of the 35th International Conference on Machine Learning*, ICML'18, pages 4477–4486. JMLR.org.
- Bertinetto, L., Henriques, J. F., Torr, P. H. S., and Vedaldi, A. (2019). Meta-learning with differentiable closed-form solvers. In *International Conference on Learning Representations*, ICLR'19.
- Chen, W.-Y., Liu, Y.-C., Kira, Z., Wang, Y.-C., and Huang, J.-B. (2019). A closer look at few-shot classification. In *International Conference on Learning Representations*, ICLR'19.
- Finn, C., Abbeel, P., and Levine, S. (2017). Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning*, ICML'17, pages 1126–1135. JMLR.org.
- Finn, C. and Levine, S. (2018). Meta-learning and universality: Deep representations and gradient descent can approximate any learning algorithm. In *International Conference on Learning Representations*, ICLR'18.
- Finn, C., Rajeswaran, A., Kakade, S., and Levine, S. (2019). Online meta-learning. In Chaudhuri, K. and Salakhutdinov, R., editors, *Proceedings of the 36th International Conference on Machine Learning*, ICML'19, pages 1920–1930. JMLR.org.
- Finn, C., Xu, K., and Levine, S. (2018). Probabilistic model-agnostic meta-learning. In *Advances in Neural Information Processing Systems 31*, NIPS'18, pages 9516–9527. Curran Associates Inc.

- Garcia, V. and Bruna, J. (2017). Few-shot learning with graph neural networks. In *International Conference on Learning Representations*, ICLR'17.
- Garnelo, M., Rosenbaum, D., Maddison, C., Ramalho, T., Saxton, D., Shanahan, M., Teh, Y. W., Rezende, D., and Eslami, S. M. A. (2018). Conditional neural processes. In Dy, J. and Krause, A., editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *ICML'18*, pages 1704–1713. JMLR.org.
- Grant, E., Finn, C., Levine, S., Darrell, T., and Griffiths, T. (2018). Recasting gradient-based meta-learning as hierarchical bayes. In *International Conference on Learning Representations*, ICLR'18.
- Graves, A., Wayne, G., and Danihelka, I. (2014). Neural Turing Machines. *arXiv preprint arXiv:1410.5401*.
- Hospedales, T., Antoniou, A., Micaelli, P., and Storkey, A. (2020). Meta-learning in neural networks: A survey. *arXiv preprint arXiv:2004.05439*.
- Koch, G., Zemel, R., and Salakhutdinov, R. (2015). Siamese Neural Networks for One-shot Image Recognition. In *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *ICML'15*. JMLR.org.
- Lake, B. M. (2019). Compositional generalization through meta sequence-to-sequence learning. In *Advances in Neural Information Processing Systems 33*, NIPS'19, pages 9791–9801. Curran Associates Inc.
- Lee, K., Maji, S., Ravichandran, A., and Soatto, S. (2019). Meta-learning with differentiable convex optimization. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 10657–10665.
- Li, K. and Malik, J. (2018). Learning to optimize neural nets. *arXiv preprint arXiv:1703.00441*.
- Li, Z., Zhou, F., Chen, F., and Li, H. (2017). Meta-SGD: Learning to learn quickly for few-shot learning. *arXiv preprint arXiv:1707.09835*.
- Mishra, N., Rohaninejad, M., Chen, X., and Abbeel, P. (2018). A simple neural attentive meta-learner. In *International Conference on Learning Representations*, ICLR'18.
- Munkhdalai, T. and Yu, H. (2017). Meta networks. In *Proceedings of the 34th International Conference on Machine Learning*, ICML'17, pages 2554–2563. JMLR.org.
- Nichol, A. and Schulman, J. (2018). Reptile: a scalable metalearning algorithm. *arXiv preprint arXiv:1803.02999*, 2:2.
- Oord, A. v. d., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A., and Kavukcuoglu, K. (2016). Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*.
- Raghu, A., Raghu, M., Bengio, S., and Vinyals, O. (2020). Rapid learning or feature reuse? towards understanding the effectiveness of MAML. In *International Conference on Learning Representations*, ICLR'20.
- Rajeswaran, A., Finn, C., Kakade, S. M., and Levine, S. (2019). Meta-learning with implicit gradients. In *Advances in Neural Information Processing Systems 32*, NIPS'19, pages 113–124. Curran Associates Inc.
- Ravi, S. and Larochelle, H. (2017). Optimization as a model for few-shot learning. In *International Conference on Learning Representations*, ICLR'17.
- Rusu, A. A., Rao, D., Sygnowski, J., Vinyals, O., Pascanu, R., Osindero, S., and Hadsell, R. (2018). Meta-learning with latent embedding optimization. In *International Conference on Learning Representations*, ICLR'18.
- Santoro, A., Bartunov, S., Botvinick, M., Wierstra, D., and Lillicrap, T. (2016). Meta-learning with memory-augmented neural networks. In *Proceedings of the 33rd International Conference on Machine Learning*, ICML'16, pages 1842–1850. JMLR.org.

- Shyam, P., Gupta, S., and Dukkipati, A. (2017). Attentive recurrent comparators. In *Proceedings of the 34th International Conference on Machine Learning, ICML'17*, pages 3173–3181. JMLR.org.
- Snell, J., Swersky, K., and Zemel, R. (2017). Prototypical networks for few-shot learning. In *Advances in Neural Information Processing Systems 30, NIPS'17*, pages 4077–4087. Curran Associates Inc.
- Sung, F., Yang, Y., Zhang, L., Xiang, T., Torr, P. H., and Hospedales, T. M. (2018). Learning to compare: Relation network for few-shot learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1199–1208.
- Tokmakov, P., Wang, Y.-X., and Hebert, M. (2019). Learning compositional representations for few-shot recognition. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 6372–6381.
- Triantafillou, E., Zhu, T., Dumoulin, V., Lamblin, P., Evci, U., Xu, K., Goroshin, R., Gelada, C., Swersky, K., Manzagol, P.-A., et al. (2019). Meta-dataset: A dataset of datasets for learning to learn from few examples. *arXiv preprint arXiv:1903.03096*.
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems 30, NIPS'17*, pages 5998–6008. Curran Associates Inc.
- Vinyals, O. (2017). Talk: Model vs optimization meta learning. <http://metalearning-symposium.ml/files/vinyals.pdf>. Neural Information Processing Systems (NIPS); accessed 06-06-2020.
- Vinyals, O., Blundell, C., Lillicrap, T., Kavukcuoglu, K., and Wierstra, D. (2016). Matching networks for one shot learning. In *Proceedings of the 30th International Conference on Neural Information Processing Systems, NIPS'16*, pages 3637–3645, USA. Curran Associates Inc.
- Vuorio, R., Cho, D.-Y., Kim, D., and Kim, J. (2018). Meta continual learning. *arXiv preprint arXiv:1806.06928*.
- Yin, M., Tucker, G., Zhou, M., Levine, S., and Finn, C. (2020). Meta-learning without memorization. In *International Conference on Learning Representations, ICLR'20*.
- Yoon, J., Kim, T., Dia, O., Kim, S., Bengio, Y., and Ahn, S. (2018). Bayesian model-agnostic meta-learning. In *Advances in Neural Information Processing Systems 31, NIPS'18*, pages 7332–7342. Curran Associates Inc.



## Automating Data Science

**Summary.** It has been observed that, in data science, a great part of the effort usually goes into various preparatory steps that precede model-building. The aim of this chapter is to focus on some of these steps. A comprehensive description of a given task to be resolved is usually supplied by the domain expert. Techniques exist that can process natural language description to obtain task descriptors (e.g., keywords), determine the task type, the domain, and the goals. This in turn can be used to search for the required domain-specific knowledge appropriate for the given task. In some situations, the data required may not be available and a plan needs to be elaborated regarding how to get it. Although not much research has been done in this area so far, we expect that progress will be made in the future. In contrast to this, the area of preprocessing and transformation has been explored by various researchers. Methods exist for selection of instances and/or elimination of outliers, discretization and other kinds of transformations. This area is sometimes referred to as *data wrangling*. These transformations can be learned by exploiting existing machine learning techniques (e.g., learning by demonstration). The final part of this chapter discusses decisions regarding the appropriate level of detail (granularity) to be used in a given task. Although it is foreseeable that further progress could be made in this area, more work is needed to determine how to do this effectively.

### 14.1 Introduction

It is well known in data science that a greater part of the effort goes into various preparatory operations, while the model-building step typically requires less effort. This has motivated researchers to examine how to automate the steps that precede the model building. In this chapter our aim is to analyze these preparatory steps. They include:

1. Defining the current problem/task
2. Identifying the appropriate domain-specific knowledge
3. Obtaining the data
4. Data preprocessing and other transformations
5. Automating model generation and its deployment
6. Automating report generation

Some steps (e.g., step 5) were discussed in detail in some of the previous chapters. A brief overview is given in Section 14.6. Step 4 will be discussed further on.

As for steps 1, 2, and 3, these are normally carried out by data science specialists and many people believe that it is difficult to automate them. This may be the reason why these steps were not included among the topics of various recent workshops on the topic of automating data science.<sup>1</sup> Although automation is difficult at this stage, it is nevertheless interesting to consider how this could be done and as a result provide assistance to data scientists.

Step 1, the issue of defining problems (or tasks), is discussed in Section 14.2. It is foreseeable that, as more experience is gathered from many different domains, some common patterns could be identified which in turn could provide a basis for at least partial automation. Section 14.3 discusses step 2: the problem of identifying useful domain-specific knowledge that could be included in the training data or the learning process itself. Section 14.4 is dedicated to step 3, discussing various strategies used for obtaining the training data. As pointed out in that section, it may happen that the data may not already be available in some application areas, and so a process must be devised regarding how to overcome this.

Section 14.5 discusses data preprocessing. This includes many types of transformations that are already well covered by AutoML methods, such as feature selection, but also *data wrangling*, which is not so easily automated but which has attracted a great deal of attention recently. Evidence of this are various workshops (e.g., AutoDS 2019 (Bie et al., 2019)) and research projects (e.g., AIDA (The Alan Turing Institute, 2020)). This section discusses also *data aggregation operations*, which are essential in many data mining applications.

Our motivation was to advance the overall understanding of this area and to share this with others. We hope that, this way, we can contribute to the development of future decision support systems in this area or even partial automation of the processes involved.

## 14.2 Defining the Current Problem/Task

The formal definition of a given problem (task) involves a number of steps which can be summarized as follows:

- Problem understanding and description
- Generating task descriptors
- Determining the task type and goals
- Identifying the task domain

Each step is discussed in detail in the respective subsection.

### 14.2.1 Problem understanding and description

Here we will consider two main types of problems: business problems and scientific problems. Regarding business problems, the initial part of the KDD process is usually identified as *business problem formulation*. We assume that this is done by a human and is formulated in natural language. For instance, if the aim is to obtain an understanding of certain activities of some institution (e.g., company or hospital), we need to define which aspects are of interest. These may include for instance:

---

<sup>1</sup>See, e.g., ADS 2019 workshop (Bie et al., 2019).

- Financial aspects (e.g. net profits or production costs)
- Proportion (or frequency) of successful cases
- Proportion (or frequency) of problematic cases (e.g., unsuccessful car models/branches that had to be closed, mortality cases in a hospital, etc.), among other aspects

This information helps to obtain a more detailed understanding of the domain and the data, enabling to focus on the appropriate data mining task.

### 14.2.2 Generating task descriptors

The task description is often done in natural language. It needs to be processed in order to extract a set of task descriptors (keywords). Let us see a few examples.

- Suppose the aim was to develop a system for credit rating and we were given its description. The description would be processed to retrieve a set of descriptors that would characterize the entity that has requested the credit (e.g., medium-sized company), in how many years the loan would be paid off, existing guarantees, purpose (e.g., amplifying installations or buying machinery), history of payments, possible spread, etc.
- If the aim were to develop a method for navigating a robot from one position to another, rather different descriptors would be required. Typically they would need to characterize the robot's initial position, direction (angle), speed, acceleration, and similar descriptors for the final set of attributes. Similarly, the descriptors could characterize the method for planning the trajectory and the avoidance strategy.
- If the aim were to provide control of a patient in an intensive unit, rather different descriptors would be required. These could include, for instance, the patient's heart rate, blood pressure, level of potassium, etc. and the need to maintain these within safe limits, which could also be specified.

This strategy was exploited, for instance, by Contreras-Ochando et al. (2019), who used certain domain-specific metafeatures to determine the type of task to carry out (e.g., certain metafeatures could indicate that the task involves transforming a date into a normalized format).

Whenever possible, the descriptors used should belong to a given set of common vocabulary/ontology terms, as this facilitates further processing.

### 14.2.3 Determining the task type and goals

Particular business or scientific objectives captured by appropriate descriptors determine, to a large extent, what kind of top-level task should be considered. If the aim was to simply obtain understanding of a particular domain, an unsupervised task (e.g., clustering) could be a good choice. If the aim is to predict values of certain variables (objective variables), the most appropriate top-level task could be classification or regression.

Determining the task type from a set of descriptors (keywords) can be formulated as a meta-level classification problem.

We note that a complex task may include various subtasks. The top-level task of a certain type (e.g., classification) may involve subtasks of a different type (e.g., regression).

## Role of learning goals

Determining which concepts (descriptors) should be brought into play is influenced by the learning goal. This issue was noted by the Russian psychologist Wygotski (1962). He drew attention to the fact that concepts arise and get developed if there is a specific need for them. Acquisition of concepts is thus a purposeful activity directed towards reaching a specific goal or a solution of a specific task.

This problem has been noted also by people in AI and ML. Various researchers, including Hunter and Ram (1992b,a), Michalski (1994), and Ram and Leake (2005), have argued that it is important to define explicit goals that guide learning. Learning is seen as a search through a knowledge space guided by the learning goal. Learning goals determine which part(s) of prior knowledge are relevant, what knowledge is to be acquired and in what form, how it is to be evaluated, and when to stop learning. The importance of planning in this process has also been identified by Hunter and Ram (1995).

## Providing a schedule for learning goals

As some more complex tasks are formulated in terms of multiple goals, it may be desirable to define also an ordering in which (some of) the goals should be achieved. This problem can be seen as the problem of *learning multiple interdependent concepts*. In effect, defining the ordering can be regarded as defining the appropriate procedural bias, as the ordering determines how the hypothesis space should be searched.

## 14.3 Identifying the Task Domain and Knowledge

The description of a given task using descriptors can be used to determine the domain to which that task belongs. Determining the domain is important, as it facilitates determining what type of data is required to solve the task.

One way to address this is to activate the appropriate domain-specific descriptors (ontology).

This process can be seen as the problem of activating the appropriate domain together with the corresponding set of domain-specific descriptors (ontology). This process can be compared to the process of activating Minsky's frames (Minsky, 1975). However, when the proposal on frames was written, the area of machine learning was not yet very advanced and so the mechanism used that would invoke frames was not well clarified at the time.

Basically, we see two different mechanisms that can be used for this aim. One involves keyword matching, and the other one classification. Both are described in the following subsections.

### Identifying the domain by matching descriptors/metafeatures

Both the task and the domain can be described using descriptors (keywords possibly organized in ontologies) or metafeatures. The domain can then be identified by matching the task and domain descriptors. The aim is to identify the domain with the closest match. It is necessary to define the appropriate measure of similarity between the task descriptors and particular domain descriptors. Naturally, one would seek the domain which is most similar to the given task.



## Identifying the domain by classification

Determining the domain can be regarded as a meta-level classification task. The input is a particular set of task descriptors of the given task, and the output is a particular domain characterized by a set of domain descriptors. As domains could be organized in a hierarchy, classification can be done at various levels of this hierarchy. For instance, the task descriptors could suggest that the given task pertains to *medicine* at a higher level and to *obstetrics* at a lower level.

## Representation of data and goals

It is important to have an appropriate representation for both data and the goals. Various schemes have been proposed in the past. For instance, Stepp and Michalski (1983) proposed *goal dependency networks* (GDN). In this chapter we have discussed the role of descriptors that help to determine the task type and the appropriate data. Some of these can be identified as descriptors of learning goals. So, for instance, *company profit* can be seen as a descriptor of data, if we are talking about a particular data item stored in a database. If, on the other hand, the aim is to predict its value from other information, then it becomes a learning goal.

## 14.4 Obtaining the Data

After the domain descriptors have been identified, it is necessary to identify and obtain the actual data. Typically, the aim is to identify a part of the data which is relevant for the task at hand, as this facilitates further processing. If the data were held in a relational database, then it would be necessary to identify the relevant part, i.e., a subset of possibly interconnected tables. Each table has a heading, and the terms used in each column often suggest the type of information contained there. However, sometimes the terms may be abbreviations or codes attributed by the designer. Hence, the problem of matching task descriptors to data descriptors would be facilitated if each name in the table heading were accompanied by the correct ontological descriptor(s).

### 14.4.1 Select existing data or plan how to obtain data?

The next important question is whether the data is already available or not. If it is available, it is possible to advance. If the data is not available or insufficient for the current task, new data needs to be gathered (e.g., by interacting with the “outside world” or by running experiments). The latter strategy was used in a robot scientist called *Adam* (King et al., 2009). This system autonomously generates hypotheses concerning functional genomics of a particular type of yeast (*Saccharomyces cerevisiae*) and then conducts tests to experimentally verify them.

### 14.4.2 Identifying the domain-specific data and background knowledge

Identifying the correct part of the data is not a trivial matter. Consider, for instance, the task of predicting whether to concede credit to a given customer. If the data did not include the relevant part (e.g., tables and the corresponding features), the learning

system would be unable to generate the correct inductive hypothesis. In this case, the search space does not include the inductive hypotheses that we would like the system to come up with.

If, on the other hand, the data unnecessarily included too many items, some of which might be irrelevant, the system might not arrive at the right hypothesis within a given time budget. Although in this case the search space would include the right inductive hypotheses, the system may have difficulty discovering them.

In inductive logic programming (ILP) the data is represented in the form of facts. As Srinivasan et al. (1996) have shown, the learning problem can often be defined using a set of constraints. One important one is  $H \wedge Bk \models E+$ , stating that the inductive hypothesis  $H$  together with the background knowledge  $Bk$  should logically imply the set of positive examples  $E+$ .<sup>2</sup> Srinivasan et al. (1996) observed that the performance of the learning system is sensitive to the type and amount of background knowledge. In particular, background knowledge that contains information irrelevant to the task considered can make the search for the correct hypothesis more difficult.

So, in general, if our aim was to automate this process, we would need to identify not only the relevant domain-specific data but also the relevant part of background knowledge. More details on some related issues are given in the following subsections.

### 14.4.3 Obtaining the data and background knowledge from different sources

As Contreras-Ochando et al. (2019) have pointed out, data science must integrate data from very different data sources. These may include databases, repositories, the web, spreadsheets, text documents, and images, among others. The integration may involve homogeneous sources (e.g., two different databases of similar type, but with different content), or heterogeneous sources (e.g. text and image).

#### Obtain data by accessing the OLAP data cube

One useful concept in this area is the so-called *OLAP data cube*, which is simply a multi-dimensional array of data (Gray et al., 2002). The operation *slice* permits to select a rectangular subset of a cube by choosing a single value for one of its dimensions. The operation *dice* is similar, but allows us to set specific values of multiple dimensions.

Several other operations are discussed in Section 14.5.4, which focuses on rather specific transformations whose aim is to alter the granularity of data.

## 14.5 Automating Data Preprocessing and Transformation

Data preprocessing and transformation can be regarded as one of many steps in the target workflow (pipeline). We will be discussing various preprocessing operations that are useful in classification pipelines. Usually, the selection of a particular preprocessing operation cannot be done in isolation from other elements in the pipeline. For instance, discretization needs to be carried out only if the particular classifier (e.g., naive Bayes) requires this.

---

<sup>2</sup>Srinivasan et al. (1996) refer to this constrain as *strong sufficiency*.

Different methods that can be used to design such a pipeline in a systematic fashion were discussed in detail in Chapter 7 and hence will not be reviewed here again. Our aim here is to complement this discussion by describing studies whose aim is to further automate the process, including tasks such as data wrangling.

As Chu et al. (2016) pointed out, data preprocessing serves either to repair data (e.g., based on some quality rules) or to transform data such that it leads to better results in further analysis (e.g., when applying a machine learning algorithm). Here, we consider the following operations:

- Data transformation/data wrangling
- Instance selection/cleaning/outlier elimination
- Data preprocessing

Data transformation/data wrangling is discussed in Subsection 14.5.1. Instance selection is discussed in Subsection 14.5.2. Data preprocessing includes various operations, such as:

- Feature selection
- Discretization
- Nominal to binary transformation
- Normalization/standardization
- Missing value imputation
- Feature generation (e.g., PCA or embeddings)

Detailed descriptions of these operations are discussed in various textbooks (see, e.g., Dasu and Johnson (2003)). Subsection 14.5.3 discusses ways of automatizing the selection of the appropriate preprocessing operation.

Although full automation of data science has been an ultimate goal of many research studies, aiming for this has shown to be computationally expensive, mainly due to the search space involved. In addition, some decisions may require domain knowledge (e.g., is a certain outlier safe to remove or not?), or there may not exist a ground truth to learn from. In Chapter 7, we discussed systems that go a long way towards automating pipeline design, such as Auto-sklearn (Feurer et al., 2015, 2019). This system includes many preprocessing operations (e.g., normalization, missing value imputation, and feature selection). However, even this state-of-the-art system is currently not capable of automating the full data science pipeline. Still, such systems can serve as a useful starting point for further developments.

The aim of this section is to cover other, quite different approaches that typically aim to automate or semi-automate a specific preprocessing technique (e.g., outlier detection). Even though these approaches are not always easy to include in AutoML systems, they do show how to solve sub-problems in isolation, and still help in building larger, efficient systems in the future.

### 14.5.1 Data transformation/data wrangling

Data wrangling is the process of transforming and mapping data from one data representation format (e.g., “raw” data) to another. The objective of this transformation is to make it appropriate for subsequent processing. So, for instance, the transformation may convert information in a spreadsheet into a tabular dataset.

## Inferring data types

This process often requires inferring the *data types* (e.g., date, float, integer, and string) of certain dataset entities (e.g., features represented in columns). Valera and Ghahramani (2017) proposed a Bayesian method that is able to detect such types based on the distribution of the data in a certain region. However, this does not work optimally if there are missing data and anomalies in the data. System *pType* by Ceritli et al. (2020) is a more recent method that employs a simpler probabilistic model that detects such anomalies and robustly infers data types.

## Some wrangling approaches

Some systems automatically infer relevant transformations, learned either from a demonstration provided by the user or from a given set of examples. However, as providing examples by the user incurs costs, ideally this is done on the basis of a limited number of examples. This is normally achieved by imposing a strong bias on the actual task (e.g., assumptions about the output format in a particular domain), enabling the system to prioritize some transformations over others.

This was exploited in one early system in this area – *Trifecta Wrangler* (Kandel et al., 2011) – which generates a ranked list of possible transformations. These transformations can be seen as simple programs, which was explored by various researchers already in the 1970s and 1980s (e.g., Mitchell (1977); Mitchell et al. (1983); Brazdil (1981), among others).

## System FOOFAH

Jin et al. (2017) developed a technique to synthesize data transformation programs from example transformations, described using input–output example pairs. Their system, FOOFAH, searches the space of possible data transformation operations to generate a program that will perform the desired transformation. In general, the aim is to transform a poorly structured grid of values (e.g., a spreadsheet table) into a relational table that can be used by ML programs. Such a transformation can be a combination of operators. Some operators are oriented towards columns, and others to rows or even to the whole table (e.g. transpose). Some column operators are shown below:

- Drop: deletes a column in the table
- Move: relocates a column from one position to another
- Merge: concatenates two columns and appends the merged column to the end
- Split: separates a column into two or more parts at the occurrences of the delimiter
- Divide: divides one column into two columns based on some predicate
- Extract: extracts the first match of a given regular expression in each cell of a designated column

The authors show that their system obtains the required programs faster than Wrangler discussed earlier.

## Usage of ILP in wrangling approaches

Some researchers followed the approach of inductive programming (IP) or inductive logic programming (ILP) (Gulwani et al., 2015), which led to a wider spread. Microsoft, for instance, has decided to include a data wrangling tool *Flashfill* (Gulwani et al., 2012) in Excel.

The ILP approaches seem particularly useful to the task of transformation, as it is possible to constrain the search by specifying the appropriate domain-specific background knowledge (BK). This was exploited for instance by Contreras-Ochando et al. (2019), who used certain domain-specific metafeatures to determine the type of task to carry out, which in turn brought into play the appropriate background knowledge. The problem tackled by these authors involved recognizing the type of a given *named entity*, which may be a date, email, name, phone number, or some other entity, together with the coding convention that depends on the respective country (e.g. France, UK, etc.). This determines which part of the BK should be activated. In one task, for instance, the goal was to extract the day of the month from inputs in different formats. For instance, “25-03-74” should give 25 and “03/29/86” should return 29.

## Systems TDE and SYNTH

TDE (Transform Data by Example) (He et al., 2018) works in a similar way, but it uses a library of program snippets that represent different transformations. The given examples are used to select the ones that generate the correct output.

SYNTH (De Raedt et al., 2018) can learn to carry out automatic completion of data in a set of worksheets. As the authors have shown, solving this problem requires a number of different steps. First, it is necessary to discover equations and/or obtain one or more predictive models that permit to calculate the values in one cell using the values in other cells. Secondly, it is necessary to apply the equations and/or induced models to fill in empty cells, parts of rows or columns. To address these different tasks, the system uses several different components:

- An automatic data wrangling system (Synth-a-Sizer) that transforms a dataset into a traditional attribute-value format so that standard machine learning systems could be applied
- System *Mercs* that induces predictive models (Van Wolputte et al., 2018)
- System *TacLe* that induces constraints and formulas in spreadsheets
- A component that ties learning and inference together

### 14.5.2 Instance selection and model compression

Some systems select a subset of data to be cleaned prior to passing them to further processing. *ActiveClean* (Krishnan et al., 2016), for instance, identifies the instances (records) that are more likely to affect the results of subsequent modeling.

The area of outlier detection/elimination is related to this. Many well-known methods exist, such as *isolation forests* (Liu et al., 2012), *one-class SVMs*, or *local outlier factors* (Breunig et al., 2000), among others. Typically these methods do not work well in unsupervised mixed-type scenarios. Eduardo et al. (2020) uses *variational autoencoders* (VAEs) to detect and repair outliers even in this setting.

One application of this is *model compression* (or *model distillation*), where a simpler and hopefully better model is obtained by selecting the training data. One pioneering

work in this area was the work of John (1995) on so-called *robust decision trees*, who studied the effects of label noise. After learning a tree, all misclassified instances were removed from the learning set and a new tree was learned. Although this process did not result in an accuracy increase, the resulting tree was much smaller. Similar approaches for *k*-NN were presented by Tomek (1976) and Wilson and Martinez (2000). Further work on filtering misclassified instances was reported by Smith and Martinez (2018).

### 14.5.3 Automating the selection of the preprocessing method

Bilalli et al. (2018, 2019) predicted which preprocessing techniques are appropriate for a given classification algorithm. The preprocessing methods include, for instance, discretization, transformation from nominal to binary, normalization and standardization of values, replacement of missing values, and introduction of principal components. For some of these operations, both supervised and unsupervised variants have been considered. Besides, some of the operations can be applied to just one attribute, while others are global and can thus be applied to all attributes. They build a meta-model in the form of random forest for each target classification algorithm and use it together with the metafeatures of the target dataset to predict which preprocessing technique should be included in the pipeline. The associated probability of classification can be used to order these operations in terms of their expected utility. The resulting system, called PERSISTANT, thus recommends which preprocessors to consider given a certain classification algorithm, but not which classifier should be used. Similarly, Schoenfeld et al. (2018) build meta-models predicting when a preprocessing algorithm is likely to improve the accuracy or runtime of a particular classifier.

Other systems extend this idea to *sequences* of preprocessing steps (sub-pipelines). Learn2Clean (Berti-Equille, 2019) attempts to select the optimal sequence of tasks for preprocessing and takes into account the given dataset with the aim of maximizing the performance of the complete workflow. Similarly, in DPD (Quemy, 2019), the sequential model-based optimization (SMBO) technique is used to automatically select and tune preprocessing operators to improve the performance score within a restricted time budget. The authors have found that, for certain NLP preprocessing operators, specific configurations are optimal for several different algorithms.

Finally, some systems produce rankings of promising workflows (pipelines) that can be used to speed up the search. Cachada et al. (2017) and Abdulrahman et al. (2018) describe a method of recommending the most suitable workflow for a target dataset. Unlike PERSISTANT, it is capable of recommending a workflow, which may include preprocessing (CFS feature selection) followed by a classifier with a certain configuration of hyperparameter settings. The recommendations are in the form of a ranking of the most useful workflows. The system can then use this ranking to conduct experiments to identify the workflow with the best performance.

### 14.5.4 Changing the granularity of representation

Finally, one of the most low-level tasks that could potentially be automated is how to extract data from a database so that it can be used for learning.

## Generating aggregate data from an OLAP cube

As we have mentioned earlier, an *OLAP data cube* is a multi-dimensional array of data (Gray et al., 2002). The operation *drill down/up* allows us to navigate in the data cube,

going from the most summarized (up) to the most detailed (down). A *roll-up* involves summarizing the data along a particular dimension. The summarization rule might be an aggregate function, such as computing totals, for one or more of the dimensions, depending on the required level of detail. Common aggregate functions are SUM, AVG, COUNT, MAX, and MIN, among others. The intermediate data structures, sometimes referred to as *cuboids*, can be organized in a lattice. So, the roll-up operation can be seen as moving up in the lattice of cuboids.

Various studies have been carried out in the past. One integrates features obtained as a result of aggregation into the data mining process. For instance, Charnay (2016) used such features in relational decision trees and random forests.

Features that result from aggregation operations are commonly used in many different systems. Let us consider one example here from robotic soccer. As Stone (2000) pointed out, we may want to identify a receiver that can safely receive the ball. One way of finding this is by establishing the distances to surrounding opponent players and identifying the nearest one. The nearest distance represents an aggregate feature.

Some data mining systems, including for instance RapidMiner, provide an aggregate operator, which is well integrated with the rest of the data mining operations (Hofmann and Klinkenberg, 2013).

All legal operations define a search space, which may be too large to search through manually. There are many possibilities regarding which tables may be selected and/or joined. Further choice arises with respect to the selection of columns or aggregation operations that can be applied to them. Hence, there is a need for a kind of automatic data selection that would be able to support this process by exploring this space and providing good recommendation(s) to the user.

The topic of changing the granularity of representation is continued in Chapter 15. However, the focus there is different from this section, as it discusses the problem of changing granularity by introducing new concepts.

## 14.6 Automating Model and Report Generation

### 14.6.1 Automating model generation and deployment

The aim of this step is to search for the best workflow (pipeline) for the given task. This can be done with a suitable AutoML/meta-level system.

AutoML systems conduct a search for the best workflow. The search may involve experiments on the target dataset with the objective of determining how different candidate workflows (and differently configured variants) perform. Chapter 6 provides more details about the techniques involved and about some systems that exist in this area.

Metalearning systems can be used, provided similar problems have been dealt with in the past and hence a corresponding meta-database exists. Chapters 2, 5, and 7 provide more details. Another possible strategy is to adapt an existing model using knowledge transfer techniques. More details on this can be found in Chapter 12.

### 14.6.2 Automating report generation

One good example in this area is the Automated Statistician (Steinruecken et al., 2019). This project aims to automate various phases of data science, including automated construction of models from data, comparison of different models, and automatic elabo-

ration of reports with minimal human intervention. The reports include, besides basic graphs and statistics, human-readable descriptions in natural language.

## References

- Abdulrahman, S. M., Cachada, M. V., and Brazdil, P. (2018). Impact of feature selection on average ranking method via metalearning. In *European Congress on Computational Methods in Applied Sciences and Engineering, 6th ECCOMAS Thematic Conference on Computational Vision and Medical Image Processing (VipIMAGE 2017)*, pages 1091–1101. Springer.
- Berti-Equille, L. (2019). Learn2clean: Optimizing the sequence of tasks for web data preparation. In *The World Wide Web Conference*, page 2580–2586. ACM, NY, USA.
- Bie, D., De Raedt, L., and Hernandez-Orallo, J., editors (2019). *ECMLPKDD Workshop on Automating Data Science (ADS)*, Würzburg, Germany. <https://sites.google.com/view/autods>.
- Bilalli, B., Abelló, A., Aluja-Banet, T., and Wrembel, R. (2018). Intelligent assistance for data pre-processing. *Computer Standards & Interf.*, 57:101–109.
- Bilalli, B., Abelló, A., Aluja-Banet, T., and Wrembel, R. (2019). PRESISTANT: Learning based assistant for data pre-processing. *Data & Knowledge Engineering*, 123.
- Brazdil, P. (1981). *Model of Error Detection and Correction*. PhD thesis, University of Edinburgh.
- Breunig, M., Kriegel, H.-P., Ng, R., and Sander, J. (2000). LOF: Identifying density-based local outliers. In *Proceedings of the MOD 2000*. ACM.
- Cachada, M., Abdulrahman, S., and Brazdil, P. (2017). Combining feature and algorithm hyperparameter selection using some metalearning methods. In *Proc. of Workshop AutoML 2017, CEUR Proceedings Vol-1998*, pages 75–87.
- Ceritli, T., Williams, C. K., and Geddes, J. (2020). ptype: probabilistic type inference. *Data Mining and Knowledge Discovery*, pages 1–35.
- Charnay, C. (2016). *Enhancing supervised learning with complex aggregate features and context sensitivity*. PhD thesis, Université de Strasbourg, Artificial Intelligence.
- Chu, X., Ilyas, I., Krishnan, S., and Wang, J. (2016). Data cleaning: Overview and emerging challenges. In *Proceedings of the International Conference on Management of Data, SIGMOD '16*, page 2201–2206.
- Contreras-Ochando, L., Ferri, C., Hernández-Orallo, J., Martínez-Plumed, F., Ramírez-Quintana, M. J., and Katayama, S. (2019). Automated data transformation with inductive programming and dynamic background knowledge. In *Proceedings of ECML PKDD 2019 Conference*.
- Dasu, T. and Johnson, T. (2003). *Exploratory Data Mining and Data Cleaning*. John Wiley & Sons, Inc., NY, USA.
- De Raedt, L., Blockeel, H., Kolb, S., Kolb, S., and Verbruggen, G. (2018). Elements of an automatic data scientist. In *Proc. of the Advances in Intelligent Data Analysis XVII, IDA 2018*, volume 11191 of LNCS. Springer.
- Eduardo, S., Nazábal, A., Williams, C. K., and Sutton, C. (2020). Robust variational autoencoders for outlier detection and repair of mixed-type data. In *International Conference on Artificial Intelligence and Statistics*, pages 4056–4066.
- Feurer, M., Klein, A., Eggenberger, K., Springenberg, J., Blum, M., and Hutter, F. (2015). Efficient and robust automated machine learning. In Cortes, C., Lawrence, N., Lee, D., Sugiyama, M., and Garnett, R., editors, *Advances in Neural Information Processing Systems 28, NIPS'15*, pages 2962–2970. Curran Associates, Inc.



- Feurer, M., Klein, A., Eggenberger, K., Springenberg, J. T., Blum, M., and Hutter, F. (2019). Auto-sklearn: Efficient and robust automated machine learning. In Hutter, F., Kotthoff, L., and Vanschoren, J., editors, *Automated Machine Learning: Methods, Systems, Challenges*, pages 113–134. Springer.
- Gray, J., Bosworth, A., Layman, A., and Pirahesh, H. (2002). Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. In *Proceedings of the International Conference on Data Engineering (ICDE)*, page 152–159.
- Gulwani, S., Harris, W., and Singh, R. (2012). Spreadsheet data manipulation using examples. *Communications of the ACM*, 55(8):97–105.
- Gulwani, S., Hernandez-Orallo, J., Kitzelmann, E., Muggleton, S., Schmid, U., and Zorn, B. (2015). Inductive programming meets the real world. *Communications of the ACM*, 58(11):90–99.
- He, Y., Chu, X., Ganjam, K., Zheng, Y., Narasayya, V., and Chaudhuri, S. (2018). Transform-data-by-example (TDE): an extensible search engine for data transformations. In *Proceedings of the VLDB Endowment*, pages 1165–1177.
- Hofmann, M. and Klinkenberg, R. (2013). *RapidMiner: Data Mining Use Cases and Business Analytics Applications*. Data Mining and Knowledge Discovery. Chapman & Hall/CRC.
- Hunter, L. and Ram, A. (1992a). Goals for learning and understanding. *Applied Intelligence*, 2(1):47–73.
- Hunter, L. and Ram, A. (1992b). The use of explicit goals for knowledge to guide inference and learning. In *Proceedings of the Eighth International Workshop on Machine Learning (ML'91)*, pages 265–269, San Mateo, CA, USA. Morgan Kaufmann.
- Hunter, L. and Ram, A. (1995). Planning to learn. In Ram, A. and Leake, D. B., editors, *Goal-Driven Learning*. MIT Press.
- Jin, Z., Anderson, M. R., Cafarella, M., and Jagadish, H. V. (2017). Foofah: A programming-by-example system for synthesizing data transformation programs. In *Proc. of the International Conference on Management of Data, SIGMOD '17*, page 1607–1610.
- John, G. H. (1995). Robust decision trees: Removing outliers from databases. In *Knowledge Discovery and Data Mining*, pages 174–179. AAAI Press.
- Kandel, S., Paepcke, A., Hellerstein, J., and Heer, J. (2011). Wrangler: Interactive visual specification of data transformation scripts. In *CHI '11, Proceedings of SIGCHI Conference on Human Factors in Computing Systems*, page 3363–3372.
- King, R. D., Rowland, J., Oliver, S. G., Young, M., Aubrey, W., Byrne, E., Liakata, M., Markham, M., Pir, P., Soldatova, L. N., Sparkes, A., Whelan, K. E., and Clare, A. (2009). The automation of science. *Science*, 324(5923):85–89.
- Krishnan, S., Wang, J., Wu, E., Franklin, M., and Goldberg, K. (2016). ActiveClean: Interactive data cleaning for statistical modeling. *PVLDB*, 9(12):948–959.
- Liu, F. T., Ting, K. M., and Zhou, Z.-H. (2012). Isolation-based anomaly detection. *ACM Trans. Knowl. Discov. Data*, 6(1):3:1–3:39.
- Michalski, R. (1994). Inferential theory of learning: Developing foundations for multi-strategy learning. In Michalski, R. and Tecuci, G., editors, *Machine Learning: A Multi-strategy Approach, Volume IV*, chapter 1, pages 3–62. Morgan Kaufmann.
- Minsky, M. (1975). A framework for representing knowledge. In Winston, P. H., editor, *The Psychology of Computer Vision*, pages 211–277. McGraw-Hill.
- Mitchell, T. (1977). *Version spaces: A candidate elimination approach to rule learning*. PhD thesis, Electrical Engineering Department, Stanford University.

- Mitchell, T., Utgoff, P., and Banerji, R. (1983). Learning by experimentation: Acquiring and refining problem-solving heuristics. In Michalski, R., Carbonell, J., and Mitchell, T., editors, *Machine Learning. Symbolic Computation*, pages 163–190. Tioga.
- Quemy, A. (2019). Data pipeline selection and optimization. In *Proc. of the Int. Workshop on Design, Optimization, Languages and Analytical Processing of Big Data, DOLAP '19*.
- Ram, A. and Leake, D. B., editors (2005). *Goal Driven Learning*. MIT Press.
- Schoenfeld, B., Giraud-Carrier, C., M. Poggeman, J. C., and Seppi, K. (2018). Feature selection for high-dimensional data: A fast correlation-based filter solution. In *Workshop AutoML 2018 @ ICML/IJCAI-ECAI*. Available at site <https://sites.google.com/site/automl2018icml/accepted-papers>.
- Smith, M. R. and Martinez, T. R. (2018). The robustness of majority voting compared to filtering misclassified instances in supervised classification tasks. *Artif. Intell. Rev.*, 49(1):105–130.
- Srinivasan, A., King, R. D., and Muggleton, S. H. (1996). The role of background knowledge: using a problem from chemistry to examine the performance of an ILP program.
- Steinruecken, C., Smith, E., Janz, D., Lloyd, J., and Ghahramani, Z. (2019). The Automatic Statistician. In Hutter, F., Kotthoff, L., and Vanschoren, J., editors, *Automated Machine Learning, Series on Challenges in Machine Learning*. Springer.
- Stepp, R. S. and Michalski, R. S. (1983). How to structure structured objects. In *Proceedings of the International Workshop on Machine Learning*, Urbana, IL, USA.
- Stone, P. (2000). *Layered Learning in Multiagent Systems: A Winning Approach to Robotic Soccer*. MIT Press.
- The Alan Turing Institute (2020). *Artificial intelligence for data analytics (AIDA)*. <https://www.turing.ac.uk/research/research-projects/artificial-intelligence-data-analytics-aida>.
- Tomek, I. (1976). An experiment with the edited nearest-neighbor rule. *IEEE Transactions on Systems, Man, and Cybernetics*, 6:448–452.
- Valera, I. and Ghahramani, Z. (2017). Automatic discovery of the statistical types of variables in a dataset. volume 70 of *Proceedings of Machine Learning Research*, pages 3521–3529, International Convention Centre, Sydney, Australia. PMLR.
- Van Wolputte, E., Korneva, E., and Blockeel, H. (2018). Mercs: multi-directional ensembles of regression and classification trees. <https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16875/16735>, pages 4276–4283.
- Wilson, D. and Martinez, T. (2000). Reduction techniques for instance-based learning algorithms. *Machine Learning*, 38(3):257–286.
- Wygotski, L. S. (1962). *Thought and Language*. MIT Press.

## Automating the Design of Complex Systems

**Summary.** This chapter discusses the issue of whether it is possible to automate the design of rather complex workflows needed when addressing more complex data science tasks. The focus here is on symbolic approaches, which continue to be relevant. The chapter starts by discussing some more complex operators, including, for instance, conditional operators and operators used in iterative processing. Next, we discuss the issue of introduction of new concepts and the changes of granularity that can be achieved as a result. We review various approaches explored in the past, such as constructive induction, propositionalization, reformulation of rules, among others, but also draw attention to some new advances, such as feature construction in deep NNs. It is foreseeable that in the future both symbolic and subsymbolic approaches will coexist in systems exhibiting a kind of functional symbiosis. There are tasks that cannot be learned in one go, but rather require a sub-division into subtasks, a plan for learning the constituents, and joining the parts together. Some of these subtasks may be interdependent. Some tasks may require an iterative process in the process of learning. This chapter discusses various examples that can stimulate both further research and some practical solutions in this rather challenging area.

### 15.1 Introduction

The aim of this chapter is to discuss the issue of automating the design of more complex systems than the ones discussed in previous chapters. The aim is to present various techniques which have proved useful in the past research, as they can be regarded as fundamental architectural building blocks of both current and future intelligent systems.

Complex systems can be divided into two major groups. One includes systems that learn from relatively small sets of examples, while the others use large sets of examples (big data) in the learning process. Let us analyze each group in more detail.

As we have mentioned, the systems in the first group typically use modest numbers of examples. Some systems of this kind were discussed in the previous chapter on automating data science and, in particular, data wrangling. These systems often learn from human-provided input, in the form of input–output pairs or even individual steps showing how a particular problem is to be solved. They often exploit symbolic formulations (e.g., ILP) and may exploit background knowledge. This chapter contains material that is closely related to these systems.

The systems in the second group typically use large sets of examples (big data) in the learning process. In this category we find various systems that learn from big data and data streams (see Chapter 11 for details). Many current systems developed for complex tasks, such as vision or machine translation, do not use symbolic methods any more, but deep neural networks (DNNs), embeddings, etc. It is interesting to note that processing the input and constructing embeddings can be seen as a process of capturing the essential information. This knowledge is useful when dealing with new problems. So a question arises of whether symbolic approaches that are discussed in this chapter are at all relevant in the light of current developments.

In our view there are various reasons why they are relevant. One is to draw attention to various common principles that have proved useful in both symbolic and subsymbolic learning. The notion of, for instance, *constructive induction* discussed in Section 15.3 can be related to the introduction of new nodes higher up in the DNN.

The second reason is related to the issue of *explainability*. Many people find it important nowadays that the current systems are able to explain how they arrived at a particular recommendation or decision. As it is difficult to do this in some systems, including NNs, a symbolic model can be maintained to model the more complex system. This way, it is possible to provide the required explanation. If the symbolic model of the underlying complex process is *faithful* in the process of modeling, this solution is more satisfactory than some rather complex explanation that is difficult to comprehend. Explainability contributes towards what some people now call *trusted AI*.

The third reason is that humans use both subsymbolic and symbolic reasoning, and presumably this has a good motive. Possibly it is related to the human capacity called *creativity*. Currently, we can only postulate what this involves, but we believe that abstract concepts play an important role in that process.

It is foreseeable that future systems would benefit from a stronger interplay between the subsymbolic learning (e.g., embeddings) and symbolic counterpart (e.g., ontologies). So, subsymbolic and symbolic approaches could co-exist in a kind of functional symbiosis. Therefore, in our view, an overview of methods and techniques that have proved useful in the process of learning complex systems, albeit symbolic, continues to be relevant.

## Overview of this chapter

More complex tasks may require plans with conditional operators or iterative processing. This issue is briefly reviewed in Section 15.2. Introduction of new concepts is addressed in Section 15.3. There are tasks that cannot be learned in one go, but rather require a sub-division into sub-tasks and then formulating a plan for learning the constituents and joining the parts together. This methodology is discussed in Section 15.4. Some tasks require an iterative process in the process of learning. More details on this can be found in Section 15.5. There are problems whose tasks are interdependent. One such problem is analyzed in Section 15.6.

## 15.2 Exploiting a Richer Set of Operators

Workflows, sometimes also called pipelines, and their more general counterparts, networks, can be seen as plans to be executed. As sometimes it is necessary to verify certain

conditions before advancing with the execution, it is necessary to enrich the set of operators to be able to deal with such situations. The constructs in various programming languages suggest the main classes that need to be considered:

- Procedures
- Conditional operators
- Repeat operators

The first type, *procedures*, is related to abstract operators discussed in Chapter 7. Conditional operators have been used already in the early days of planning (Dean and Boddy, 1988). This is useful in many situations. Consider, for instance, a robot plan to go from one location to another. The plan could state: *If no obstacle is in the way, go straight, otherwise initiate an “avoidance plan”*. Repeat operators are needed because, as the name suggests, some operations need to be repeated. Kietz et al. (2012), for instance, used this operator in their planning system to design a KDD workflow. It was used to repeat the operation of preprocessing each column of a given data table.

### 15.3 Changing the Granularity by Introducing New Concepts

In Chapter 7 we discussed methods that would come up with the best possible workflow for a given task. We note that the basic building blocks were given. This included both the descriptors of the data (features, attributes) and also the descriptions of given operators (e.g., a particular preprocessing operation or application of a particular classifier). A good question that we wish to address here is: where do concepts come from?

Before trying to answer this question, let us clarify what we mean by “concepts” here. Basically, one set of concepts describes *states* and another *events* or its special subcategories, namely *processes* and *actions*, that relate one state to another.

There are two possible ways these can be introduced in a system: One is by introducing them from the exterior. The other is by some autonomous process controlled by the agent. More details about the two alternatives follow.

#### Introducing new concepts from external sources

Researchers who study human learning in children note that many concepts are acquired from interactions with their progenitors. The new concepts undoubtedly shape the process of further learning.

Some AI systems use also external sources. One common source is the internet. Many information extraction tasks benefit by accessing Wikipedia pages. One famous system in this area is Watson, which is a question-answering system capable of answering questions posed in natural language (Ferrucci et al., 2013). This system was initially developed to answer questions on the quiz show Jeopardy!

Although many systems exist that interact with the exterior, to the best of our knowledge not much work has been done up to now in the area that would combine the introduction of new concepts from the exterior with their introduction relying on autonomous methods.

## Introducing new concepts autonomously

When talking about the introduction of concepts, we need to be aware that this is often a continued process, as De Raedt (2008) (Chapter 7) pointed out. Introduced concepts may undergo various phases of revision later. In the following subsections we provide pointers to some past work in this area.

### 15.3.1 Defining new concepts by clustering

Clustering (or cluster analysis) is a branch of machine learning that groups the data items that share certain attributes (features). It belongs to the so-called unsupervised learning methods, as the data points (examples) have not been classified, labeled, classified, or categorized. Instead of responding to the classes, clustering identifies commonalities in the data, represented by attributes (features). So, the clusters (groups) obtained by this process can be regarded as new concepts. The system can give internal names to these concepts (e.g., concept#12), but whenever it is necessary to communicate with other agents, including humans, commonly established names need to be adopted.

### 15.3.2 Constructive induction

Michalski is known for introducing the term *constructive induction*. Diettrich and Michalski (1983) have defined constructive induction as an induction process that changes the description space, that is, a process that produces new descriptors (e.g., features) that were not present in the input data.

### 15.3.3 Reformulation of theories consisting of rules

#### Reformulating theories by specialization

System ELM (Brazdil, 1981, 1984) could learn to solve arithmetic and algebraic problems. The given rules (clauses) were reformulated as a result of learning. Many operations involved could be described as specializations.

The set of problems given to the systems involved a certain order. Simpler problems preceded the more complex ones, as often the solutions of simpler problems could be reused later. These included some problems from the area of arithmetic, such as  $4 + (1 + 2) = X1$ , and algebra, namely equations with one unknown such as,  $(2 + 1) + X1 = 5$ . In the learning phase the correct result was also given, together with some intermediate correct steps leading to the solution, referred to as a *trace*.

The given rules expressed valid operations in arithmetic and algebra, similar to Peano axioms. This includes a successor function and its inverse, i.e., a predecessor function. Besides, rules expressing associativity and commutativity were also given. Some rules also enabled the introduction of a new variable. For instance, the goal  $X1 + X2 = X3$  could be transformed into  $X1 = X4$  &  $X4 + X2 = X3$ .

The system searched for a solution by expanding the search space. The given trace provided a useful help in the search, but if enough computing power were given, the system could reach the solution even without it. After the solution has been reached, the aim of the system was to reformulate the given rules, so as to avoid the wrong steps. This was done in two ways. The first one involved imposing some ordering constraints on the existing rules in the form of explicit commands, such as  $r_i > r_j$ , expressing that

rule  $r_i$  should be given preference to  $r_j$ . All rules followed in effect a partial order. So, the system employed certain aspects of *preference learning* (Fürnkranz and Hüllermeier, 2003; Fürnkranz and Hüllermeier, 2011).

Cycles were not permitted. To avoid them, the system used the second method, namely constraining the applicability of the rules. The constraints included, for instance,  $int(X_i)$ , expressing that  $X_i$  is an integer,  $var(X_i)$ , expressing that  $X_i$  is a variable, and  $X_i := X_j$  expressing that  $X_i$  and  $X_j$  can be unified. One of the main ideas of this work was to analyze the solution traces and gather examples of situations where the given rule was used correctly, referred to as *selection contexts*, together with examples where the rule was applied incorrectly (*rejection contexts*).

The aim of the system was to find the most specific term that would be able to discriminate between the two. This system bears some similarities to system LEX (Mitchell, 1977, 1982), which was conceived around the same time. The concepts in ELM are not organized in a lattice according to their generality as in LEX, which is well known for the introduction of version spaces.

## Folding and unfolding

Burstall and Darlington (1977) considered the issue of reformulating programs. They defined various operations (unfolding, folding) that enabled them to do this. Here we review briefly the operation of *folding*:

*If  $E \leftarrow E'$  and  $F \leftarrow F'$  are equations and there is some occurrence in  $F'$  of an instance of  $E'$ , replace it by the corresponding instance of  $E$ , obtaining  $F''$ ; then add the equation  $F \leftarrow F''$ .*

The operation of *folding* is defined in a similar way, but instead of searching for an occurrence of an instance of  $E'$  in  $F'$ , it searches for an instance of  $E$ .

Although the definition of  $F \leftarrow F''$  is new, we would be reluctant to regard it as a “new concept”. For this, the concept would have to satisfy other criteria, such as having wide applicability across several different tasks.

## Absorption

The operation of *absorption* defined by Sammut (1981) has a similar aim to *folding*. This operator was exploited later in a propositional learning system called DUCE (Muggleton, 1987) and its first-order upgrade CIGOL (Muggleton and Buntine, 1988) together with three other operators, namely identification, intra-construction, and inter-construction. Let us analyze one example that illustrates the use of absorption, which was adapted from De Raedt (2008) (Chapter 7). For instance, given the theory consisting of clauses

$$\begin{aligned} \text{primate} &\leftarrow \text{twoLegs}, \text{noWings} \\ \text{man} &\leftarrow \text{twoLegs}, \text{noWings}, \text{notHair}, \text{noTail} \end{aligned}$$

the application of the *absorption* operator transforms the second clause into

$$\text{man} \leftarrow \text{primate}, \text{notHair}, \text{noTail}$$

We note that this operator changes the clause body in one of the clauses but does not introduce a new concept (a new clause with a new symbol in the clause head).

### 15.3.4 Introduction of new concepts expressed as rules

Muggleton and Buntine (1988) have observed that several rules containing a common conjunction of premises can be rewritten by replacing the common group by a new concept. This operation is captured by so-called *inter-construction* (Muggleton and Buntine, 1988; Muggleton, 1991). The operator was defined for both the propositional and first-order setting. Here we review briefly one propositional example adapted from (De Raedt, 2008) (Ch.7). Suppose the input theory is

$$\begin{aligned} man &\leftarrow twoLegs, noWings, notHairy, noTail \\ gorilla &\leftarrow twoLegs, noWings, Hairy, noTail, black \end{aligned}$$

The combination of *twoLegs, noWings* that appears in both definitions of *man* and *gorilla* can be singled out and substituted by a new concept with an internal name (e.g., *p*). The system can ask the user to suggest an appropriate name, and let us assume that this name is *primate*. So the revised theory will be

$$\begin{aligned} primate &\leftarrow twoLegs, noWings \\ man &\leftarrow primate, notHairy, noTail \\ gorilla &\leftarrow primate, Hairy, noTail, black \end{aligned}$$

The inter-construction operator could be seen as a mechanism for constructing new features in the sense intended by Michalski when discussing constructive induction.

Although the examples presented here include concepts like “man”, “gorilla”, etc., there is no reason why the operations discussed here, including e.g., inter-construction, could not be applied also to sequences of operators.

### 15.3.5 Propositionalisation

The term *propositionalization* was introduced in first-order learning/inductive logic programming (ILP). The aim was to make ILP learning more effective. This technique was exploited in LINUS (Lavrač et al., 1991; Lavrač and Džeroski, 1994). This system employs propositional learners in a more expressive logic programming framework. The algorithm solves ILP problems in the following three steps:

- Transform the learning problem from relational to attribute-value (propositional) form
- Solve the transformed problem by an attribute-value (propositional) learner
- Transform the learned concept back into relational form

Propositionalization is probably quite close in spirit to what Michalski intended, since this does result in new features, which are then used to represent the given data instances.

### 15.3.6 Automatic feature construction in deep NNs

Deep neural networks have become popular in recent years, mainly due to various successful applications in image and speech recognition. Each layer of nodes is involved in training a distinct set of features while exploiting the output of previous layers. The higher layers typically contain more complex features, as they aggregate and recombine features from the previous layer. The features can be exported from the NN and reused in other applications.

We note that this mechanism can be compared to the method of reusing solutions of subtasks in further learning, discussed in Section 15.4. It indicates that the mechanism is quite general and can be of use in various, rather different settings.



### 15.3.7 Reusing new concepts to redefine ontologies

Various approaches that enable the definition of new concepts that were discussed in this chapter can be applied also to operator sequences. The operator *inter-construction* of Muggleton and Buntine (1988) can be used to identify common subsequences of operators and replace them by a new *abstract operator*. If this process is continued, it is possible to envisage that in the end we can obtain an ontology of abstract and concrete operators that represent an important part of many current HTN planning systems.

## 15.4 Reusing New Concepts in Further Learning

The idea of reusing the solutions of subtasks in further learning permeates the whole area of ML. Let us consider again Figure 7.2. It suggests that a “DM operation” can be accomplished by carrying out a *preprocessing operation*, followed by *model-building operation* and *post-processing operation*. The figure gives more details at lower levels. For instance, it determines what kind of preprocessing operations can be considered.

As we have pointed out in Chapter 7, the ontology captures certain declarative and procedural bias, which is useful for constructing solutions to new tasks.

### Example: using acquired skills in learning more complex behavior

The notion that solutions of sub-problems can be reused when solving other, more complex problems was discussed by various other authors. Stone (2000), for instance, discusses a strategy called *layered learning* which starts with learning basic skills. The acquired skills can be considered as primitive actions. After these have been learned, the system proceeds to learn more complex actions.

A similar position was also put forward by Lake et al. (2017), who expressed the view that one should reuse approaches that worked well before. The consequence of this is that, with every new skill learned, learning new skills becomes easier.

Let us analyze an example of simulated soccer discussed by Stone (2000) concerned with passing a ball to another agent, referred to as the *receiving agent*, who must intercept the ball. Intercepting the ball represents a skill that has been acquired before. The learned ability can be reused.

As there are several players in the field, the passer must decide to whom the ball should be passed. Here, the identifier of the player can be regarded as the parameter that needs to be learned.

The passer announces his intention to pass, and the teammates reply when they are ready to receive. The passer chooses a receiver randomly during training and announces to whom it is passing. The receiver and four nearest opponents attempt to get the ball using the learned interception skill. The training example is classified as a success if the receiver manages to advance the ball towards the opponent’s goal, and a failure otherwise. The authors point out that the performance can be increased further if the passer is given other options, besides just passing the ball. That is, if there are no conditions to pass, the agent may decide to continue to dribble the ball.

## 15.5 Iterative Learning

The notion of using the output of learning in further learning seems quite general. In this subsection we discuss the problem of learning recursive definitions that follows this idea. This is an important issue, as many concepts are best represented this way.

Various approaches have been proposed in the literature. Burstall and Darlington (1977) have proposed a system for reformulating recursive definitions for a given task. Model inference system (MIS) of Shapiro (1996) conducted a general-to-specific search for clauses covering examples and was able to synthesize recursive definitions. Quite a detailed account of this system is presented by De Raedt (2008). Various other systems followed, including, for instance, RTL (Baroglio et al., 1992), CRUSTACEAN (Aha et al., 1994), and SKILit (Jorge and Brazdil, 1996; Jorge, 1998), among others.

Here we focus on one method mentioned above (SKILit) that is able to synthesize the correct definitions with a relatively high probability on the basis of a few randomly chosen examples. However, in principle, other systems (e.g., ALEPH) could have been used too. The method does not assume any a priori knowledge of the solution except the domain-specific knowledge in the form of a clause-structure grammar similar to grammars discussed in Section 8.3.

Suppose we are interested in synthesizing an algorithm for processing structured objects, such as lists. Before doing this, we would want to capture (and exploit) the following idea: *If you want to process a structured object using some procedure (P), decompose it into parts, then invoke the same procedure recursively, and then join the partial solutions.*

We can conceive a grammar to capture this. In this domain, the general structure of a clause (possibly recursive) can be specified as follows:

$$\text{body}(P) \rightarrow \text{decomp}, \text{test}, \text{recursion}(P), \text{comp}$$

The grammar specifies that, in this domain, four different groups of literals are needed. The first group decomposes a list into parts. The second one carries out a test whose outcome is either true or false. The third group enables the introduction of the recursive call. The fourth group consists of composition literals that enable us to construct the output from parts.<sup>1</sup> Symbol  $P$  carries the name of the predicate of the head literal whose definition we wish to synthesize. If, for instance, the aim is to synthesize *insertion sort*, this symbol would represent *isort*.

Let us examine just the definition of the recursion group, but skip all other details, as these can be found in the original paper.

$$\begin{aligned} \text{recursion}(P) &\rightarrow \text{recursive\_lit}(P). \\ \text{recursion}(P) &\rightarrow \text{recursive\_lit}(P), \text{recursion}(P). \\ \text{recursive\_lit}(P) &\rightarrow [P]. \end{aligned}$$

We note that the recursive group contains a link to itself, as the name suggests. So, a question arises about how to adapt the method that follows a bottom-up approach, discussed in Section 15.4, namely learning sub-concepts before learning higher-level concepts. The objective of this subsection is to clarify just this point.

Wherever a link exists to itself in the grammar or the corresponding graph, the solution consists of repeating the learning cycle more than once. In each step a tentative theory,  $T_i$ , is produced and subsequently reused as background knowledge in the next cycle of the induction process (see Figure 15.1). If the stopping criterion is satisfied, the process terminates.

<sup>1</sup>The reader can compare this with the grammar used in the design of KDD workflows (Section 7.2).

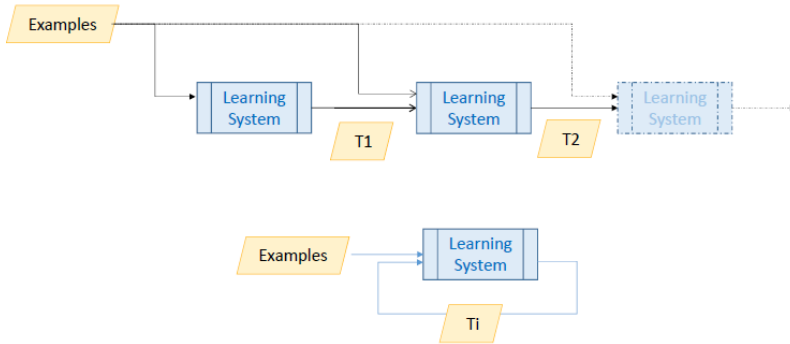


Fig. 15.1: Iterative learning

**Example: learning the definition of *insertion sort***

Let us examine how the method of iterative induction helped synthesizing the definition of insertion sort (*isort/2*) (Jorge and Brazdil, 1996). First, let us see which predicates were made available to the system as domain-specific metaknowledge.

Initially, these include some basic list handling predicates, including *split/4* and *concat/3*, among others. Some properties generated in the first step of iterative induction, corresponding to theory *T1*, are shown below:

$$\begin{aligned} isort([A, B], [A, B]) &\leftarrow A < B. \\ isort([A, B], [B, A]) &\leftarrow B < A \end{aligned}$$

The properties induced by the system represent a correct (but specific) program for sorting two-element lists. The first clause establishes that, if the list is already sorted (i.e., the first element is smaller than the second one), the order should be maintained. The second clause takes care of swapping the two elements when necessary. It is easy to see that both properties generalize many concrete examples. Thanks to these discovered properties, the system was often able to generate the correct definition in the next step.

In another experiment (Jorge, 1998), the predicate *insertb(A, B, C)* was added to the background knowledge. This predicate inserts element *A* into list *B* and generates *C* as a result. In this run, the property

$$isort([A, B], C) \leftarrow insertb(A, [B], C)$$

was generated as theory *T2*. In the next cycle the correct theory *T3* was generated:

$$\begin{aligned} isort([], []). \\ isort([A|B], C) &\leftarrow isort(B, D), insertb(A, D, C). \end{aligned}$$

The method described was able to synthesize the correct definitions (with relatively high probability) of various predicates on the basis of a few examples. Overall, the accuracies were of the order of 90% or more when only five positive examples were given.

## 15.6 Learning to Solve Interdependent Tasks

There are situations where we need to control two or more processes in a coordinated manner. Let us consider some situations when this occurs. Consider, for instance, how we put a car into motion (assuming that the car does not use automatic gear change). Having started the motor, we need to keep releasing the clutch and pressing the accelerator. Both actions need to be carried out in a coordinated manner, as otherwise the car would stall.

A similar situation occurs when controlling a simulated plane, as some maneuvers involve more than one control. For instance, if the aim is to turn left, it is necessary to determine whether to adjust not only the *elevators*, but also the *ailerons* and the *thrust*.

Let us analyze a situation discussed by Camacho and Brazdil (2001) which involves two controls, control *i* and control *j*, which affect one another. The general situation is illustrated in Figure 15.2. The interdependence of the two controls is illustrated by a link interconnecting them.

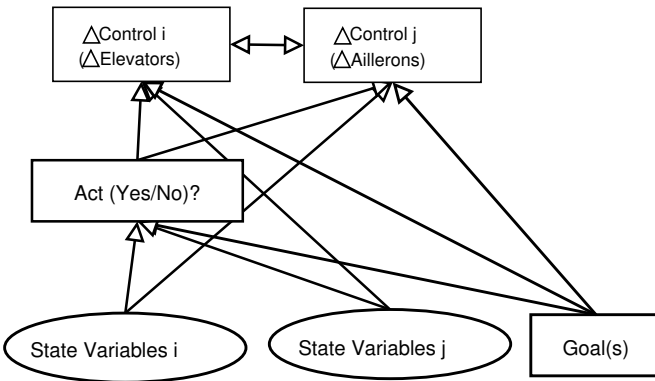


Fig. 15.2: Concept graph with two interdependent controls

If we were to ignore the effect of control *j* (*ailerons*), learning the change of control *i* (*elevators*) at time *t* would involve the state variables at time *t* - 1. In addition, we would need to consider the goals and whether there is a need to act (values at time *t* - 1). As our aim is to capture the effect of the other controls, we need to add the relevant information to the model. Here we need to add information about the change of control *j* (*ailerons*) used at time *t* - 1 to the model for control *i* (*elevators*).

A similar approach is adopted when learning the change of control *j* (*ailerons*). The information used involves the state variables at time *t* - 1, the current goal, and the current value of the change of control *i* (*elevators*) at time *t* - 1.

The method outlined was validated using extensive experiments (Camacho and Brazdil, 2001). It was shown that, if the strategy outlined is followed, it is possible to acquire the ability to deal with several controls at the same time in an apparently coordinated manner.

The approach follows the basic methodology of bottom-up learning (or layered learning) discussed earlier in Section 15.4. Due to the interdependence of concepts, the approach can be regarded as a variant of iterative learning discussed in Section 15.5. Some concepts learned are used as input in the next phase of learning.

## References

- Aha, D. W., Lapointe, S., Ling, C. X., and Matwin, S. (1994). Inverting implication with small training set. In Bergadano, F. and De Raedt, L., editors, *Machine Learning: ECML-94, European Conference on Machine Learning, Catania, Italy*, volume 784 of *Lecture Notes in Artificial Intelligence*, pages 31–48. Springer.
- Baroglio, C., Giordana, A., and Saitta, L. (1992). Learning mutually dependent relations. *Journal of Intelligent Information Systems*, 1:159–176.
- Brazdil, P. (1981). *Model of Error Detection and Correction*. PhD thesis, University of Edinburgh.
- Brazdil, P. (1984). Use of derivation trees in discrimination. In O’Shea, T., editor, *ECAI 1984 - Proceedings of 6th European Conference on Artificial Intelligence*, pages 239–244. North-Holland.
- Burstall, R. M. and Darlington, J. (1977). A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67.
- Camacho, R. and Brazdil, P. (2001). Improving the robustness and encoding complexity of behavioural clones. In De Raedt, L. and Flach, P., editors, *Proceedings of the 12th European Conference on Machine Learning (ECML ’01)*, LNAI 2167, pages 37–48, Freiburg, Germany. Springer.
- De Raedt, L. (2008). *Logical and Relational Learning*. Springer.
- Dean, T. and Boddy, M. (1988). Reasoning about partially ordered events. *Artificial Intelligence*, 36:375–399.
- Diettrich, T. and Michalski, R. (1983). A comparative review of selected methods for learning from examples. In Michalski, R., Carbonell, J., and Mitchell, T., editors, *Machine Learning: An Artificial Intelligence Approach*, pages 41–82. Tioga Publishing Company.
- Ferrucci, D., Levas, A., Bagchi, S., Gondek, D., and Mueller, E. (2013). Watson: Beyond Jeopardy! *Artificial Intelligence*, 199:93–105.
- Fürnkranz, J. and Hüllermeier, E. (2003). Pairwise preference learning and ranking. In Lavrač, N., Gamberger, D., Blockeel, H., and Todorovski, L., editors, *Proceedings of the 14th European Conference on Machine Learning (ECML2003)*, volume 2837 of LNAI, pages 145–156. Springer-Verlag.
- Fürnkranz, J. and Hüllermeier, E. (2011). *Preference Learning*. Springer-Verlag.
- Jorge, A. M. (1998). *Iterative Induction of Logic Programs*. PhD thesis, Faculty of Sciences, University of Porto.
- Jorge, A. M. and Brazdil, P. (1996). Architecture for iterative learning of recursive definitions. In De Raedt, L., editor, *Advances in Inductive Logic Programming*, volume 32 of *Frontiers in Artificial Intelligence and applications*. IOS Press.
- Kietz, J.-U., Serban, F., Bernstein, A., and Fischer, S. (2012). Designing KDD-Workflows via HTN-Planning for Intelligent Discovery Assistance. In Vanschoren, J., Brazdil, P., and Kietz, J.-U., editors, *PlanLearn-2012, 5th Planning to Learn Workshop WS28 at ECAI-2012, Montpellier, France*.
- Lake, B. M., Ullman, T. D., Tenenbaum, J. B., and Gershman, S. J. (2017). Building machines that learn and think like people. *Beh. and Brain Sciences*, 40.
- Lavrač, N. and Džeroski, S. (1994). *Inductive Logic Programming: Techniques and Applications*, chapter 5. LINUS: Using attribute-value learners in an ILP framework, pages 81–122. Ellis Horwood.
- Lavrač, N., Džeroski, S., and Grobelnik, M. (1991). Learning non-recursive definitions of relations with LINUS. In *Proceedings of the 5th Working Session on Learning*, pages 265–281. Springer.

- Mitchell, T. (1977). *Version spaces: A candidate elimination approach to rule learning*. PhD thesis, Electrical Engineering Department, Stanford University.
- Mitchell, T. (1982). Generalization as Search. *Artificial Intelligence*, 18(2):203–226.
- Muggleton, S. (1987). Duce: an oracle-based approach to constructive induction. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence*, pages 287–292. Morgan Kaufmann.
- Muggleton, S. and Buntine, R. (1988). Machine invention of first-order predicated by inverting resolution. In *Proceedings of the 5th International Workshop on Machine Learning*, pages 339–351. Morgan Kaufmann.
- Muggleton, S. H. (1991). Inverting resolution principle. In Hayes, J. E., Michie, D., and Tyugu, É., editors, *Machine Intelligence 12: Towards an Automated Logic and Thought*.
- Sammut, C. (1981). Concept learning by experiment. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence, Vancouver*.
- Shapiro, E., editor (1996). *Algorithmic Program Debugging*. MIT Press.
- Stone, P. (2000). *Layered Learning in Multiagent Systems: A Winning Approach to Robotic Soccer*. MIT Press.

**Organizing and Exploiting Metadata**





## Metadata Repositories

**Summary.** This chapter presents a review of online repositories where researchers can share data, code, and experiments. In particular, it covers OpenML, an online platform for sharing and organizing machine learning data automatically. OpenML contains thousands of datasets and algorithms, and millions of experimental results. We describe the basic philosophy involved, and its basic components: datasets, tasks, flows, setups, runs, and benchmark suites. OpenML has API bindings in various programming languages, making it easy for users to interact with the API in their native language. One important feature of OpenML is the integration into various machine learning toolboxes, such as Scikit-learn, Weka, and mlR. Users of these toolboxes can automatically upload all their results, leading to a large repository of experimental results.

### 16.1 Introduction

All around the globe, thousands of machine learning experiments are being executed on a daily basis, generating a constant stream of empirical information on machine learning techniques. If we could capture, store, and organize all these results, they would provide a rich and versatile resource for many different metalearning applications.

This chapter covers OpenML (Vanschoren et al., 2014), an online platform for machine learning researchers to share and organize machine learning data automatically and in fine detail. OpenML engenders a dynamic approach to experimentation, in which experiments can be freely shared, linked together, and immediately reused by researchers all over the world. At the same time, it keeps a detailed log of all meta-data about the datasets, algorithms, and experimental results, which enables us to learn across all these datasets and experiments, and then transfer this information when learning new tasks.

### 16.2 Organizing the World Machine Learning Information

Paradoxically, while the machine learning community so greatly values the proper collection of data to allow trustworthy analysis, there is surprisingly little work on systematically collecting and organizing the outputs of machine learning experiments in a way that allows meta-level learning.

### 16.2.1 The need for better metadata

This has a profound effect on progress in machine learning. Indeed, without access to reusable prior experiments to build on, each study has to start from scratch. In practice, this limits the depth of many studies, which makes them less generalizable, less interpretable, or even downright contradictory or biased (Aha, 1992; Hand, 2006; Keogh and Kasetty, 2003; Hoste and Daelemans, 2005; Perlich et al., 2003). This makes it very hard for us as a community to correctly interpret the literature and guide future work. Moreover, the competitive mindset of machine learning research often focuses more on small studies dominating the state of the art, rather than large-scale, rigorous and informative analysis (Sculley et al., 2018). The way that we run machine learning experiments is also full of undocumented assumptions and decisions, and a lot of this detail never makes it into papers. As a result, machine learning is grappling with a reproducibility crisis (Hutson, 2018; Hirsh, 2008).

Machine learning experiments can be very sensitive to their exact inputs, such as the exact training sets, hyperparameter settings, implementation details, and evaluation procedures. It is therefore crucial to log and document these exactly. If even humans grapple to understand the exact meaning and truthfulness of empirical results, then metalearning techniques that automatically learn from this data can be very easily misled. If we want to have any chance of obtaining deep and generalizable metalearning results, we first need to collect and organize extensive, finely detailed, and correct metadata. This goes beyond the resources of any individual researcher. It is a community-wide effort that requires good practices as well as the necessary tools to systematically gather detailed empirical data and pushing it to online platforms that help us structure and reuse the world's machine learning information.

### 16.2.2 Tools and initiatives

Several initiatives do aim to partially alleviate these problems, for instance through the creation of dataset repositories. The UCI repository (Dheeru and Taniskidou, 2017) and LIBSVM (Chang and Lin, 2011) offer a wide range of datasets. Many more focused repositories also exist, such as UCR (Chen et al., 2015) for time series data and Mulan (Tsoumakas et al., 2011) for multilabel datasets. More recent initiatives additionally provide programmatic access to datasets, such as the `kaggle.com` and PMLB (Olson et al., 2017) APIs for downloading general (mostly classification) datasets, the KEEL (Alcala et al., 2010) API for imbalanced classification and datasets with missing values, and the `skdata` (Bergstra et al., 2015) API for downloading computer vision and natural language processing datasets.

Other platforms additionally link datasets to reproducible experiments.<sup>1</sup> Early initiatives include StatLog (Michie et al., 1994), MetaL (Brazdil et al., 2009), DELVE,<sup>2</sup> and MLcomp,<sup>3</sup> but none of these projects is still being maintained. Experiment databases for machine learning (Vanschoren et al., 2012), the forerunners of OpenML, were among the first to organize large amounts of empirical results and make them queryable through online interfaces, but they required users to manually translate their experiments into a common format, which was tedious and prone to errors and omissions. More recently,

<sup>1</sup>Data mining challenge platforms, such as `kaggle.com`, `chalearn.org`, and `aicrowd.com` do share leaderboard results, but these are often not reproducible.

<sup>2</sup><http://www.cs.toronto.edu/~delve/>

<sup>3</sup>`mlcomp.org`

the OpenAI Gym (Brockman et al., 2016) allows running and evaluating reproducible reinforcement learning experiments, but lacks rich, organized metadata about complete training episodes. AI benchmarking systems such as MLPerf<sup>4</sup> and reproducibility initiatives such as PapersWithCode<sup>5</sup> do provide very interesting evaluation results but are often too targeted to provide general metadata for subsequent metalearning. It is important to note, though, that reproducibility is a key requirement for metalearning, since we need to be able to trust the metadata on which we build.

## 16.3 OpenML

Vanschoren et al. (2014) introduced the OpenML project, an online platform for sharing datasets and reproducible experiments. OpenML provides APIs (in Python, Java, and R) for downloading data in uniform formats into popular machine learning libraries, and uploading and comparing the ensuing results. It also provides metadata for standardizing evaluations (e.g., predefined train-test splits) and for in-depth analysis of evaluation results.

OpenML is integrated into various popular machine learning tools, such as Weka (Hall et al., 2009), R (Bischl et al., 2016b), Scikit-learn (Buitinck et al., 2013; Pedregosa et al., 2011), and MOA (Bifet et al., 2010a), and several more integrations are in progress, including deep learning tools. This allows anyone to easily import datasets into these tools, pick any algorithm or workflow to run, and automatically share all obtained results. Results are being produced locally: everyone that participates can run experiments locally (or anywhere they want) and afterwards share the results on OpenML. The web interface provides easy access to all collected data and code, compares all results obtained on the same data or algorithms, builds data visualizations, and supports online discussions.

OpenML offers various services to share and find datasets, to download or create scientific *tasks*, to share and find algorithm pipelines (called *flows*), and to share and organize experiments (called *runs*).

### 16.3.1 Datasets

For many metalearning applications, every dataset is a single metadata point. Hence, it is crucial to provide a rich and growing set of varied datasets. OpenML allows datasets to be uploaded directly from the environments where they were created (e.g., from a Python script). The data can be uploaded by a single API call, or simply referenced by a URL. This URL may be a landing page with further information or terms of use, or it may be an API call to large repositories of scientific data such as the SDSS (Szalay et al., 2002). OpenML will automatically version each dataset and make sure that empirical results are linked to specific versions. Authors can license their data and add citation requests. Finally, extra information can be added, such as the (default) target attribute(s) in labeled data, or the row-id attribute for data where instances are named.

Next, OpenML will compute an array of data characteristics, also called *metafeatures*, often categorized as either simple, statistical, information theoretic or landmarks (Pfahringer et al., 2000). Table 16.1 shows some of the metafeatures computed by OpenML.

<sup>4</sup><https://mlperf.org/>

<sup>5</sup><https://paperswithcode.com/>

Table 16.1: Standard metafeatures that are available in OpenML. Table taken from van Rijn (2016).

Category	Metafeatures
Simple	# Instances, # Attributes, # Classes, Dimensionality, Default Accuracy, # Observations with Missing Values, # Missing Values, % Observations with Missing Values, % Missing Values, # Numeric Attributes, # Nominal Attributes, # Binary Attributes, % Numeric Attributes, % Nominal Attributes, % Binary Attributes, Majority Class Size, % Majority Class, Minority Class Size, % Minority Class
Statistical	Mean of Means of Numeric Attributes, Mean Standard Deviation of Numeric Attributes, Mean Kurtosis of Numeric Attributes, Mean Skewness of Numeric Attributes
Information theoretic	Class Entropy, Mean Attribute Entropy, Mean Mutual Information, Equivalent Number of Attributes, Noise to Signal Ratio
Landmarkers	Accuracy of Decision Stump, Kappa of Decision Stump, Area under the ROC Curve of Decision Stump, Accuracy of Naive Bayes, Kappa of Naive Bayes, Area under the ROC Curve of Naive Bayes, Accuracy of $k$ -NN, Kappa of $k$ -NN, Area under the ROC Curve of $k$ -NN, ...

The datasets and their metadata can be retrieved via an online search engine, via the REST API as JSON and XML, and as native Python, R, or Java data structures using the corresponding APIs. This structured metadata includes user-provided descriptions, extracted attribution information, metafeatures, and even statistics of the data distribution.

### 16.3.2 Task types

A dataset alone does not constitute a scientific task. We must first agree on what types of results are expected to be shared. This is expressed in *task types*: they define what types of inputs are given, which types of output are expected to be returned, and what scientific protocols should be used. For instance, classification tasks should include well-defined cross-validation procedures and labeled input data and require predictions as outputs.

OpenML currently covers supervised classification, supervised regression, clustering, learning curve analysis, data stream classification, survival analysis, and subgroup discovery. These are very generally defined: a classification can cover text, image, or any other type of classification. Each task types comes with information on how models should be trained and evaluated, such as predefined cross-validation splits for classification, and prequential (test-then-train) splits for data streams.

### 16.3.3 Tasks

Tasks are instantiations of task types with specific inputs. An example of such a task is shown in Figure 16.1. In this case, it is a classification task defined on the MNIST dataset (version 1). Next to the dataset, the task includes the target attribute and the evaluation

## Given inputs

source_data	anneal (1)	Dataset (required)
estimation_procedure	10-fold Cross-validation	EstimationProcedure (required)
evaluation_measures	predictive_accuracy	String (optional)
target_feature	class	String (required)
data_splits	<a href="http://www.openml.org/api_splits/get/1/1/Task_1_splits.arff">http://www.openml.org/api_splits/get/1/1/Task_1_splits.arff</a>	TrainTestSplits (hidden)

## Expected outputs

model	A file containing the model built on all the input data	File (optional)
evaluations	A list of user-defined evaluations of the task as key-value pairs	KeyValue (optional)
predictions	An arff file with the predictions of a model	Predictions (required)

Fig. 16.1: Example of an OpenML task description

procedure (here, 10-fold cross-validation) used to generate the train and test splits. The required outputs for this task are the predictions for all test instances and, optionally, the models built and evaluations calculated by the user. OpenML will always compute a large range of evaluation measures on the server to ensure objective comparison. The preferred evaluation measure can be selected afterwards depending on the type of meta-level analysis.

Tasks can again be viewed or downloaded through the website, REST API, or language-specific APIs, including a list of all algorithms trained on that task and their evaluations. These can then be reused in different metalearning applications.





### 16.3.4 Flows

*Flows* are implementations of single algorithms, workflows (also known as pipelines), or scripts designed to solve a given task. For actual pipelines, the flows will define the exact components of the pipelines and how they fit together, as well as details about each component, such as the list of hyperparameters that could be tuned. For each hyperparameter, the name, description, and default value (if known) are stored. OpenML also stores metadata such as the dependencies and citation information, to enable the flows to be rebuilt and reused later on.

Flows can be updated as often as needed. OpenML will version each uploaded flow, while users can provide their own version name for reference. As with datasets, each flow has its own page which combines all known information and all results obtained by running the flow on OpenML tasks (see Figure 16.2).

It is important to emphasize that flows are typically not executed on the OpenML server. They can be executed locally or on any computer server the user has access to.

 moa.HoeffdingTree

 Visibility: public 
  Uploaded on 24-06-2014 by [Jan van Rijn](#)
 Moa.2014.03 
  270 runs

A Hoeffding tree (VFDT) is an incremental, anytime decision tree induction algorithm that is capable of learning from massive data streams, assuming that the distribution generating examples does not change over time. Hoeffding trees exploit the fact that a small sample can often be enough to choose an optimal splitting attribute. This idea is supported mathematically by the Hoeffding bound, which quantifies the number of observations (in our case, examples) needed to estimate some statistics within a prescribed precision (in our case, the goodness of an attribute).

Please cite: Geoff Hulten, Laurie Spencer, Pedro Domingos: Mining time-changing data streams. In: ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining, 97–106, 2001

## Parameters

b	binarySplits: Only allow binary splits.	default: false
c	splitConfidence: The allowable error in split decision, values closer to 0 will take longer to decide.	default: 1.0E-7
e	memoryEstimatePeriod: How many instances between memory consumption checks.	default: 1000000
g	gracePeriod: The number of instances a leaf should observe between split attempts.	default: 200
l	leafprediction: Leaf prediction to use.	default: NBAdaptive
m	maxByteSize: Maximum memory consumed by the tree.	default: 33554432
p	noPrePrune: Disable pre-pruning.	default: false
q	nbThreshold: The number of instances a leaf should observe before permitting Naive Bayes.	default: 0
r	removePoorAtts: Disable poor attributes.	default: false
s	splitCriterion: Split criterion to use.	default: InfoGainSplitCriterion
t	tieThreshold: Threshold below which a split will be forced to break ties.	default: 0.05
z	stopMemManagement: Stop growing as soon as memory limit is hit.	default: false

Fig. 16.2: Example of an OpenML flow

### 16.3.5 Setups

A setup is the combination of a flow and a certain configuration of the hyperparameters. Setups allow for analysis on the effect of hyperparameters, as is shown in Chapter 17. Setups that are run with default parameter settings are flagged as such.

### 16.3.6 Runs

Runs are applications of flows on a specific task. They are submitted by uploading the required outputs (e.g., predictions) together with the task id, the flow id, and any parameter settings. Each run also has its own page with all details and results, shown partially

in Figure 16.3. In this case, it is a classification run, where the predictions of the specific task are uploaded, and the evaluation measures are calculated on the server. Based on the parameter settings, the run is also linked to a setup.

OpenML calculates detailed evaluation results, including per-fold predictions. For class-specific measures such as area under the ROC curve, precision, and recall, per-class results are stored. Additional information, such as run times and details on hardware, can be provided by the user.

Because each run is linked to a specific task, flow, setup, and author, OpenML can filter, aggregate, and visualize results accordingly. Depending on the metalearning application, different meta-datasets can be built, including the dataset metafeatures, flow hyperparameter settings, and run evaluation results.

### 16.3.7 Studies and benchmark suites

Finally, OpenML allows sets of tasks and runs to be bundled. This makes it easier to define specific sets of tasks for a specific goal, and to group a specific set of runs (and their evaluation results) to obtain a fixed set of metadata for subsequent analysis. A set of tasks also implies a specific set of underlying datasets, and a set of runs implies a specific set of flows and tasks.

A special use case for a set of tasks is a *benchmark suite* (Bischl et al., 2021), a set of tasks that are selected to evaluate algorithms under a precisely specified set of conditions. They allow experiments run on these datasets to be clearly interpretable, comparable, and reproducible. Benchmarking suites can be created and retrieved using the existing OpenML interfaces and APIs. A first example of such a suite is the OpenML-CC18 (Bischl et al., 2021). Subsequently, these tasks could be easily downloaded and then classifiers could be run on them using different libraries, including Scikit-learn (Pedregosa et al., 2011), mlr (Bischl et al., 2016a), and Weka (Hall et al., 2009), through existing OpenML bindings (Casalicchio et al., 2017; van Rijn et al., 2015; Feurer et al., 2019a).

### 16.3.8 Integrations of OpenML in machine learning environments

OpenML is integrated into several popular machine learning environments, so that it can be used out of the box. These integrations are offered either as libraries (e.g., R or Python packages) or as plugins for existing toolboxes.

Figure 16.4 shows how OpenML is integrated in WEKA's Experimenter (Hall et al., 2009). After selecting OpenML as the result destination and providing login credentials, a number of tasks can be added through a dialogue. The plug-in supports the use of filters (for preprocessing operations), uploading of parameter sweep traces (for parameter optimization), and uploading of human-readable model representations produced by WEKA.

Other integrations include MOA (Bifet et al., 2010a), for running experiments and doing metalearning on data streams (Bifet et al., 2010b; Read et al., 2012), and Rapid-Miner (Hofmann and Klinkenberg, 2013), for running complex workflows (van Rijn and Vanschoren, 2015).

Researchers who use R or Python can use the *openml* package, provided on central repository databases CRAN and PyPI. Figure 16.5 is an example showing how to download a task and upload a run in R (together with all metadata). Once a classifier is created (line 3), it takes two function calls to download the task into memory (line 4) and run

## ★ Run 24996

🔧 Task 59 (Supervised Classification) 📄 Iris 📁 Uploaded on 13-08-2014 by [Jan van Rijn](#)





## Flow

weka.J48 Ross Quinlan (1993). C4.5: Programs for Machine Learning.

weka.J48\_C 0.25

weka.J48\_M 2

## Result files

 Description	XML file describing the run, including user-defined evaluation measures.	xml
 Model readable	A human-readable description of the model that was built.	model
 Model serialized	A serialized description of the model that can be read by the tool that generated it.	model
 Predictions	ARFF file with instance-level predictions generated by the model.	arff

## Evaluations

	0.9565 ± 0.0516			
Area under the roc curve	Iris-setosa	Iris-versicolor	Iris-virginica	
	0.98	0.9408	0.9488	
	actual \ predicted	Iris-setosa	Iris-versicolor	Iris-virginica
Confusion matrix	Iris-setosa	48	2	0
	Iris-versicolor	0	47	3
	Iris-virginica	0	3	47
	0.9479 ± 0.0496			
Precision	Iris-setosa	Iris-versicolor	Iris-virginica	
	1	0.9038	0.94	
Predictive accuracy	0.9467 ± 0.0653			
	0.9467 ± 0.0653			
Recall	Iris-setosa	Iris-versicolor	Iris-virginica	
	0.96	0.94	0.94	

Fig. 16.3: Example of an OpenML run



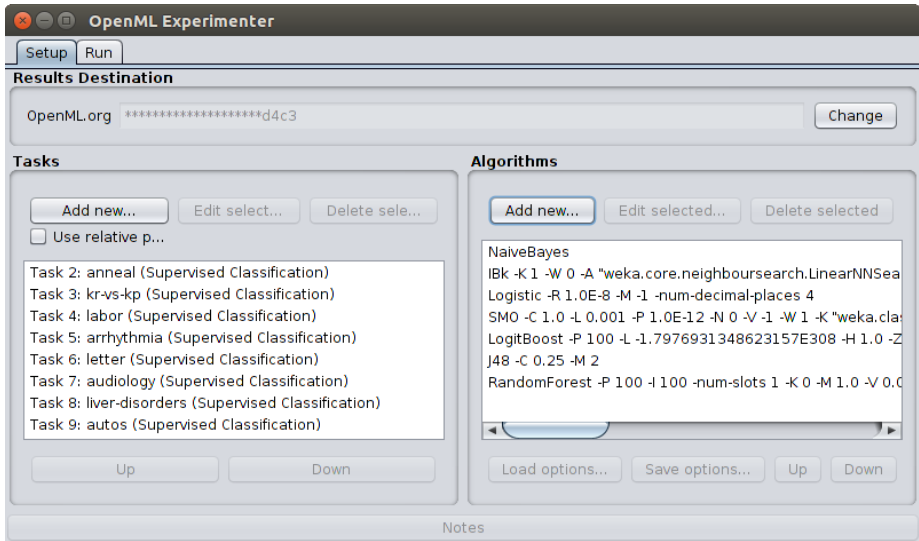


Fig. 16.4: WEKA integration of OpenML

the classifier (line 5). After the classifier has been applied, it takes a single function call to upload all the results (line 6).

```

1 library(mlr)
2 library(OpenML)
3 lrn = makeLearner("classif.randomForest")
4 task = getOMLTask(6)
5 run = runTaskMlr(task, lrn)
6 uploadOMLRun(run)

```

Fig. 16.5: R code to run a random forest classifier as implemented in mlr on the letter task (task id = 6)

The Python code is very similar (see Figure 16.6) and works very similarly to the R code. The function calls `get_task()` and `run_model_on_task` are used to download the task into memory (line 4) and run the classifier on the task (line 5). Finally, the member function `run.publish` uploads the results to OpenML (line 6).

As usual, the Java code is a bit more verbose (see Figure 16.7). For brevity, the imports in the header have been omitted. The Java function `executeTask` runs the classifier on the task and automatically uploads each executed run to the server (line 8).

These APIs also allow for convenient downloading of all results in various formats.

```

1 from sklearn import ensemble
2 from openml import tasks, runs
3 clf = ensemble.RandomForestClassifier()
4 task = tasks.get_task(6)
5 run = runs.run_model_on_task(clf, task)
6 run.publish()

```

Fig. 16.6: Python code to run a random forest classifier as implemented in Scikit-learn on the letter task (task id = 6)

```

1 public static void runTasksAndUpload() throws Exception {
2     OpenmlConnector openml = new OpenmlConnector();
3     openml.setApiKey("FILL_IN_OPENML_API_KEY");
4     Classifier forest = new RandomForest();
5     Task task = openml.taskGet(6);
6     Instances d = InstancesHelper.getDatasetFromTask(openml, task
7         );
8     Pair<Integer, Run> result = RunOpenmlJob.executeTask(
9         openml, new WekaConfig(), task.getTask_id(), forest);
10    Run run = openml.runGet(result.getLeft());
11 }

```

Fig. 16.7: Java code to run a random forest classifier as implemented in Weka on the letter task (task id = 6)

### Example of one study exploiting existing evaluation results

Figure 16.8 is an example of Python code showing how detailed evaluation results can be downloaded to study the effect of hyperparameters of the SVM algorithm. The resulting plots are shown in Figure 16.9.

In the next chapter, we explore how the experimental data available in OpenML can be used to gain new insights into the relationship between properties of data, workflows, and performance.

## References

- Aha, D. W. (1992). Generalizing from case studies: A case study. In Sleeman, D. and Edwards, P., editors, *Proceedings of the Ninth International Workshop on Machine Learning (ML92)*, pages 1–10. Morgan Kaufmann.
- Alcala, J., Fernandez, A., Luengo, J., Derrac, J., Garcia, S., Sanchez, L., and Herrera, F. (2010). Keel datamining software tool: Data set repository, integration of algorithms and experimental analysis framework. *Journal of Multiple-Valued Logic and Soft Computing*, 17(2-3):255–287.
- Bergstra, J., Pinto, N., and Cox, D. (2015). SkData: data sets and algorithm evaluation protocols in Python. *Computational Science & Discovery*, 8(1).

```

1 import openml;
2 import numpy as np
3 import matplotlib.pyplot as plt
4 df = openml.evaluations.list_evaluations_setups(
5     'predictive_accuracy', flow=[8353], task=[6],
6     output_format='dataframe',
7     parameters_in_separate_columns=True,
8 ) # Choose SVM flow (e.g. 8353) and dataset 'letter' (task 6).
9 hp_names = ['sklearn.svm.classes.SVC(16)_C', 'sklearn.svm.
10             classes.SVC(16)_gamma']
11 df[hp_names] = df[hp_names].astype(float).apply(np.log)
12 C, gamma, score = df[hp_names[0]], df[hp_names[1]], df['value']
13 cntnr = plt.tricontourf(
14     C, gamma, score, levels=12, cmap='RdBu_r')
15 plt.colorbar(cntnr, label='accuracy')
16 plt.xlim((min(C), max(C))); plt.ylim((min(gamma), max(gamma)))
17 plt.xlabel('C (log10)', size=16);
18 plt.ylabel('gamma (log10)', size=16)
19 plt.title('SVM performance landscape', size=20)

```

Fig. 16.8: Code for retrieving the predictive accuracy of a SVM classifier on the “letter” dataset and creating a contour plot (retrieved from Feurer et al. (2019b))

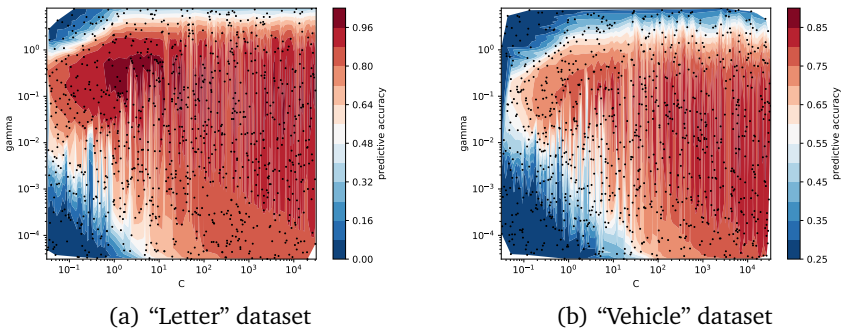


Fig. 16.9: Surface plots of the  $\gamma$  and  $C$  hyperparameter of SVM classifier. Both axes show the value of a hyperparameter, and the color of the grid shows how well a certain configuration performed (retrieved from Feurer et al. (2019b))

Bifet, A., Holmes, G., Kirkby, R., and Pfahringer, B. (2010a). MOA: Massive Online Analysis. *J. Mach. Learn. Res.*, 11:1601–1604.

- Bifet, A., Holmes, G., and Pfahringer, B. (2010b). Leveraging Bagging for Evolving Data Streams. In *Machine Learning and Knowledge Discovery in Databases*, volume 6321 of *Lecture Notes in Computer Science*, pages 135–150. Springer.
- Bischl, B., Casalicchio, G., Feurer, M., Gijsbers, P., Hutter, F., Lang, M., Mantovani, R. G., van Rijn, J. N., and Vanschoren, J. (2021). OpenML benchmarking suites. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, NIPS'21.
- Bischl, B., Kerschke, P., Kotthoff, L., Lindauer, M., Malitsky, Y., Fréchet, A., Hoos, H., Hutter, F., Leyton-Brown, K., Tierney, K., et al. (2016a). ASlib: A benchmark library for algorithm selection. *Artificial Intelligence*, 237:41–58.
- Bischl, B., Lang, M., Kotthoff, L., Schiffner, J., Richter, J., Studerus, E., Casalicchio, G., and Jones, Z. M. (2016b). mlr: Machine Learning in R. *Journal of Machine Learning Research*, 17(170):1–5.
- Brazdil, P., Giraud-Carrier, C., Soares, C., and Vilalta, R. (2009). *Metalearning: Applications to data mining*. Springer.
- Brockman, G., Cheung, V., Petteersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). OpenAI Gym. *arXiv:1606.01540*.
- Buitinck, L., Louppe, G., Blondel, M., Pedregosa, F., Mueller, A., Grisel, O., Niculae, V., Prettenhofer, P., Gramfort, A., Grobler, J., Layton, R., VanderPlas, J., Joly, A., Holt, B., and Varoquaux, G. (2013). API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122.
- Casalicchio, G., Bossek, J., Lang, M., Kirchhoff, D., Kerschke, P., Hofner, B., Seibold, H., Vanschoren, J., and Bischl, B. (2017). OpenML: An R package to connect to the machine learning platform OpenML. *Computational Statistics*.
- Chang, C. C. and Lin, C. J. (2011). LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3):27.
- Chen, Y., Keogh, E., Hu, B., Begum, N., Bagnall, A., Mueen, A., and Batista, G. (2015). The UCR time series classification archive. [www.cs.ucr.edu/~eamonn/time\\_series\\_data/](http://www.cs.ucr.edu/~eamonn/time_series_data/).
- Dheeru, D. and Taniskidou, E. K. (2017). UCI machine learning repository.
- Feurer, M., Klein, A., Eggenberger, K., Springenberg, J. T., Blum, M., and Hutter, F. (2019a). Auto-sklearn: Efficient and robust automated machine learning. In Hutter, F., Kotthoff, L., and Vanschoren, J., editors, *Automated Machine Learning: Methods, Systems, Challenges*, pages 113–134. Springer.
- Feurer, M., van Rijn, J. N., Kadra, A., Gijsbers, P., Mallik, N., Ravi, S., Müller, A., Vanschoren, J., and Hutter, F. (2019b). OpenML-Python: an extensible Python API for OpenML. *arXiv preprint arXiv:1911.02490*.
- Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The WEKA Data Mining Software: An Update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18.
- Hand, D. (2006). Classifier technology and the illusion of progress. *Statistical Science*, 21(1):1–14.
- Hirsh, H. (2008). Data mining research: Current status and future opportunities. *Statistical Analysis and Data Mining*, 1(2):104–107.
- Hofmann, M. and Klinkenberg, R. (2013). *RapidMiner: Data Mining Use Cases and Business Analytics Applications*. Data Mining and Knowledge Discovery. Chapman & Hall/CRC.

- Hoste, V. and Daelemans, W. (2005). Comparing learning approaches to coreference resolution. There is more to it than bias. In Giraud-Carrier, C., Vilalta, R., and Brazdil, P., editors, *Proceedings of the ICML 2005 Workshop on Meta-Learning*, pages 20–27.
- Hutson, M. (2018). Missing data hinder replication of artificial intelligence studies. *Science*.
- Keogh, E. and Kasetty, S. (2003). On the need for time series data mining benchmarks: A survey and empirical demonstration. *Data Mining and Knowledge Discovery*, 7(4):349–371.
- Michie, D., Spiegelhalter, D. J., and Taylor, C. C. (1994). *Machine Learning, Neural and Statistical Classification*. Ellis Horwood.
- Olson, R. S., La Cava, W., Orzechowski, P., Urbanowicz, R. J., and Moore, J. H. (2017). PMLB: A large benchmark suite for machine learning evaluation and comparison. *BioData Mining*, 10(36).
- Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., and Dubourg, V. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12(Oct):2825–2830.
- Perlich, C., Provost, F., and Simonoff, J. (2003). Tree induction vs. logistic regression: A learning-curve analysis. *Journal of Machine Learning Research*, 4:211–255.
- Pfahring, B., Bensusan, H., and Giraud-Carrier, C. (2000). Meta-learning by landmarking various learning algorithms. In Langley, P., editor, *Proceedings of the 17th International Conference on Machine Learning, ICML'00*, pages 743–750.
- Read, J., Bifet, A., Pfahring, B., and Holmes, G. (2012). Batch-Incremental versus Instance-Incremental Learning in Dynamic and Evolving Data. In *Advances in Intelligent Data Analysis XI*, pages 313–323. Springer.
- Sculley, D., Snoek, J., Wiltschko, A., and Rahimi, A. (2018). Winner’s curse? on pace, progress, and empirical rigor. In *Workshop of the International Conference on Representation Learning (ICLR)*.
- Szalay, A. S., Gray, J., Thakar, A. R., Kunszt, P. Z., Malik, T., Raddick, J., Stoughton, C., and vandenBerg, J. (2002). The SDSS SkyServer: public access to the Sloan digital sky server data. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 570–581. ACM.
- Tsoumakas, G., Spyromitros-Xioufis, E., Vilcek, J., and Vlahavas, I. (2011). MULAN: A Java library for multi-label learning. *JMLR*, pages 2411–2414.
- van Rijn, J. N. (2016). *Massively collaborative machine learning*. PhD thesis, Leiden University.
- van Rijn, J. N., Holmes, G., Pfahring, B., and Vanschoren, J. (2015). Having a Blast: Meta-Learning and Heterogeneous Ensembles for Data Streams. In *2015 IEEE International Conference on Data Mining (ICDM)*, pages 1003–1008. IEEE.
- van Rijn, J. N. and Vanschoren, J. (2015). Sharing RapidMiner workflows and experiments with OpenML. In Vanschoren, J., Brazdil, P., Giraud-Carrier, C., and Kotthoff, L., editors, *Proceedings of the 2015 International Workshop on Meta-Learning and Algorithm Selection (MetaSel)*, number 1455 in CEUR Workshop Proceedings, pages 93–103.
- Vanschoren, J., Blockeel, H., Pfahring, B., and Holmes, G. (2012). Experiment databases: a new way to share, organize and learn from experiments. *Machine Learning*, 87(2):127–158.
- Vanschoren, J., van Rijn, J. N., Bischl, B., and Torgo, L. (2014). OpenML: networked science in machine learning. *ACM SIGKDD Explorations Newsletter*, 15(2):49–60.



## Learning from Metadata in Repositories

**Summary.** This chapter describes the various types of experiments that can be done with the vast amount of data, stored in experiment databases. We focus on three types of experiments done with the data stored in OpenML. First, we describe experiments that determine how a certain algorithm works on a given dataset. We discuss experiments that show which algorithm performs best on a given dataset and experiments determining the effect of a given hyperparameter on the performance. Second, we describe experiments that determine how a certain algorithm works across datasets. We discuss experiments that determine the statistical significance of differences of performance, the effects of hyperparameter optimization of chosen algorithms across datasets, and experiments that determine which algorithms generally make similar predictions. Finally, we describe experiments that determine the effect of certain data or workflow characteristics on the performance. By taking into consideration the various metafeatures that have been calculated, we can explore trends when a given type of algorithm (e.g., linear models, models including feature selection) performs better than another.

### 17.1 Introduction

OpenML described in the previous chapter (Chapter 16) includes information about a vast set of experiments, which have been collected and organized in a structured way. This allows to explore the data to conduct various studies and this way gain useful knowledge about how algorithms behave. As Vanschoren et al. (2012) pointed out, the studies can be divided into the following groups, depending on what the overall aim is:

- Model-level analysis, whose main aim is to analyze *how* a given algorithm performs
- Data-level analysis, whose aim is to investigate *when* an algorithm performs well
- Method-level analysis, whose aim is to determine *why* an algorithm performs the way it does

This chapter follows loosely the groups detailed above. Section 17.2 describes how the performance of different algorithms or their configurations varies on some datasets. Section 17.3 extends this study to determine how the performance varies across datasets. Section 17.4 presents studies whose aim is to investigate the interplay between the performance of algorithms and measurable features (e.g., algorithm properties or metafeatures) across datasets.





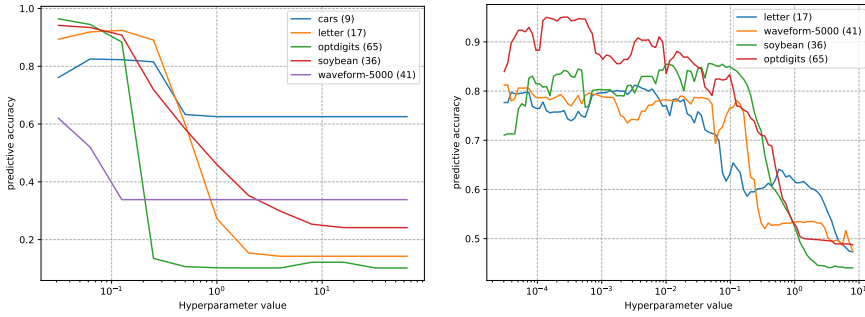


Fig. 17.2: The effect of varying SVM Fig. 17.3: Marginal of the SVM gamma hyperparameter

the performance of some algorithms (flows) (e.g., weka.SMO(RBFKernel)) has a great variation, as the hyperparameter settings for this flow span across a wide range.

### 17.2.2 Effect of varying some hyperparameter settings

In this section we present a study of the effects of varying certain hyperparameter settings on performance. We focus on the effects of the *gamma* parameter of SVM with the RBF kernel, as implemented in Weka. All the other hyperparameters were left with their respective default settings.

Figure 17.2 shows these effects for some datasets only. As we can see, the performance curves follow a somewhat similar trend for some datasets. In the case of “waveform”, “soybean”, and “optdigits”, the performance degrades to the default accuracy. In the case of “letter” and “car”, the performance first increases until it has reached an optimum and then degrades to the default accuracy. The optimal value of the parameter is different for all datasets used in this study, as would be expected. It seems that setting this value too low is less harmful for performance than setting it too high.

This study shows a local effect of how a particular hyperparameter affects performance under the assumption that all other hyperparameters use a fixed setting (default values).

### Global effect of a hyperparameter, based on a marginal

A *marginal* defines, for each hyperparameter, the expected performance when all other hyperparameters have been averaged across all possible values. Although this seems infeasible to compute, Hutter et al. (2014) showed that it can be computed efficiently by using tree-based *surrogate models*.

Figure 17.3 plots the marginal of the gamma parameter of support vector machines with the RBF kernel, as implemented in Scikit-learn, on several datasets. Note that the values of the marginal are lower than the values in the plot that shows the local effect. This is because these are averaged across all possible values for the other hyperparameters, which likely involve also many suboptimal values. This shows a global effect of how the hyperparameter gamma affects performance. Using the marginal solves the question regarding which value the other hyperparameters should be set to.

In Chapter 8, we discussed the work of van Rijn and Hutter (2018), who showed how one can use the marginal to determine which hyperparameters are important to optimize.

## 17.3 Performance Analysis of Algorithms across Datasets

In this section we discuss experiments whose aim is to analyze how the performance of different algorithms (and/or configurations) varies across different datasets. We discuss three types of experiments. The first one concerns the benchmarking of algorithms, the second one studies the effect of hyperparameter optimization, and the third type analyzes how several algorithms differ in predictions.

### 17.3.1 Effect of using different classifiers with default hyperparameters

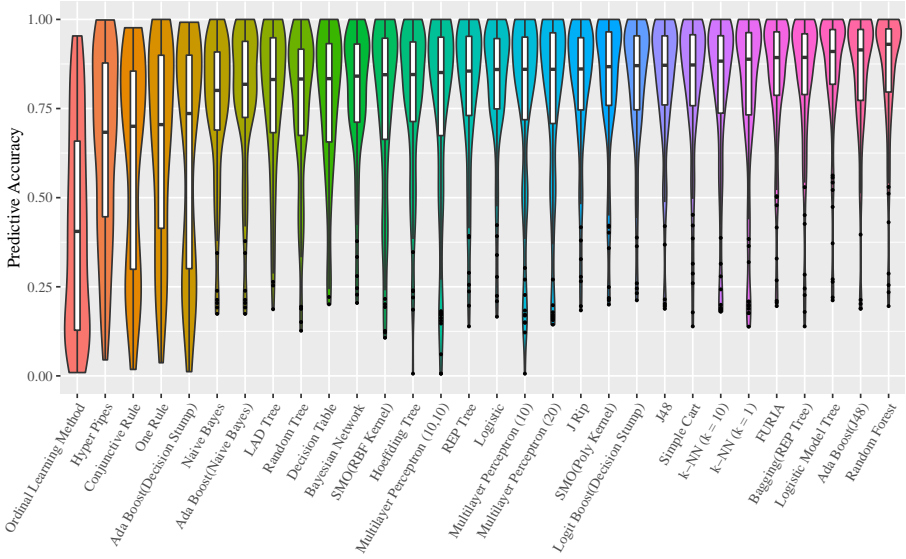
In order to assess the performance of an algorithm in a robust way, the benchmarking and evaluation should involve a large number of datasets. In this section we discuss a study to illustrate this. We have selected a set of classifiers and datasets from OpenML and the corresponding metadata, which includes the predictive accuracy and area under the ROC curve (AUC).

As the performance of a particular algorithm varies on different datasets, it is possible to elaborate violin plots, which are similar to box plots, except that they also show the probability density of the data at different values, smoothed by a kernel density estimator. Figure 17.4 shows violin plots (with integrated box plots) of the results of various Weka classifiers with default hyperparameter settings across 105 datasets. The classifiers are sorted by the median. Classifiers to the right perform generally better than classifiers to the left. Random forest (Breiman, 2001) performs best on average on this set of datasets. Some other ensemble methods perform well, e.g., adaptive boosting (Freund and Schapire, 1996) and logistic boosting (Friedman et al., 1998). Logistic model tree (LMT) (Landwehr et al., 2005) (which is a combination of trees and logistic regression) also performs reasonably well.

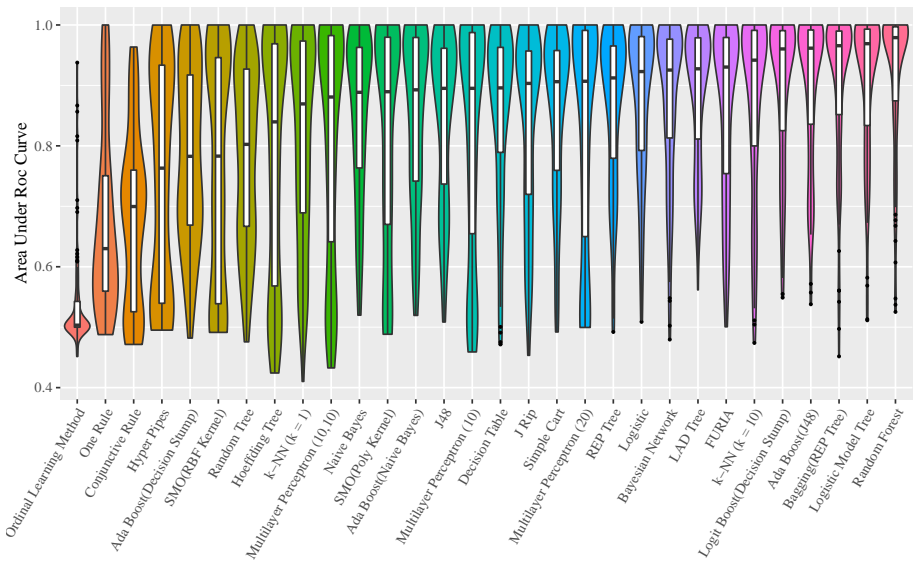
Note that, when a classifier is ranked low, it does not necessarily mean that it is a badly performing classifier overall. It might very well be that the classifier is specialized on a certain type of dataset, or just has hyperparameters that need to be tuned. For example, we note that the support vector machine with the RBF kernel is not well positioned in this ranking. However, as we saw in Figure 17.1, some variants of this classifier (variants with particular hyperparameter settings) are among the best performing ones on the “letter” dataset. As such, it is rare in modern applications to benchmark algorithms without applying proper hyperparameter optimization.

### Assessing statistical significance

To assess the statistical significance of the above results, we can use the Friedman test accompanied by the posthoc Nemenyi test, which were discussed in Chapter 3. Figure 17.5 shows the result of the Nemenyi test on the predictive accuracy of the classifiers from Figure 17.4. The classifiers are sorted by their average rank (lower is better). We see a similar ordering of classifiers as in Figure 17.4. Here, we also see that the logistic model tree and random forest perform best.



(a) Accuracy



(b) Area under the ROC curve

Fig. 17.4: Ranking of algorithms across a set of 105 datasets. This image was taken from van Rijn (2016)

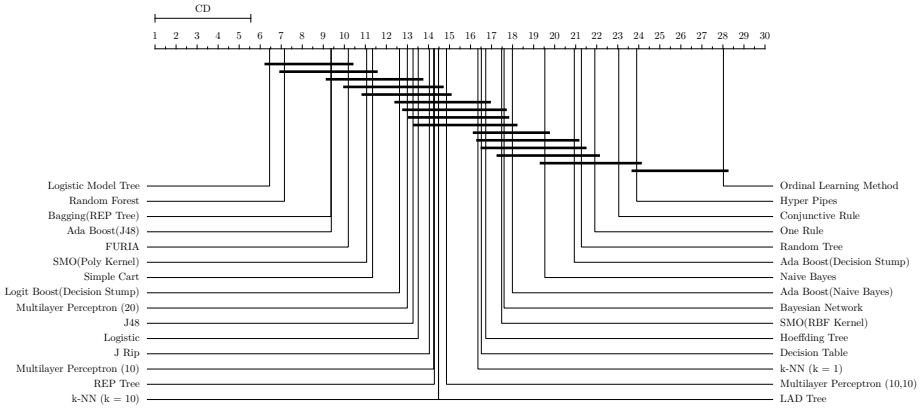


Fig. 17.5: Results of Nemenyi test ( $\alpha = 0.05$ ) on the predictive accuracy of classifiers in OpenML. This image was taken from van Rijn (2016)

Classifiers that are connected by a horizontal line are statistically equivalent. For instance, the figure shows that there is no statistical evidence that logistic model tree is better than FURIA. On the other hand, there is statistical evidence that the logistic model tree is better than the “Simple CART” algorithm.

When applied correctly, statistical tests give a more reliable assessment of the performance of algorithms than simply comparing performance values (see Chapter 3 for details).

### 17.3.2 Effect of hyperparameter optimization

It has been widely recognized that the performance of algorithms is affected heavily by hyperparameter optimization (Lavesson and Davidsson, 2006; Hutter et al., 2011; Bergstra and Bengio, 2012; Snoek et al., 2012; Domhan et al., 2015; Klein et al., 2017; Li et al., 2017; Thomas et al., 2018; Falkner et al., 2018). This section describes an experiment showing how OpenML can be exploited to investigate this relationship.

Different search strategies exist for identifying the best set of hyperparameter settings for a given algorithm. Several of those are discussed in Chapter 6.

In order to obtain an unbiased recommendation, it is common to follow the nested cross-validation procedure (see Chapter 3), which splits the data into a training set, validation set, and test set. The training set is used to train various variants with different hyperparameter settings. These variants are then evaluated on the validation set. The model that performs best on the validation set is recommended, and its performance is determined on the test set, representing an *unseen data sample*.

OpenML uses the notion of *tasks* to define the training set and test set; it is up to the user to split off a part from the training set into a validation set.

In the following illustrative example, we compare two Weka classifiers with optimized hyperparameters against the corresponding versions with default hyperparameters across different datasets. Figure 17.6 shows the results of a decision tree and a

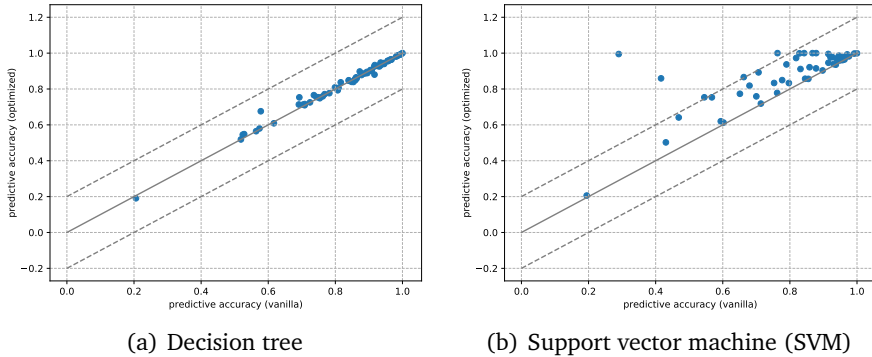


Fig. 17.6: Comparing performance of optimized hyperparameter of two classifiers against the default configuration

support vector machine (SVM) with the RBF kernel. For each dataset, both classifiers were applied twice; once with default hyperparameters and the second time with hyperparameter settings obtained by a hyperparameter optimization method (Weka's MultiSearch module). The results are presented in a scatter plot. Each dot represents the performance of both the default hyperparameters and optimized hyperparameters on a specific dataset. The performance of the default hyperparameters is displayed on the  $x$ -axis, and the performance of the optimized hyperparameters on the  $y$ -axis.

The diagonal solid line shows the break-even points; all points on this line represent an experiment for which hyperparameter optimization neither benefited, nor harmed the predictive performance. Each dot that is above (below) the solid line represents a dataset on which the version of the classifier with optimized hyperparameters performs better (worse) than the non-optimized version. Note that it is possible that hyperparameter optimization can deteriorate performance. The selection of a configuration is based on the performance of this configuration on the validation set; the selected configuration may not perform as well on the test set. However, this should not happen too often.

The plot reveals that support vector machines often benefit from hyperparameter optimization. For almost all datasets, the version with optimized hyperparameters outperforms the version with standard hyperparameters. For the decision tree algorithm this benefit is less apparent. The effect of this hyperparameter optimization method seems negligible on most datasets. We base this on the fact that most data points are scattered around the aforementioned diagonal line.

Most state-of-the-art algorithms, such as deep neural networks and gradient boosting, benefit from hyperparameter optimization. As such, in all realistic settings the question is usually not whether to use hyperparameter optimization, but rather which hyperparameter optimization technique to use.

### 17.3.3 Identifying algorithms (workflows) with similar predictions

This section describes a study whose aim is to identify classifiers with similar (or dissimilar) performance on individual examples. Identifying classifiers with similar performance is important, as it permits to substitute one classifier (which may be slow to train) by

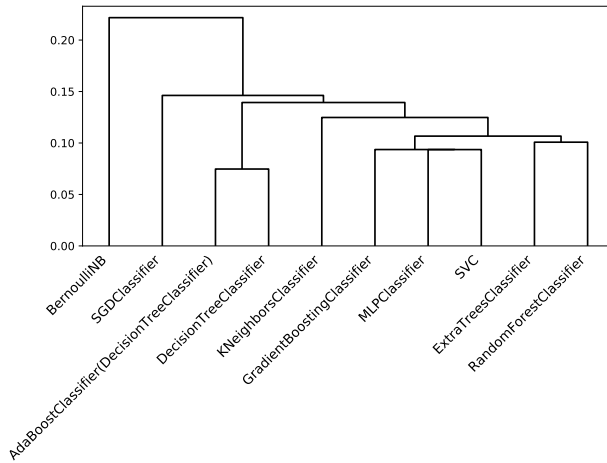


Fig. 17.7: Hierarchical clustering of Auto-sklearn classifiers

another. Identifying classifiers with different predictions is also important for another reason. As shown in Chapters 9 and 10, many ensembles (e.g., bagging ensembles) require a set of diverse classifiers. Simply assessing the performance of classifiers is not enough. Two classifiers can have similar performance but differ in some instances they are tested on.

The methodology adopted here is based on the proposal of Lee and Giraud-Carrier (2011) which used the *classifier output difference (COD)* metric, based on the difference in predictions between a pair of classifiers. The reader can consult Chapter 8 (Section 8.5) for details.

Hierarchical agglomerative clustering (HAC) converts this information into a hierarchical clustering. It starts by assigning each observation to its own cluster, and greedily joins the two clusters with the smallest distance (Rokach and Maimon, 2005). The *complete linkage* strategy is used to determine the distance between two clusters. Formally, the distance between two clusters  $A$  and  $B$  is defined as  $\max\{COD(a, b) : a \in A, b \in B\}$ .

Figure 17.7 shows the resulting dendrogram built for all classifiers from Auto-sklearn. This figure uses test results obtained on 45 datasets from the OpenML-CC18 benchmark suite (Bischl et al., 2021). It shows which classifiers make relatively similar predictions. Apparently, the Adaboost classifier and decision tree classifier make very similar predictions on this set of datasets, whereas the predictions of the Bernoulli naive Bayes classifier are rather different from the others. This could of course also happen when a classifier performs rather badly, while all the others have quite good performance.

## 17.4 Effect of Specific Data/Workflow Characteristics on Performance

In this section we present three experiments. In the first one, we investigate on what type of datasets non-linear models have an advantage over linear models. This study involves both algorithm properties (whether it produces a linear model or a non-linear model) and data characteristics. In the second study we investigate the effect of feature selection on algorithm performance. As in the first case, it involves algorithm properties and data characteristics. In the final experiment, we investigate the tunability of algorithms and the effect of optimizing a given hyperparameter.

### 17.4.1 Effect of selecting linear vs. non-linear models

In this section we show that previous experiments can be used to investigate the relationship between certain groups (types) of algorithms and performance.

Strang et al. (2018) conducted experiments with the objective of comparing performance results of linear and non-linear models. The results were optimized using 200 iterations of random search. The linear models included linear SVM, linear neural networks (NNs) (no hidden layer), and decision stumps; the non-linear ones included SVM with RBF kernel, NNs with hidden layers, and decision trees. Strang et al. (2018) aimed to investigate for which type of datasets the former are better than the latter.

Intuitively, non-linear models have the capacity to outperform linear models. However, linear models are still used in practice, since they are simpler, computationally more efficient, and easier to interpret than their non-linear counterparts.

Figure 17.8 shows the results of the experiment involving SVM and NNs. Each dot in this figure represents the experiment of a linear version and a non-linear version of a model on different datasets. Its position is determined by two dataset characteristics, namely *number of observations* ( $x$ -axis) and *number of features* ( $y$ -axis). The color of each dot shows which type of model was better on the respective dataset. Red (blue) color indicates that the non-linear model was significantly better (worse) than the linear one. Grey color is used for cases when the difference was not statistically significant according to the Nemenyi test.

The background color shows which type of classifier is dominant in the respective region, based on a  $k$ -nearest-neighbor model with  $k = 5$ .<sup>2</sup> Figure 17.8 suggests that the non-linear models are dominant in the red regions characterized by a large number of data points (instances). Linear models are seldom significantly better than their non-linear counterparts. There are many datasets on which the performance of the two types of models is comparable, according to a statistical test. These appear in the grey region that occupies a portion of the area.

Despite the low number of datasets on which a linear model is superior, the results show that linear models may still be useful in many situations. When the performance is comparable, one argument in their favor is that they are typically faster to train and simpler to analyze. An open question is whether the results would still hold if non-linear models with advanced regularization schemes were used.

---

<sup>2</sup>The shape of the background coloring is affected by the fact that it is determined in Euclidean space and represented in log space.

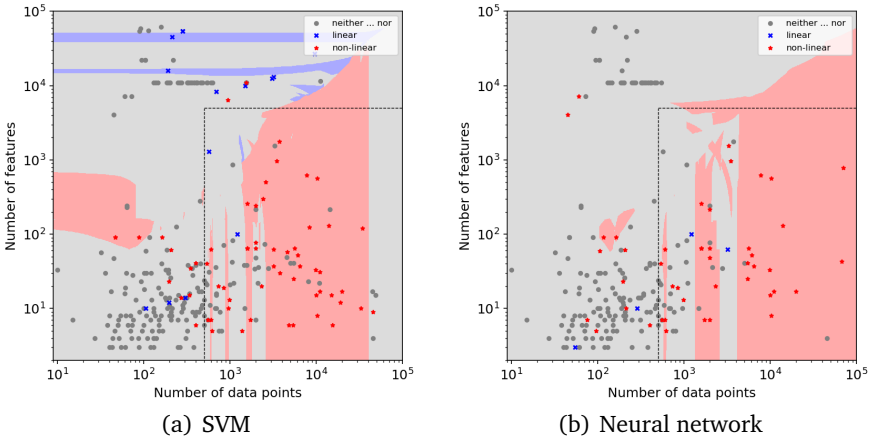


Fig. 17.8: Linear vs. non-linear models, plotted against two data characteristics. The image was taken from Strang et al. (2018)

### 17.4.2 Effect of employing feature selection

Data preprocessing is often considered an important factor that can affect the performance of classification algorithms. The experiments in OpenML can help us investigate specific questions related to preprocessing.

In this section we consider the following questions: *Does feature selection improve classification performance? If so, how do the classifier type and dataset properties influence this?*

This section is based upon the experiments presented by Post et al. (2016). For each dataset, the authors ran a classifier without feature selection and then repeated this with feature selection. There are many different feature selection methods. In this experiment, *correlation-based feature subset selection* was used. This method tries to identify features that are highly correlated with the target attribute and uncorrelated with each other (Hall, 1999).

The results are shown in the form of a scatter plot, similar to the one in the previous section. In Figure 17.9 each dot represents a dataset used in the experiment. Its position is determined by the number of features (attributes) of that dataset (*x*-axis) and the corresponding number of instances (*y*-axis). The color of the dot shows whether feature selection yielded better or worse result. Green (red) color is used when the performance with feature selection is better (worse) than without feature selection. Blue color is used when the performance is not significantly different.

These plots show some expected behavior, as well as some interesting patterns. First of all, they reveal that feature selection is most beneficial for methods such as *k*-NN and naive Bayes. This is what one might expect: due to the curse of dimensionality, the nearest-neighbor methods can suffer from too many attributes (Radovanović et al., 2010) and naive Bayes is vulnerable to correlated features (John and Langley, 1995).



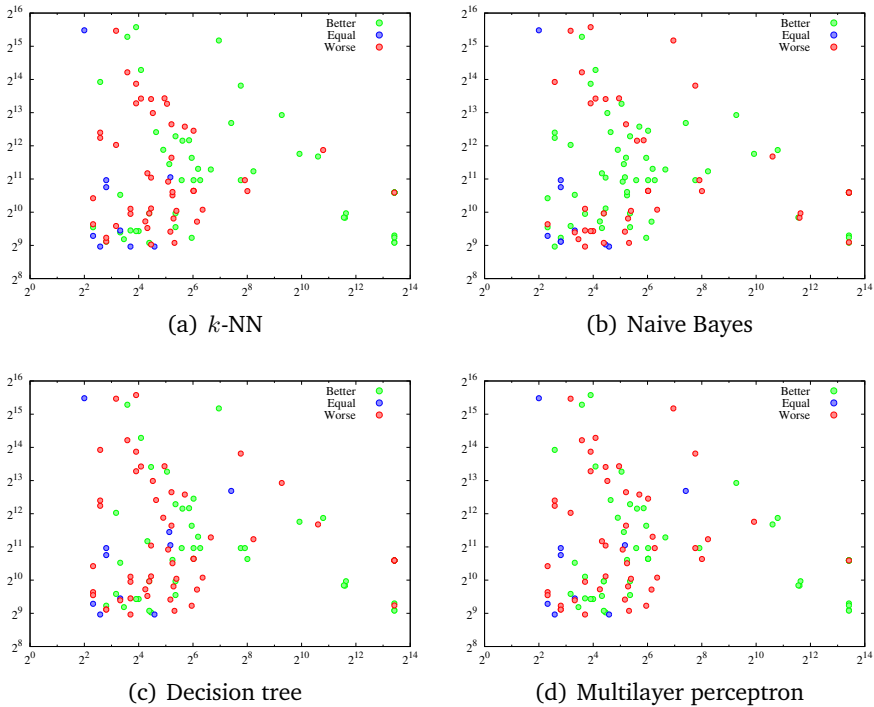


Fig. 17.9: The effect of feature selection on classifier performance and its dependence on the number of features ( $x$ -axis) and number of instances ( $y$ -axis). This image was taken from van Rijn (2016)

Also, this study confirms that, when using  $k$ -NN, feature selection yields good results on datasets with many features (Post et al., 2016).

We also note some unexpected behavior. For example, it has been noted that some tree induction algorithms have built-in protection against irrelevant features (Quinlan, 1986). However, Figure 17.9 shows that, in many cases, feature selection is still beneficial. Also, the multilayer perceptron model is considered to be capable of selecting relevant features. As the figure shows, the situation is not so clear-cut.

Altogether, it is hard to draw clear conclusions regarding the correlation between the aforementioned dataset characteristics (metafeatures) and performance. It is conceivable that more complex models capable of predicting when to use feature selection or not can be developed (Post et al., 2016).

### 17.4.3 Effect of specific hyperparameter settings

As has been shown in various other chapters of this book, metalearning and AutoML systems explore a set of pre-specified alternatives in the process of elaborating a specific solution for the target task. As was shown in Chapter 8, which discussed *configuration*

spaces, the alternatives typically include different algorithms, the associated hyperparameters, and, for each one, the possible values or ranges.

In this section we focus first on some specific algorithms and investigate whether their performance can be improved by tuning their hyperparameters. In our second study, we consider different hyperparameters of some specific algorithms and investigate which of these hyperparameters have the most significant effect on performance. The answers to these questions facilitate the process of providing good recommendations, as it is possible to focus on those options that actually matter and disregard the ones that are irrelevant.

## Tunability across algorithms

Probst et al. (2019) define the notion of *tunability* across algorithms, which is defined for each algorithm as the difference between the default hyperparameters and the best found hyperparameters for a given dataset. Figure 17.10(a) shows these results. This study confirms the common belief that tuning the hyperparameters of SVM is indeed of importance.

## Tunability across hyperparameters

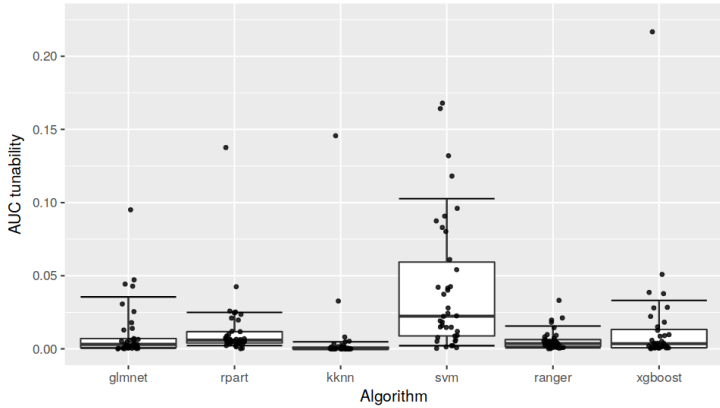
Probst et al. (2019) define also tunability across different hyperparameters of a particular algorithm. This is done by comparing the performance of the default hyperparameters against the performance of the algorithm with a *single hyperparameter* optimized. Figure 17.10(b) shows these results. Based on this figure, one could conjecture that the hyperparameters “mtry” and “samples.fraction” have the greatest influence on performance.

## Hyperparameter importance across datasets based on functional ANOVA

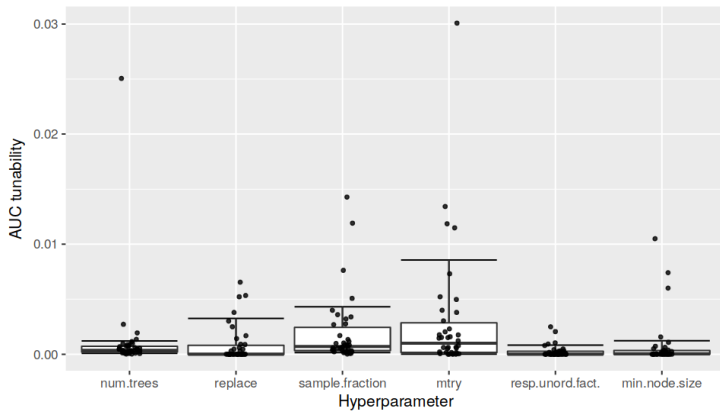
Functional ANOVA was discussed in Chapter 8 (Section 8.4). It decomposes the variance of the general model into additive components relative to a given set of hyperparameters (Hutter et al., 2014; van Rijn and Hutter, 2018). For each algorithm, a *marginal* can be calculated for each hyperparameter (or a combination of hyperparameters).<sup>3</sup> The importance of a given hyperparameter (or combination of hyperparameters) is related to this marginal. The higher the value, the more important the hyperparameter. Figure 17.11 shows the results for the Scikit-learn implementations of random forests, Adaboost, and support vector machine (RBF kernel).<sup>4</sup> The results of Figure 17.11(a) agree, to a certain level, with the results from Figure 17.10(b). In these comparisons we have to take into account that some hyperparameter names do not always agree across toolboxes. For instance, the mlR hyperparameter “mtry” is the same as the Scikit-learn hyperparameter “max features”. Both methods discussed in this section identified this hyperparameter as important. There is also a disagreement in relation to some hyperparameters. For example, Scikit-learn identifies “min. samples leaf” as the most important hyperparameter, whereas mlR does not define it. van Rijn and Hutter (2018) speculate that this is because “min. samples leaf” is not as important as it seems, since it has a strong default

<sup>3</sup>The concept of *marginal* was also exploited in Section 17.2.2.

<sup>4</sup>Sharma et al. (2019) apply the same methodology on residual neural networks for image classification.



(a) Tunability across algorithms

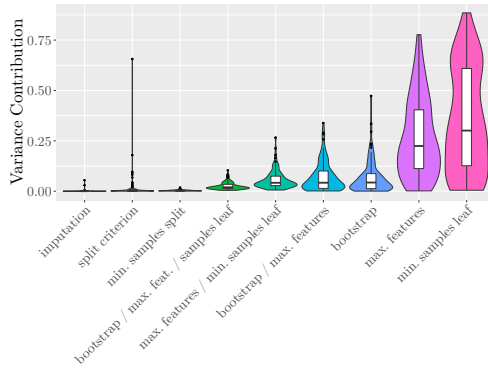


(b) Tunability across random forest hyperparameters

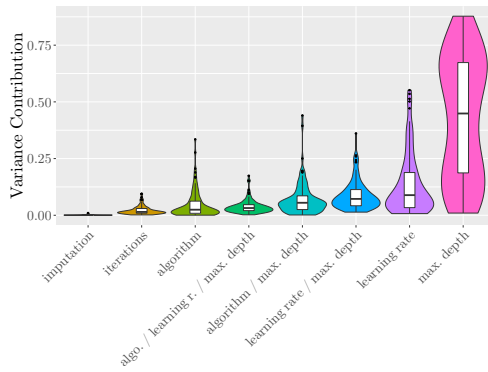
Fig. 17.10: Tunability results. The images were taken from Probst et al. (2019)

value that performs well across datasets. Furthermore, Probst et al. (2019) identify the hyperparameter “samples.fraction” as important, whereas the most closely related hyperparameter in Scikit-learn, “bootstrap”, seems rather unimportant. Further investigation is required to completely understand this.

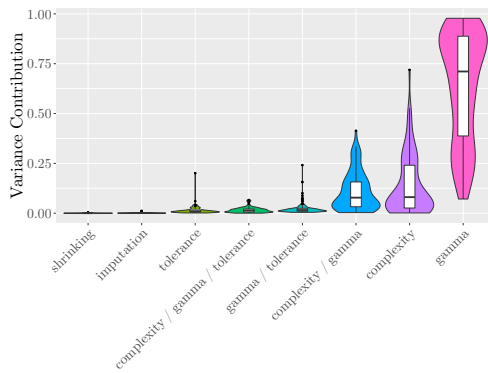
The experiments described in this section increase our understanding of which hyperparameters to focus on in the process of tuning. One important question is how we can use this knowledge to restructure the given configuration space and how it could be incorporated into the new generation of more advanced metalearning/AutoML systems.



(a) Random forest



(b) Adaboost



(c) Support vector machine (RBF kernel)

Fig. 17.11: Variance contribution across datasets. Images taken from van Rijn and Hutter (2018)

## 17.5 Summary

In this chapter, we have presented several experiments that were based upon the results in OpenML. These experiments were organized in blocks of increasing complexity. Section 17.2 describes studies conducted with simple experimental setups. In one, we have examined how the performance of algorithms varies on a single dataset. In another, we have examined the effect of a hyperparameter on a small set of datasets, leading to a better understanding of the algorithms.

As we know, experiments conducted on a single dataset typically do not generalize well, so Section 17.3 extends this by considering various datasets. One study was oriented towards comparisons of performance of different algorithms across different datasets. Another study showed the effect of hyperparameter optimization, allowing the reader to assess its potential effects. The final study was concerned with determining which classifiers make similar predictions and using this information to construct a hierarchical clustering, following Lee and Giraud-Carrier (2011).

The aim in Section 17.4 was to relate the performance to specific data and algorithm characteristics. We have described two studies. In the first one the results showed when it is advantageous to use (or not) feature selection, and similarly, when it is advantageous to use (non-)linear models. The last study aimed to increase our understanding about which hyperparameters are important, which can guide us towards building better configuration spaces and metalearning/AutoML systems.

## References

- Bergstra, J. and Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305.
- Bischi, B., Casalicchio, G., Feurer, M., Gijbbers, P., Hutter, F., Lang, M., Mantovani, R. G., van Rijn, J. N., and Vanschoren, J. (2021). OpenML benchmarking suites. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, NIPS’21.
- Breiman, L. (2001). Random forests. *Machine learning*, 45(1):5–32.
- Domhan, T., Springenberg, J. T., and Hutter, F. (2015). Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*.
- Falkner, S., Klein, A., and Hutter, F. (2018). BOHB: Robust and efficient hyperparameter optimization at scale. In Dy, J. and Krause, A., editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *ICML’18*, pages 1437–1446. JMLR.org.
- Freund, Y. and Schapire, R. (1996). Experiments with a new boosting algorithm. In *Proceedings of the 13th International Conference on Machine Learning*, ICML’96, pages 148–156.
- Frey, P. W. and Slate, D. J. (1991). Letter recognition using Holland-style adaptive classifiers. *Machine Learning*, 6:161–182.
- Friedman, J., Hastie, T., and Tibshirani, R. (1998). Additive logistic regression: a statistical view of boosting. *Annals of Statistics*, 28:2000.
- Hall, M. (1999). *Correlation-based feature selection for machine learning*. PhD thesis, University of Waikato.

- Hutter, F., Hoos, H., and Leyton-Brown, K. (2014). An efficient approach for assessing hyperparameter importance. In *Proceedings of the 31st International Conference on Machine Learning, ICML'14*, pages 754–762.
- Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2011). Sequential model-based optimization for general algorithm configuration. *LION*, 5:507–523.
- John, G. H. and Langley, P. (1995). Estimating continuous distributions in Bayesian classifiers. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, pages 338–345. Morgan Kaufmann.
- Klein, A., Falkner, S., Bartels, S., Hennig, P., and Hutter, F. (2017). Fast Bayesian optimization of machine learning hyperparameters on large datasets. In *Proc. of AISTATS 2017*.
- Landwehr, N., Hall, M., and Frank, E. (2005). Logistic model trees. *Machine Learning*, 59(1-2):161–205.
- Lavesson, N. and Davidsson, P. (2006). Quantifying the impact of learning algorithm parameter tuning. In *AAAI*, volume 6, pages 395–400.
- Lee, J. W. and Giraud-Carrier, C. (2011). A metric for unsupervised metalearning. *Intelligent Data Analysis*, 15(6):827–841.
- Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., and Talwalkar, A. (2017). Hyperband: Bandit-Based Configuration Evaluation for Hyperparameter Optimization. In *Proc. of ICLR 2017*.
- Post, M. J., van der Putten, P., and van Rijn, J. N. (2016). Does feature selection improve classification? a large scale experiment in OpenML. In *Advances in Intelligent Data Analysis XV*, pages 158–170. Springer.
- Probst, P., Boulesteix, A.-L., and Bischl, B. (2019). Tunability: Importance of hyperparameters of machine learning algorithms. *Journal of Machine Learning Research*, 20(53):1–32.
- Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, 1:81–106.
- Radovanović, M., Nanopoulos, A., and Ivanović, M. (2010). Hubs in space: Popular nearest neighbors in high-dimensional data. *JMLR*, 11:2487–2531.
- Rokach, L. and Maimon, O. (2005). Clustering methods. In *Data Mining and Knowledge Discovery Handbook*, pages 321–352. Springer.
- Sharma, A., van Rijn, J. N., Hutter, F., and Müller, A. (2019). Hyperparameter importance for image classification by residual neural networks. In Kralj Novak, P., Šmuc, T., and Džeroski, S., editors, *Discovery Science*, pages 112–126. Springer International Publishing.
- Snoek, J., Larochelle, H., and Adams, R. P. (2012). Practical Bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems 25, NIPS'12*, page 2951–2959.
- Strang, B., van der Putten, P., van Rijn, J. N., and Hutter, F. (2018). Don't rule out simple models prematurely: A large scale benchmark comparing linear and non-linear classifiers in OpenML. In *International Symposium on Intelligent Data Analysis*, pages 303–315. Springer.
- Thomas, J., Coors, S., and Bischl, B. (2018). Automatic gradient boosting. *arXiv preprint arXiv:1807.03873*.
- van Rijn, J. N. (2016). *Massively collaborative machine learning*. PhD thesis, Leiden University.
- van Rijn, J. N. and Hutter, F. (2018). Hyperparameter importance across datasets. In *KDD '18: The 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM.

Vanschoren, J., Blockeel, H., Pfahringer, B., and Holmes, G. (2012). Experiment databases: a new way to share, organize and learn from experiments. *Machine Learning*, 87(2):127–158.





## Concluding Remarks

**Summary.** As metaknowledge has a central role in many approaches discussed in this book, we address the issue of what kind of metaknowledge is used in different metalearning/AutoML tasks, such as algorithm selection, hyperparameter optimization, and workflow generation. We draw attention to the fact that some metaknowledge is acquired (learned) by the systems, while other is given (e.g., different aspects of the given configuration space). This chapter continues by discussing future challenges, such as how to achieve better integration of metalearning and AutoML approaches, and what kind of guidance could be provided by the system when configuring metalearning/AutoML systems to new settings. This task may involve (semi-)automatic reduction of configuration spaces to make the search more effective. The last part of this chapter discusses various challenges encountered when trying to automate different steps of data science.

### 18.1 Introduction

This chapter includes two sections. In the first one, we analyze various forms of metaknowledge used in different approaches discussed in this book. Our aim is to present a unified view on this topic. The second section presents some future challenges in the area of metalearning, with applications to automated machine learning (AutoML). Our aim is to help researchers find interesting and promising lines of work.

### 18.2 Form of Metaknowledge Used in Different Approaches

Different metalearning approaches discussed in this book can be seen as involving two levels, a base-level and a meta-level. The former can be seen as a repository of base-level solutions or partially constructed solutions to address real-world problems, such as diagnosing a patient or predicting the lifetime value of a customer. These would typically include some machine learning algorithm with configured hyperparameters. More complex solutions can be represented as workflows or pipelines of operations and may involve rather complex structures, such as ensembles or deep neural networks.

The meta-level involves different types of information useful in the process of identifying the best possible configured base-level algorithm/workflow/structure for the current task. Various approaches were described in this book regarding how this can be

done. Metaknowledge, i.e., knowledge about the behavior of base-level processes, plays an important role in this process. It is normally derived (or learned) by the metalearning system from the given metadata. Consequently, we will refer in this chapter to this metaknowledge as *learned (or acquired) metaknowledge*.

However, the functioning of metalearning systems is also affected by the application setup, which involves different entities, including:

- A portfolio of base-level algorithms
- Hyperparameters of base-level algorithms and their possible settings
- Ontologies/grammars/operators determining admissible combinations of the above elements
- Application-specific constraints, such as computational resources available, delivery time, and model explainability requirements

The description of these entities is, in this chapter, referred to as *configuration metaknowledge*. These concepts are important, as they determine the space of alternatives that the system can consider in its search for the potentially best one. Normally this metaknowledge is provided by the data scientist. However, as we have shown in Chapter 8, some parts of configuration metaknowledge can be refined by the system.

Our aim here is to revisit some of the approaches discussed in this book, namely:

- Algorithm selection approaches
- Hyperparameter configuration
- Workflow generation
- Knowledge transfer across deep neural networks

and analyze the form of metaknowledge used in each case, with a particular focus on both learned metaknowledge and configuration metaknowledge.

### 18.2.1 Metaknowledge in algorithm selection approaches

In the following subsections, we distinguish between approaches that use either *prior metadata*, i.e., metadata acquired solely on prior tasks, and *dynamic metadata*, both acquired on prior tasks and updated on the current task.

#### Ranking approaches that use prior metadata

Chapter 2 discusses ranking-based approaches to algorithm selection. These approaches use performance metadata obtained on different prior tasks/ datasets to construct an *average ranking*. This ranking is followed on the target dataset until a given time budget has been exhausted. Dataset characteristics can be used to preselect a subset of datasets most similar to the target dataset and, thus, guide the search using the relevant metadata. The best algorithm identified is used to obtain the predictions on the target dataset. So in this approach, the average ranking represents the learned metaknowledge, ready to be applied to the new task. This approach requires that appropriate configuration metaknowledge is given.

Despite its simplicity, this approach can suggest quite good algorithms or workflows for the target dataset and achieve good performance. This is due to the fact that the given configuration metaknowledge can include rather complex structures, such as ensembles or deep neural networks and various variants of these tuned on specific tasks. So if the new task is similar to one of the past tasks, a ready-made solution is available. One disadvantage of this method is that the configuration metaknowledge is in an extensional form (e.g., a ranking of workflows (pipelines)), and also, that the ranking is static.

## Approaches that exploit dynamic metadata

Some approaches discussed in Chapter 5 were conceived to overcome one of the shortcomings associated with the previous approach. They exploit pairwise tests and performance-based dataset characteristics (metafeatures). So the performance metadata is updated as the tests proceed on the current dataset. In other words, the metadata is changing dynamically.

The approach of *active testing* uses *estimates of performance gain* to characterize pairs of models (trained algorithms). These estimates represent the expected difference between the performance of the two models. So the metaknowledge is in the form of such estimates. This information is used by the system to suggest algorithms or workflows that may obtain higher performance than the incumbent (current best candidate).

A combination of prior and dynamic metadata is also often used in the application of metalearning for ensemble methods, which is discussed in Chapter 10.

### 18.2.2 Metaknowledge in approaches for hyperparameter optimization

Chapter 6 discusses how metalearning can be used for hyperparameter optimization. Various approaches discussed there explore the fact that the “performance surface” of different algorithm configurations (configurations with different hyperparameter settings) is rather “smooth” for many algorithms.<sup>1</sup> Consequently, it is possible to construct a meta-level model (often referred to as a *surrogate model*) that can be used in the search for configurations which are likely to lead to higher performance. The use of surrogate models has the advantage that it enables to identify the promising new configurations fast.

The meta-level model could be seen as a part of the learned metaknowledge. This approach is dynamic, because as more configurations have been examined, the meta-level model becomes more refined. Thus the refined model is, in turn, capable of providing more reliable suggestions regarding which hyperparameter settings could lead to better performance. The final model identified represents also learned metaknowledge, albeit generated on a specific task.

Some approaches use metaknowledge learned both on prior datasets and on the current dataset in this process, as was shown in Chapter 6. Some surrogate models can be employed across tasks. They learn a similarity function over tasks and can therefore transfer knowledge about parts of the configuration that work well to similar tasks.

### 18.2.3 Metaknowledge in workflow design

As the space of alternative workflows can be potentially very large, it is difficult to rely on extensional approaches consisting of the enumeration of alternatives. Various means have been designed to overcome this problem. The representation adopted is *intensional*, permitting to generate various possible workflows or their configurations. Chapter 7 discusses some representations commonly used in this area. These include:

- Ontologies (see Section 7.2)
- Context free grammars (CFGs) (see Section 7.2)
- Abstract/concrete operators of a planning system (see Section 7.3)

---

<sup>1</sup>Some algorithms (e.g., SVM with a specific kernel) do not really have a smooth performance surface.

As shown in Chapter 7, each form has certain advantages and disadvantages. These representations normally embody the knowledge of data scientists, hence are a part of configuration metaknowledge.

As we have pointed out in Chapter 7, the process of searching for the potentially best workflow can be seen as a process of gradually refining the metaknowledge acquired on the current task (learned metaknowledge).

#### **18.2.4 Metaknowledge in transfer learning and in deep neural networks**

Transfer learning, discussed in Chapter 12, is concerned with transferring (parts of) models, such as, parameters (i.e., weights), from one task to another.

Both the area of transfer learning and deep neural networks discuss the use of pre-trained models on a set of relatively homogeneous datasets (meta-examples). This model can then be used (more precisely, its parameters) as the initial model on the target dataset (meta-example) (see, e.g., MAML discussed in Chapter 13). This model is then fine-tuned with new instances from the new task. This leads to satisfactory performance, even in problems where the number of available (labeled) instances is small. This scenario is often called *few-shot learning*.

So, if we use the terms used in the chapter, we note that the initial model (or the parameter weights) represent learned metaknowledge that is transferred to the target task. Although this still requires some training procedure on data from the target task, the required amount of data and time are often eminently smaller than when one would start without metaknowledge.

### **18.3 Future Challenges**

Recent years have witnessed a large growth in research in the areas of metalearning and AutoML as well as their integration into data science tools. However, many interesting and relevant problems remain unsolved, or better solutions could be developed. In the following subsections we mention some of them.

#### **18.3.1 Design of metafeatures relating dataset characteristics and performance**

Preselecting datasets according to their similarity to the target dataset involves one of the most complex challenges in the development of metalearning systems: designing metafeatures that represent characteristics of the datasets that affect the (relative) performance of algorithms, as discussed in Chapter 4. Despite the different types of metafeatures that have been proposed and some work on the systematization of their design and usage, new useful contributions could still be made.

#### **18.3.2 Further integration of metalearning and AutoML approaches**

In Section 6 we have discussed some approaches whose aim was to explore both metaknowledge acquired on past problems and metaknowledge acquired on the current problem. The former capture general information about the behavior of learning processes,

while the latter enables an adjustment of the solution to the current task. It is foreseeable that these approaches could be further improved. So the question is: what is the best method of integrating the two types of metaknowledge? This question can be addressed along different lines. Some are discussed in the following sections.

### 18.3.3 Automating the adaptation to the current task

It seems desirable that metalearning/AutoML systems should have the ability to carry out automatic adaptation (self-configuration) to the current task. This could involve, for instance, identifying the appropriate domain-specific knowledge, or learned metaknowledge that was acquired on similar tasks. Additionally, metaknowledge from the current task could enrich the metaknowledge acquired earlier.

#### Automating metadata acquisition

In Chapter 8 we have discussed some conditions that need to be satisfied so that we would have a competent AutoML/metalearning system. One of those conditions involves the size of metadata, which determines the quality of acquired metaknowledge. So the question is whether one should keep exploring the domain by conducting new tests or just focus on exploitation of the existing metaknowledge. Chapter 8 (Section 8.9), which discusses some strategies used in the area of multi-armed bandits, offers some suggestions regarding this problem. Still, more work is needed to provide the best possible solution to the issue.

### 18.3.4 Automating the reduction of configuration spaces

Many systems may adopt a strategy of defining a rather large configuration space in the search for a solution. This is motivated by the fact that, if the space was too small, the system would not be able to find a satisfactory model. However, configuration spaces that are too large also have disadvantages. As there are too many options to explore, the system may require a very long time to find the potentially best solution. Additionally, many alternatives may be redundant (i.e., lead to the same or a very similar model), which do not add value and waste resources. Chapter 8 discusses some strategies that can be used to reduce these spaces. The configuration spaces are affected by:

- The contents of a given portfolio with base-level algorithms
- The hyperparameters of algorithms and their possible values
- Ontologies/grammars/operators determining the space of admissible workflows

Each of these cases is analyzed next in a separate subsection.

#### Automating the reduction of base-level algorithms

One solution to the problem of reduction of base-level algorithms, based on metaknowledge acquired on prior tasks, was described in Chapter 8 (Section 8.5). The method included two basic steps: identify competent algorithms and eliminate redundant algorithms. It is conceivable that the proposed method can be further improved.

We note that even configuration metaknowledge can also be revised in an automatic way. This process can be seen as *meta-metalearning*, as its aim is to revise the existing metaknowledge.

## Automating the reduction of hyperparameter spaces

Chapter 8 (Section 8.4) discusses methods that enable to identify the most important hyperparameters of a given algorithm. This is useful, as it enables the human designer to redefine the configuration space accordingly. However, the challenge is to design a system that would do this in an automatic way. One step in this direction are surrogate models that transfer knowledge about the behavior of parts of the space of hyperparameters across models (see Chapter 6).

## Automating the reduction of workflow (pipeline) spaces

The given ontologies/grammars/operators can be regarded as another form of configuration metaknowledge, determining which workflows (pipelines) are admissible in the configuration space. The challenge is how to design a system that would be capable of learning/updating this type of metaknowledge automatically, for instance, from given examples of admissible/inadmissible workflows.

### 18.3.5 Automating data stream mining

Many real-world datasets are in fact continuous streams of data. Various approaches that employ metalearning were discussed in Chapter 11, which covered several approaches:

Several methods split the stream into various intervals of equal size and extract metafeatures per interval. It is possible to build a meta-model based on such features and predict what would be the potentially best algorithm (e.g., classifier) for the next interval.

Alternatively, BLAST trains multiple classifiers over the course of the data stream, and selects an appropriate classifier for a next set of observations, based on the performance on previous observations.

Finally, streams can contain seasonality and recurring concepts. By storing previous models, ensemble and/or metalearning approaches can be used to select an appropriate method for a next part of the stream.

Despite the variety of approaches, various challenges remain. One of the key aspects of the data stream setting is the occurrence of concept drift. At any moment in time, a gradual or abrupt change in the data can degrade the performance of earlier trained models. While drift detectors have served to overcome this difficulty, new work on self-assessing machine learning tools can play an important role in improving this aspect (König et al., 2020).

### 18.3.6 Automating neural network parameter configuration

Training deep neural networks is known to be both a data-intensive and time-consuming process. Modern metalearning approaches can help to solve these issues by transferring knowledge from related tasks. For many approaches, this knowledge transfer is on the level of parameters, rather than hyperparameters. *Few-shot learning* aims to apply these techniques to scenarios where only very few data items are available.

As these techniques are applicable to relatively homogeneous data sources, so a question arises regarding how to estimate the “degree of homogeneity”, as this could determine how much training is needed beyond few examples that would normally be used. As such, an important challenge is determining which datasets can be considered as “similar”, and developing techniques that automatically detect from which datasets knowledge can be transferred.

### 18.3.7 Automating of data science

The area of data science has attracted a lot of attention recently, as it represents a more general framework than, for instance, data mining. Consequently, the question of whether it is possible to use automated methods arose and was addressed by some researchers. In Chapter 14 we have presented our perspective of what data science should involve. In this chapter we have considered the following stages:

1. Defining the current problem/task
2. Identifying the appropriate domain-specific knowledge
3. Obtaining the data
4. Automating data preprocessing and transformation
5. Changing the granularity of representation
6. Searching for the best workflow
7. Automating report generation

Although every stage has its challenges, the work on some stages has advanced more than on others. For instance, various papers discuss methods appropriate for stage 4, involving automation of data preprocessing and transformation. The reader can consult Chapter 14 (Section 14.3) to obtain more details on this issue.

Stage 6, which is concerned with the search for the potentially best algorithm/configuration/workflow, was discussed in various chapters of this book. Stage 7 was briefly discussed in Chapter 14. It can be expected that the existing work provides a good basis for further progress.

In our view, stages 1, 2, 3, and 5 represent a more serious challenge, as not much work has been done that could aid automation. In the following subsections these stages are addressed in more detail.

#### Definition of the current problem/task

In Chapter 14 we argued that, although the initial description needs to be provided by the domain expert (e.g., in natural language), further processing can be done with the help of (semi-)automatic methods. We have presented some initial ideas on this matter, which involve the usage of “task descriptors (keywords)”, that can be used in further processing, such as the one discussed in the next subsection.

#### Identifying the appropriate domain specific metaknowledge

This issue is relevant for any system that aspires to resolve rather diverse problems. Here, the domain-specific knowledge involves metadata, metaknowledge acquired on prior tasks, and definition of appropriate configuration spaces (configuration meta-knowledge). This has the advantage that it enables to focus on a smaller configuration space, and consequently the search for the potentially best solution is easier.

#### Obtaining the data

This step may be easy to carry out, provided someone has already told the system where the data is. Typically the data would be stored and retrieved from some kind of database or OLAP. This stage may, however, present challenges if we want the system to be more independent and not always rely on human assistance. As pointed out in Section 14.2,

we may encounter a problem for which hardly any data is available. A great deal of human scientific work is of this kind. The scientist has to elaborate a plan, which normally involves some form of interaction with the outside world, or appropriate sites on the internet, to obtain the required data. In Section 14.2, we mention, for instance, the system *Robot Scientist* that does just this. Some data science systems of the next generation may want to include this facet in their design.

## Changing the granularity of representation

This area is also not without challenges. As was pointed out in Chapter 14, many practical applications exploit the information in databases or in an OLAP cube to generate appropriate *aggregate data* so as to be able to advance with data mining. Although some papers addressed this issue in the past and the topic is of utmost importance in many practical applications, this did not seem to have been reflected much at some recent data science workshops (e.g., ADS 2019 mentioned before).

However, as pointed out in Chapter 15, other changes of granularity can be considered, apart from the aggregation mentioned above. So the challenge is whether these processes could be incorporated into a more advanced AutoDS system.

### 18.3.8 Automating the design of solutions with more complex structures

In Chapter 15 we discuss various future lines of research. It is pointed out that not all solutions can be represented in the form of workflows, as more complex structures may be required (e.g., conditional operators, iteration, etc.). So the challenge is how to extend the existing metalearning/AutoML approaches to enable solutions with more complex structures.

### 18.3.9 Designing metalearning/AutoML platforms

Various chapters of this book not only discussed different approaches but also provided details about the corresponding implemented systems. Particular attention was given to existing metalearning/AutoML platforms, as these can typically be applied to many diverse problems. So the platforms are very important not only for future research, but also for real-world deployment.

However, the platforms do not always include the latest advances in the area. So the challenge is to extend the existing platforms, design new ones, and conduct comparative studies that enable to identify the more competitive variants for further work.

## Final Challenge to the Reader

In the last couple of years, we have seen many new solutions to both old and new problems, and many new challenges arose in this context. With the rise of AutoML and metalearning for deep neural networks, new research communities have emerged. We hope that we have gathered some interest in the reader that would inspire him/her to participate in the development of respective solutions. We plan to review these challenges and their solutions in the next edition of the book.



## References

- König, M., Hoos, H. H., and van Rijn, J. N. (2020). Towards algorithm-agnostic uncertainty estimation: Predicting classification error in an automated machine learning setting. In *7th ICML Workshop on Automated Machine Learning (AutoML)*.



---

# Index

- $\epsilon$ -decreasing strategy, 163
- $\epsilon$ -first strategy, 163
- $\epsilon$ -greedy strategy, 163
- $k$ -shot learning, 225
- $n$ -way  $k$ -shot learning, 225, 239
  
- A3R, 31
- absorption, 287
- abstract operators, 132
- acquisition function, 110
- active testing (AT), 93
- adaptive batch classifiers, 203
- adaptive capping, 108
- adaptive dataset similarity, 117
- adaptive ensembles, 203
- adequacy of configuration spaces, 146
- aggregate functions, 278
- aggregate operator
  - in Rapid Miner, 279
- aggregating data, 278
- AI planning, 131
- algorithm
  - configuration (AC), 104
  - recommendation, 19
  - selection, 19
- algorithm recommendation
  - for data streams, 201
- algorithm selection, 77
- ALMA, 196
- AlphaD3M system, 136
- approximate ranking trees (ARTs), 93
- AR\*, 32
- arbitrating, 182
- ART forests, 93
  
- ART trees, 93
- attentive recurrent comparators, 241, 247
- Auto-WEKA
  - parameter space, 127
- Automated Statistician, 279
- automatic feature construction
  - in DNNs, 288
- automatic machine learning (AutoML), 104
- automating
  - data preprocessing, 274
  - data science, 269
  - report generation, 279
  - the design of complex systems, 283
  - workflow design, 123
- AutoML, 104
- average ranking, 27
  
- bagging, 170
- base-level operators, 132
- batch-incremental classifiers, 203
- best classifier on last interval, 209, 210
- best-first search, 107
- bias
  - control, 155
  - domain-specific, 155
  - weak/strong, 155
- bias vs. variance in metalearning, 231
- Blast, 209, 210
- Boltzmann exploration, 163
- boosting, 172
- bounds
  - for domain adaptation, 231
  - on generalization error, 229

- using algorithmic stability, 230
- budget, 24
- business problem, 270
- cascade generalization, 175
- cascading, 177
- CASH, 104
- catastrophic forgetting, 226
- CHADE, 196
- changing the granularity of representation, 278
- characterizing diversity, 159
  - by correlation of rankings, 159
- choice of base-classifiers, 213
- CITRUS, 136
- class overlap, 59
- class separability, 59
- classification models at meta-level, 83
- classifier output difference (COD), 153, 193, 213
- classifiers
  - adaptive, 203
  - batch-incremental, 203
- clause structure grammar, 290
- cleaning, 277
- clustering trees, 81
- CMA-ES, 107
- COD, 153, 213
- cold-start problem, 113
- combined algorithm selection and hyperparameter optimization (CASH), 104
- combined measure, 31
- combining
  - accuracy and time, 31
- combining base-learners, 169
- competence region, 195
- complete metadata, 159
- concept drift, 202
  - detector, 215
- conditional hyperparameter, 103, 145
- conditional neural processes, 241, 255
- conditional operators, 285
- configuration space, 144
  - continuous, 145
  - discrete, 145
- configuration spaces, 98, 143
  - adequacy, 146
  - principles of constitution, 147
  - reduction, 151
- constructive induction, 286
- context-free grammars (CFGs), 128
  - induction, 130
  - limitations, 130
- contextual-bandit problem, 164
- continuous configuration space, 145
- control layer
  - in two-layered architecture, 215
- controlling
  - bias, 155
- cuboids, 279
- data
  - characteristics, 53
  - preprocessing, 274
  - science, 269
  - transformation, 275
- data characterization
  - algorithm-specific, 68
  - dynamic, 68
  - in clustering, 63
  - in regression tasks, 60
  - in time series prediction, 62
  - in unsupervised learning, 63
  - iterative, 68
  - task-dependent, 54
  - useful for ranking pairs of algorithms, 69
- data envelopment analysis, 33
- data preprocessing
  - cleaning, 277
  - instance selection, 277
  - outlier elimination, 277
- data stream, 201
  - ensembles, 210
- data wrangling, 275
- dataset, 25
  - repositories, 157, 298
  - segmentation, 158
  - similarity, 25, 34
  - variants, 157
- datasets
  - 2D footprints, 158
  - synthetic, 157
  - which are needed?, 156
  - with discriminatory power, 158
- DEA, 33
- declarative bias, 125
- default settings, 104
- defining

- abstract operators, 289
  - configuration space, 125
- delegating, 180
- descriptors of learning goals, 273
- discounted cumulative gain (DCG), 51
- discrete configuration space, 145
- DM processes, 123
- domain-specific
  - metafeatures, 271
- double-loop architecture, 225
- drift detection, 207
  - metafeatures, 207
- drift detector
  - ADWIN, 207
  - DDM, 207
- drill down/up, 278
- dynamic
  - algorithm selection, 205
  - classifier selection, 190, 195
  - data characterization, 68
  - selection of models, 195
- empirical risk, 228
- ensemble learning, 189
  - generation, 192
  - integration, 195
  - iterative process, 193
  - phases, 192
  - pruning, 192
- ensemble of surrogate models (SGPT), 116
- ensemble recommendation
  - by ensemble learning, 192
  - by selection, 191
- ensembles
  - accuracy-weighted, 214
  - adaptive, 203
  - for data streams, 210
  - heterogeneous, 191
  - hierarchical, 197
  - homogeneous, 191
  - in Auto-sklearn, 195
- ensembles of algorithms, 169
- entropy search, 111
- error correlation, 153
- establishing hyperparameter importance, 149
- evaluation, 30
  - by correlation, 45
  - of rankings, 30, 39
  - of recommendations, 39
- evolutionary computation (EC), 197
- evolutionary search, 107
- expected improvement (EI), 111
- expected loss, 228
- experiment design, 118
- explainability, 284
- fading factors, 211
- faithful symbolic model, 284
- feature drift, 213
- feature-based transfer learning, 222
- few-shot learning, 240
- Flashfill wrangler, 277
- folding and unfolding, 287
- FOOFAH wrangler, 276
- Friedman's test, 50
- functional transfer, 221
  - in neural networks, 223
- Gaussian mixture model (GMM), 159
- Gaussian processes (GPs), 80, 111
  - with multi-kernel learning (MKL-GP), 115
- goal dependency networks (GDN), 273
- golden standard, 46
- gradient descent, 108
- granularity of representation, 285
- graph neural networks, 241, 246
- grid search, 105
- Hasse diagrams, 20
- heterogeneous ensembles, 191
- heuristic ranker, 128
- heuristic search methods, 107
- hierarchical ensembles, 197
  - generation, 197
- hierarchical planning, 132
  - abstract operators, 132
  - base-level operators, 132
  - HTN planning, 133
  - ML-Plan, 133
- hill climbing, 107
- Hoeffding bound, 203
- Hoeffding tree, 203
- homogeneous ensembles, 191
- homogeneous transfer, 221
- HTN planning, 133, 289
- Hyperband, 109
- hypergradients, 108

- hyperparameter
  - categorical, 103
  - conditional, 103
  - numeric, 103
  - optimization (HPO), 104
- hyperparameter importance, 148
  - ablation analysis, 148
  - forward selection, 148
  - functional ANOVA, 148
- hyperparameter log-scale, 145
- hyperparameters
  - establishing importance, 149
- identifying the domain-specific
  - background knowledge, 273
  - data, 273
- identifying the task domain, 272
  - by classification, 273
  - by matching descriptors, 272
- incomplete metadata, 159
  - effects, 162
  - using estimates, 161
- inductive logic programming (ILP), 274, 288
- initialization
  - of configuration, 113
- inner loop, 225
- input warping, 145
- instance selection, 277
- instance-based transfer learning, 222
- Intelligent Discovery Assistant (IDA), 126
- inter-construction, 288
- interplay between subsymbolic and symbolic models, 284
- interval estimation strategy, 164
- interval of confidence, 22
- introducing new concepts, 285
  - autonomously, 286
  - by clustering, 286
  - from external sources, 285
- introduction of new concepts
  - absorption, 287
  - folding and unfolding, 287
  - inter-construction, 288
  - propositionalization, 288
- invariant transformations, 224
- iterated local search, 108
- iterative data characterization, 68
- iterative learning, 290
- iterative pairwise tests, 90
- KDD process, 124, 270
- Kendall's Tau correlation, 46
- kernel density estimation, 112
- landmarkers, 84
  - in ensemble learning, 197
- latent embedding optimization, 261
- layered learning, 289
- learning
  - k*-shot, 225
  - inter-dependent tasks, 292
  - multi-task, 223
  - nested, 225
  - rankings, 115
  - similarity, 225
  - with recurrent neural networks, 226
- learning goals, 272
  - schedule, 272
- learning similarity, 225
- learning-to-learn, 219, 229
- loose accuracy, 50
- loss curve, 31, 47
- loss-time curve, 49
- LSTM optimizer, 241, 257
- macro-operators, 136
- marginal, 149
- marginal contribution, 195
  - of algorithms, 148
  - Shapley value, 148
  - to performance, 148
- matching descriptors (metafeatures), 272
- matching networks, 241, 245
- median rank, 26
- memorization problem, 263
- memory-augmented neural networks, 226, 241, 249
- meta networks, 241, 251
- meta-decision trees, 183
- meta-level performance, 208
- meta-model, 24
  - instance-based (IBL), 80
  - regression-based, 78
- Meta-SGD, 261
- MetaBags, 196
- metadata, 24
  - complete, 159
  - incomplete, 159
  - repositories, 297
- metadistribution, 229

- metafeature-based approaches, 205
- metafeatures, 34, 53
  - aggregated, 65
  - complexity-based, 58
  - concept variation, 58
  - domain-specific, 271
  - in ensemble learning, 194
  - information-theoretic, 56
  - landmarker sets, 58
  - landmarkers, 57
  - model-based, 56
  - relational representation, 69
  - relative landmarks, 57
  - selection, 67
  - simple, 55
  - statistical, 55
  - subsampling landmarks, 58
- metalearning, 219, 225
  - approaches, 19
  - in ensemble construction, 189
  - in ensemble learning, 189, 195
  - metric-based, 237, 240, 241, 248, 263
  - model-based, 237, 241, 248, 255, 263
  - optimization-based, 237, 241, 256, 263
- metalearning approaches, 77
- metaparameter, 225
- MiningMart, 136
- ML-Plan, 133
- MOA framework, 213
- model, 24
  - meta-level, 24
- Model inference system (MIS), 290
- model-agnostic meta-learning, 241, 259
- model-based
  - metafeatures, 56
- Monte Carlo tree search (MCTS), 136
- multi-armed bandits (MAB), 162
- multi-criteria analysis, 33
- multi-fidelity techniques, 108
- multi-task learning, 223, 227
- multitarget prediction, 81
  
- Naive AutoML, 136
- negative transfer, 220
- Nemenyi test, 50
- nested learning, 225
- non-homogeneous transfer, 221
- non-propositional approaches, 98
- non-stationarity, 202
- normalizing performance, 42, 80
  
- obtaining data, 273
  - by planning, 273
  - from different sources, 274
  - from OLAP data cube, 274
- OLAP data cube, 274, 278
- online
  - bagging, 204
  - boosting, 204
  - learning, 202
  - MAML, 203
- ontologies, 126, 289
  - DM ontology of Intelligent Discovery Assistant (IDA), 126
  - shortcomings, 127
- OpenAI Gym, 299
- OpenML, 297, 299
  - dataset characteristics, 299
  - datasets, 299
  - flows, 301
  - integration in Python, 305
  - integration in R, 303
  - integration in Weka, 303
  - runs, 302
  - setups, 302
  - tasks, 300
- operators, 131
  - abstract, 132
  - base-level, 132
  - conditional, 285
  - manual selection, 131
  - post-conditions/effects, 132
  - preconditions, 132
  - repeat, 285
- optimistic reward estimate, 164
- order of experiments
  - does it matter?, 162
  - multi-armed bandits, 162
  - sequential model-based optimization, 162
- outer loop, 225
- outlier elimination, 277
  
- pairwise comparisons of algorithms, 83
- parameter-based transfer learning, 222
- partial
  - learning curves, 85
  - ranking, 90
- PCC-DES, 196
- performance, 22
  - aggregation, 26

- comparable, 22
- high-performance regions, 117
- loss, 47
- metadata, 24
- models, 80
- normalization, 80
- planning
  - exploiting metaknowledge, 135
- plans, 284
- portfolio, 144
  - characterizing diversity, 159
  - of algorithms, 191
  - of ensemble elements, 191
  - of successful workflows, 137
  - reduction, 151
- pricing strategies (POKER), 164
- probability matching strategy, 163
- probability of improvement (PI), 111
- problem
  - description, 270
  - understanding, 270
- procedural bias, 128
- programmatically access to datasets, 298
- propositionalization, 288
- prototypical networks, 246
- Q-statistic, 193
- quantile regression, 117
- quantile regression forest (QRF), 117
- query set, 240
- random forests (RFs), 80
  - as surrogate models, 112
- random search, 106
- rank correlation
  - Spearman's, 46
  - weighted, 47
- ranking, 20
  - accuracy, 46
  - average, 27
  - complete, 20
  - evaluation, 30
  - incomplete, 20, 34, 35
  - linear, 23
  - meta-model, 28
  - of successful plans (workflows), 136
  - partial, 20
  - quasi-linear (weak), 23
  - total, 20
- Rapid Miner
  - aggregate operator, 279
  - recommendation, 20
    - forms, 20
    - of algorithm, 20
    - of workflows (pipelines), 32
  - recurring meta-level models, 213
  - reducing
    - configuration spaces, 151
    - portfolios, 151
  - reduction
    - by envelopment curve, 154
  - reformulating
    - ontologies, 289
    - recursive definitions, 290
    - theories, 286
  - reformulation of theories
    - by specialization, 286
  - reinforcement learning optimizer, 241, 258
  - relation networks, 246
  - repeat operators, 285
  - report generation, 279
  - representation of goals, 273
  - representational transfer, 221
  - reproducibility, 298
  - reptile, 241, 261
  - rescaling, 42
  - reusing new concepts
    - in further learning, 289
    - to redefine ontologies, 289
  - reusing prior experiments, 298
  - reusing solutions, 289
  - Robot Scientist, 273
  - roll-up, 278
  - schedule for learning goals, 272
  - scheduling experiments, 162
    - $\epsilon$ -decreasing, 163
    - $\epsilon$ -first, 163
    - $\epsilon$ -greedy, 163
    - interval estimation strategy, 164
    - multi-armed bandits, 162
    - optimistic reward estimate, 164
    - pricing strategies (POKER), 164
    - probability matching, 163
    - SoftMax, 163
    - upper confidence bound (UCB), 164
  - search
    - for the best workflow, 279
  - search methods



- that exploit metaknowledge, 114
- selection
  - of preprocessing method, 278
  - of workflows, 278
- sequential model-based
  - optimization (SMBO), 110
  - search, 110
- SGPT, 116
- Shapley value, 148
- shifting window, 206
- Siamese neural networks, 241
- siamese neural networks, 244
- significant wins, 27
- similar learning curve, 86
- similarity
  - correlation-based, 70
  - cosine-based, 70
  - performance-based, 97
- similarity measures, 69
- simple neural attentive meta-learner, 241, 253
- SKILit, 290
- sliding window, 206
- SoftMax strategy, 163
- source domain, 220
- Spearman's rank correlation, 46
- SPegasos, 203
- stacking, 173
- statistical test, 22
- stochastic gradient descent (SGD), 108, 203, 226, 262
- stream classifiers, 213
- strong sufficiency, 274
- subsampling landmarks, 84
- successive halving, 108
- sum of wins and losses, 88
- supervised
  - learning, 238
  - metalearning, 238
- supervised transfer, 221
- support set, 240
- surrogate
  - collaborative tuning (SCoT), 115
  - model, 110
- SVM
  - effects of hyperparameters settings, 306
- SVMRank, 115
- SYNTH, 277
- Synth-a-Sizer wrangler, 277
- synthetic datasets, 157
- tabling, 136
- target domain, 220
- task
  - descriptor generation, 271
  - domain, 272
  - goals, 271
  - type, 271
- task embedding, 263
- task-dependent characterization, 54
- templates, 198
- top- $n$  strategy, 28
- TPOT, 133
- transfer
  - across tasks, 219
  - by clustering, 227
  - functional, 221
  - homogeneous, 221
  - in kernel methods, 227
  - in neural networks, 223
  - in parametric Bayesian models, 227
  - learning, 219, 220
  - negative, 220
  - non-homogeneous, 221
  - of knowledge, 219
  - representational, 221
  - supervised, 221
  - unsupervised, 221
- transfer acquisition function (TAF), 116
- transfer learning
  - feature-based, 222
  - instance-based, 222
  - parameter-based, 222
- transformation
  - of data, 274
- tree-structured Parzen estimator (TPE), 112
- Trifacta wrangler, 276
- true ranking, 46
- true risk, 228
- trusted AI, 284
- two-layered architecture
  - control layer, 215
- unsupervised transfer, 221
- upper confidence bound (UCB), 164
- upper confidence bounds (UCB), 111
- Vapnik–Chervonenkis (VC) dimension, 228
- version spaces, 154

warm-start strategy, 113  
window size, 208  
workflow design  
  manual, 131  
  using planning, 132  
workflow/pipeline design, 123

workflows, 284  
wrangling, 275  
  Flashfill, 277  
  FOOFAH, 276  
  Synth-a-Sizer, 277  
  Trifacta, 276

