TECHNISCHE
UNIVERSITÄT
WIEN

VIENNA
UNIVERSITY OF
TECHNOLOGY

MAGISTERARBEIT

Meta-Learning: Machine Learning with No Idea?

Ausgeführt am Insitut für

Statistik und Wahrscheinlichkeitstheorie
der Technischen Universität Wien

unter der Anleitung von

Univ.-Prof. Dipl.-Ing. Dr.techn. Friedrich Leisch

durch

Manuel J. A. Eugster, Bakk.techn.
Columbusgasse 41/9
1100 Wien

Wien, Jänner 2007

TECHNISCHE
UNIVERSITÄT
WIEN

VIENNA
UNIVERSITY OF
TECHNOLOGY

MASTER THESIS

Meta-Learning:
Machine Learning with No Idea?

Performed at the Departement of

Statistics and Probability Theory
Vienna University of Technology

advised by

Univ.-Prof. Dipl.-Ing. Dr.techn. Friedrich Leisch

by

Manuel J. A. Eugster, Bakk.techn.
Columbusgasse 41/9
1100 Vienna

Vienna, January 2007

Acknowledgement

Now as almost everything is in place and the thesis is on the home stretch, I will take the time to thank the people that helped me in one or the other way.

First of all I would like to thank Professor Friedrich Leisch for accepting me as graduand and just let me freely think about my chosen thesis topic. Then I thank Dr. David Meyer to give me his results about his benchmark studies to shorten my computation time. And I thank Dr. Torsten Hothorn to explain me, in spite of the useR! 2006 conference stress, the theory of benchmark studies and the therefor used statistical tests in detail.

A big thank you to Theresia Ledinegg for her english proofreading and thanks to Stefan Schnabel and Jürgen Schatzmann for reading through the thesis and giving general hints.

A really really big thank you to my girlfriend Sarah, my parents Hartwig and Maria and my sister Carola. Thanks for supporting me in all my ingenious and less ingenious ideas, this is for you!

Abstract

The goal of this thesis is to provide an answer to a question, which came to my mind in different lectures about machine learning, data mining and neural networks:

> "And how do I now choose the right method for a new learning problem?"

Meta-Learning provides a possible answer. Thereby the idea is, to use the quality information of machine learning methods on already treated learning problems and generate a suggestion for the new learning problem.

The considered learning problems are restricted to classification problems. The quality and the consequent ranking of different classification methods on a learning problem are determined with benchmark experiments. Concrete quality measurements are the misclassification rate and the time of a benchmark experiment. To relate different problems, a characterisation with statistical and information-theoretic measures is defined, and a similarity measures with attention to the specialties of the characterisation is introduced.

On the basis of these data a new problem is defined, which answers the initially placed question. There exist three different methods in the literature. The first two methods define the problem as classification respectively as regression problem with most different possibilities for the response variable. The third method uses similarities between problems and a schema to cause the ranking. All three approaches are introduced, a concrete formulation is done for a regression problem with the Nadaraya-Watson estimator, and for the third method with the average ranks, success rate ratios and adjusted ratio of ratios schemata.

The practical implementation of the theoretical elucidations takes place with the `R` system for statistical computing and a case study with 21 learning problems and 6 methods shows the concrete usage.

Zusammenfassung

Das Ziel dieser Masterarbeit ist es, eine Antwort auf die Frage zu finden, die sich mir in verschiedenen Vorlesungen über Machine Learning, Data Mining und Neuronale Netzwerke gestellt hat:

> “Und wie entscheide ich mich jetzt für die richtige Methode bei einem neuen Lernproblem?”

Eine mögliche Antwort liefert meta-learning. Dabei wird das Wissen bezüglich der Qualtität verschiedener Machine Learning-Methoden aus schon behandelten Lernproblemen für einen Vorschlag für das neue Lernproblem verwendet.

Die betrachteten Probleme beschränken sich auf Klassifikationsprobleme. Die Qualität und die daraus resultierende Reihenfolge verschiedener Klassifikationsmethoden für ein Lernproblem wird mittels Benchmark-Experimenten bestimmt. Maße für die Qualität sind die Missklassifikationsrate und die Zeit welche ein Benchmark-Experiment benötigt. Um die verschiedenen Probleme in Relation bringen zu können, wird eine Charakterisierung mittels statistischen und informationstheoretischen Maßen definiert. Ebenso wird ein Ähnlichkeitsmaß mit Behandlung der speziellen Eigenschaften der Charakterisierung eingeführt.

Auf Basis dieser Daten wird ein neues Problem definiert, welches eine Antwort auf die eingangs gestellte Frage liefert. In der bestehenden Literatur existieren dazu drei verschiedene Methoden. Die ersten beiden Methoden formulieren das Problem als Klassifikations- bzw. Regressionsproblem mit unterschiedlichsten Möglichkeiten für die abhängige Variable. Die dritte Methode verwendet die Ähnlichkeit von Problemen und ein Schema für die Reihenfolgebestimmung. Alle drei Ansätze werden einführend erklärt, eine konkrete Formulierung wird für ein Regressionsproblem mittels Nadaraya-Watson-Schätzer und der dritte Methode mit den average ranks, success rate ratios und adjusted ratio of ratios Schemata angegeben.

Die praktische Umsetzung der theoretischen Erläuterungen erfolgt mit dem R-System für statistische Berechnungen und ein Fallbeispiel mit 21 Lernproblemen und 6 Methoden zeigt eine konkrete Anwendung.

Contents

Chapter 1

Introduction

The goal of this chapter is to point out the idea of my master thesis. Therefor I give a rough overview about machine learning and meta-learning with its various techniques. Then I point out my framework and explain the case study, which I will use through the whole thesis.

1.1 Machine Learning

The aim of machine learning is to construct algorithms which are able to learn to solve a problem given some prior knowledge about it. Mitchell (1997) defines a well-posed learning problem as follows:

> “A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E”.

A first rough division of the tasks T is based on their goals. In supervised learning , the goal is to predict the value of an outcome measure given a number of input measures. Unsupervised learning is the task to describe associations and patterns among a set of input measures. Furthermore, supervised learning tasks are classified based on the type of their outputs: regression when it is a quantitative and classification when it is a qualitative ouput (Hastie et al., 2001). One typical unsupervised learning task is cluster analysis.

The experience E is given in most cases with a set of samples of the concrete problem. Figure 1.1 shows such a set of samples for simple example problems of all three named tasks.

The performance measure P expresses the “how good has a computer program learned the problem” as a number. Typical measures are misclassification using a confusion matrix for classification problems, and the mean-squared error for regression problems. For a formal definition of these performance measures see Hastie et al. (2001, Section 2.4). Other performance measures are the time an algorithm needs to learn a model or the time a learned model needs to predict data.

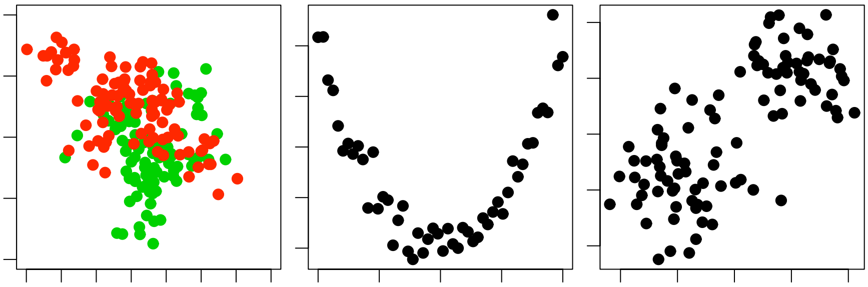

Figure 1.1: A classification problem with two classes `red` and `green`. A regression problem and a cluster analysis problem.

Statistics and machine learning offer a lot of algorithms to tackle these problems given in figure 1.1. Meyer et al. (2003), for example, specify sixteen classification and nine regression algorithms in their paper. Each of them yields a different result, and the data analyst has to decide on the strength of his preferences, experiences and background knowledge about the data which algorithm is the right one. A possible choice and the resultant solution is given in the following figure 1.2:

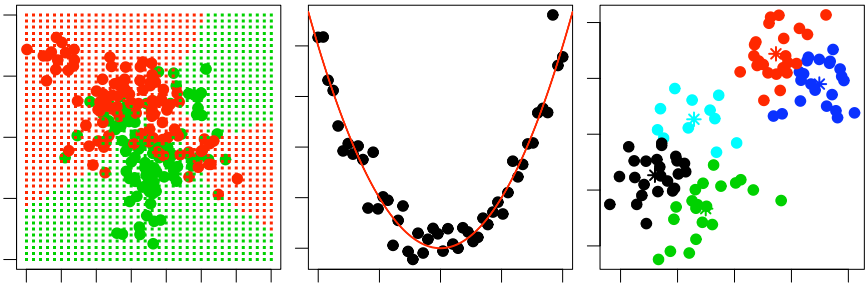

Figure 1.2: Classification: k-nearest-neighbour with 15 neighbours. Regression: quadratic regression. Cluster analysis: k-means with 5 cluster centres.

Let us focus on the task of classification . If the data analyst had made another decision the result would be different (see figure 1.3). And as we can see in the plots, all three methods make misclassifications. So, how do we decide?

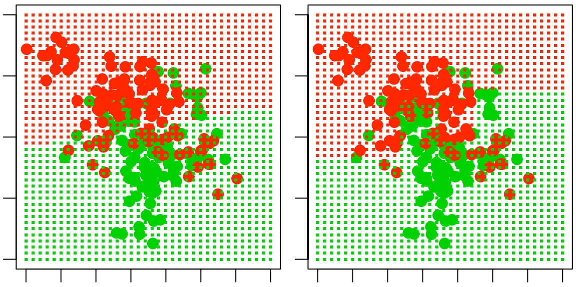

Figure 1.3: Another two results for the classification problem. The right one is obtained with linear discriminant analysis and the left one with classification trees.

One possibility to make a decision, without depending on the preferences and experiences of the analyst, is to learn the problem with all available methods and then use the one with the highest performance measure. This is a very time consuming process, especially if one uses benchmarking methods to ensure the statistical correctness of the decision and after determining the best algorithm, all other benchmark results are not used anymore.

Now, Meta-Learning is the idea to collect these results and use them for the next problem. A new problem is compared with the known ones and the performance measures of similar problems are used as suggestion. Therefor, meta-learning is just machine learning with another level of scope (Vilalta and Drissi, 2002). According to the learning problem definition of Mitchell the task T is to give method suggestions for a problem, and the experience E consists of the methods performance measures on old problems. The performance measure P is the difference between a suggestion and the true performance measure.

To clarify that this approach works, one has to look into the theoretical background of machine learning methods. I use the framework of computational learning theory (chapter 7 in Mitchell, 1997; Angluin, 1992) to explain the coherences.

1.1.1 Theoretical Remarks about Classification

Classification is defined (according to Mitchell, 1997, chapter 7) by a set X of all possible input tuples, a set Y of all possible nominal outputs and the target function $f : X \to Y$.

This function associates each input tuple with a class, i.e. the class label of an input tuple $x \in X$ is $f(x) \in Y$.

The problem is that normally we do not know the target function f, but have a finite set of training samples $D = \{\langle x, f(x)\rangle | x \in X\}$ available. Based on this knowledge, one would like to approximate the target function f by a hypothesis $h : X \rightarrow Y$ as well as possible. The hypothesis function h is element of a hypothesis space H which is a set of functions from X to Y, where each function could approximate f.

The hypothesis space H is defined through a learning algorithm l. This function maps the training examples D to a hypothesis $h \in H$. As one can imagine, different learning algorithms describe different classes of functions and make different assumptions to generalize the examples given. This is called the inductive bias and this is the reason why an algorithm is better than the rest for specific types of problems. Figure 1.4 is an attempt to visualize this.

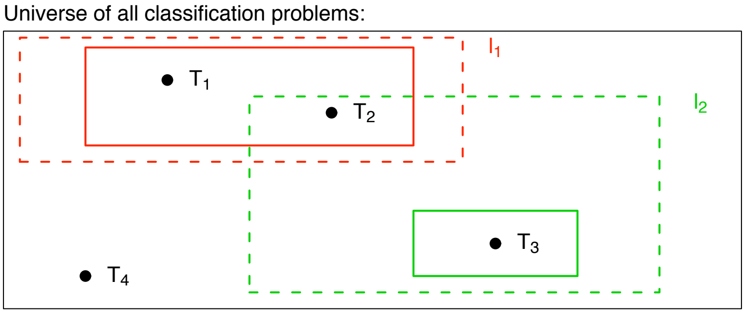

Figure 1.4: (Idea from Vilalta and Drissi, 2002) Visualisation of two learning algorithms, their hypothesis space (dashed lines) and their favored hypothesis space (solid lines). T_1 and T_2 are best learned by l_1. T_2 could also be learned by l_2 but the result would be bad, because through the inductive bias. T_3 is best learned by l_2 and no algorithm can learn T_4.

Additional to the inductive bias, no "super" algorithm exists which performs better on all possible problems than all others. This is a result of the "no free lunch theorems" (see Wolpert, 2001), for a visualisation see the following figure 1.5. It shows two algorithms, one specialized and one general-purpose. The first one is specialised on a specific problem class and achieves very high performances on these problems, but low performances on problems of other classes. On the other hand, the general-purpose algorithm achieves similar performances on all problems. But its performances are, of course, not as high as the performances of the specialised algorithm on its problem class.

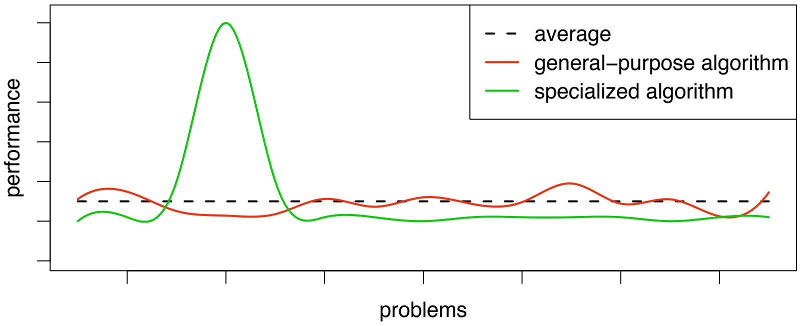

Figure 1.5: Visualisation of the “no free lunch theorem” with a general-purpose algorithm and an algorithm which is specialised on a specific problem class.

Referring to those two facts the decision of the right algorithm proves a difficult problem. There is no “super” algorithm and by reaching a selection the hypothesis space is fixed without having a hint, whether the problem can be best explained through the hypothesis in the fixed space.

1.2 Meta-Learning

The theoretical idea behind meta-learning is best explained with figure 1.4 about the inductive bias of algorithms. Through the selection of an algorithm, the hypothesis space, where to search for a good approximation of the problem, is fixed. Now, meta-learning is the task to choose the hypothesis space, and hence the corresponding best algorithm or best combination of algorithms (which is called a composite classifier), based on the experience gained on other problems. Therefor, for a given set of candidate algorithms L and with respect to performance measure P, the three central questions meta-learning can give answers to, are (after Giraud-Carrier et al., 2004):

1. Which one is the best algorithm $l \in L$?
2. Which is the preferred ordering $l_1 \leq l_2 \leq \ldots \leq l_n$, $l_i \in L$?
3. What is the form of the composite classifier?

Various techniques have been developed to answer the above questions. A detailed list is given in Vilalta et al. (2004), this is a short overview of the main directions :

Meta-Learning for machine learning The characterization of datasets can be performed using a variety of statistical, information-theoretic, and model-based approaches. Match meta-features to algorithms and use this information for model selection or ranking.

Combining algorithms Information collected from the performance of a set of learning algorithms at the base level can be combined through a meta-learner.

Inductive transfer and learning to learn Within the learning-to-learn paradigm, a continuous learner can extract knowledge across domains or tasks to accelerate the rate of learning convergence.

Dynamic bias selection The learning strategy can be modified in an attempt to shift this strategy dynamically. A meta-learner in effect explores not only the space of hypotheses within a fixed family set, but, in addition, the space of families of hypotheses.

The goal of this thesis is, to use the meta-learning for machine learning approach to answer question number one and two. I use statistical and information-theoretic measures (the meta-features) to characterize a dataset. Together with the benchmark results of candidate algorithms on a dataset, is this the experience E or the meta-knowledge base.

Designated to this approach, the second important point is the interpretation of the meta-knowledge base. The issue is how to estimate the performance measures of the candidate algorithms for a new unknown problem and derivate the best algorithm or a ranking of them.

There are different approaches. In Brazdil and Soares (2000) the authors compare ranking methods which only use the performance measures. In Soares and Brazdil (2000) the same authors introduce a method called zooming which uses the meta-features to use only performance measures of the k-nearest datasets. Köpf et al. (2000) and Bensusan and Kalousis (2001) solve the meta-learning problem as regression and Kalousis (2002) uses a very highly sophisticated approach “which closely simulates the evaluation and comparison process followed by an analyst when he has to select among a set of inducers the one that achieves the best accuracy” (meta-learning as classification).

I introduce the three main approaches and show a solution for the meta-learning as regression problem approach and the zoomed ranking approach.

1.3 The Framework

In this section I want to centralise the above said and establish a concrete framework . Vilalta et al. (2004) and Kalousis (2002) each describe one in their paper. I combine ideas from both and adapt them to my idea of meta-learning. Figure 1.6 illustrates the architecture of my framework.

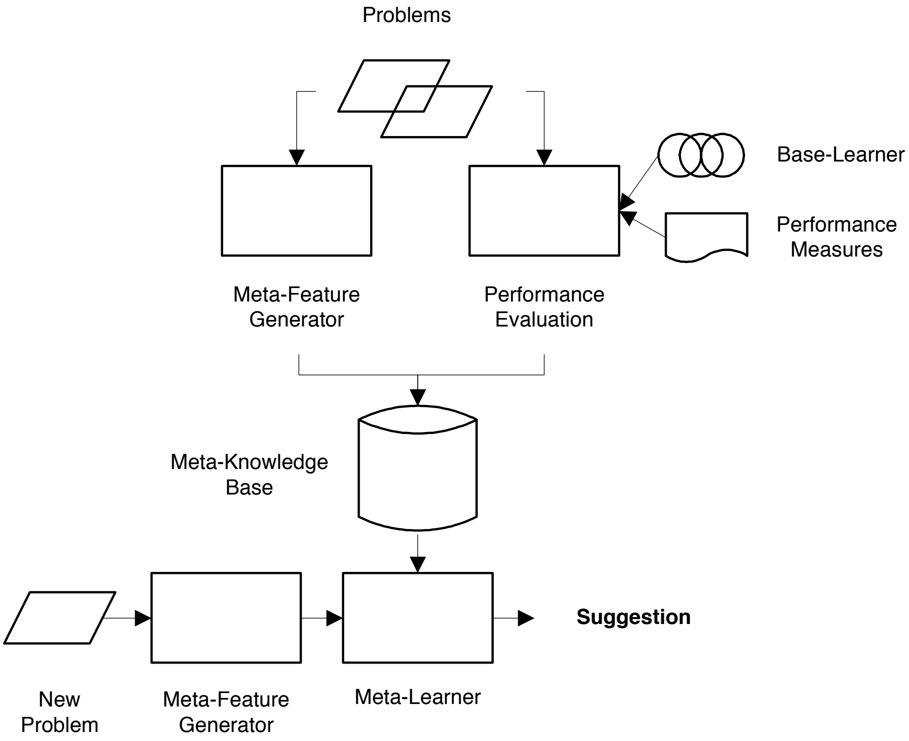

Figure 1.6: The meta-learning framework. The conceptional main parts are performance evaluation, meta-feature generation and meta-learning.

Conceptually, the framework (and the following chapters of this thesis) splits into three parts:

Performance evaluation Define some candidate algorithms (or base-learner) and performance measures. Evaluate the performance on a set of problems (chapter 2).

Meta-Feature generation Generate a characterisation of a problem using statistical and information-theoretic measures (chapter 3). Together with the performance evaluation results this is the meta-knowledge base (the experience E).

Meta-Learning Use the meta-knowledge base to give algorithm suggestions (in form of the best or a ranking) for a new problem (chapter 4).

Additionally, an important point is the description of the problems. In this thesis, a problem is given in table form. The rows are the samples, one column is the response and the other columns are the attributes (or input) of the problem. But a simple table, matrix or data frame cannot express this relation of the attributes. Therefor, I developed an abstraction of the dataset concept, with optimisation for machine learning methods and benchmark experiments, see section 1.4.

1.3.1 Implementation

The implementation of the framework is done using the `R` language and environment[1] and the `S4` classes and methods mechanism (Chambers, 1998). I use this environment because it provides all the machine learning algorithms and statistical tests I need, therefor I can focus on writing the framework and the meta-learning parts.

The implementation splits into three packages, `dataset`, `bench` and `latem`. The first one is the extension of the data set concept as annotated above. It is excluded into a separate package because both other packages need this concept. `bench` contains the performance evaluation parts and `latem` the meta-feature generation and meta-learning methods.

The idea and the design of the packages are explained at the end in the corresponding chapters. I use simple UML notation to show interesting parts of the classes. The `R` implementation is presented in short usage examples. Documentation of the classes, methods and functions, and more information like vignettes, examples and demos can be found in each package. At the present time they are available at `http://mjae.net/mlnoi/`, but the idea is to make them available at CRAN[2], whereas then some structural changes will be accomplished.

[1] Version 2.3.1, `http://www.r-project.org/`.

[2] The Comprehensive R Archive Network, `http://cran.r-project.org/`.

1.4 The dataset Package

Data can be available in many forms. The simplest one is data in table form (rows are samples and columns are attributes), but it can also be a data generating process which gives you an example every time you ask. This package abstracts the data concept and defines an interface how a dataset has to look like, never mind in which form the data are available. Once this interface is defined, it is quite plain to simplify the handling with data for machine learning methods and benchmark experiments.

1.4.1 Concept

Data make no sense until the attributes are related together. You have to define the semantic of the data, i.e. at minimum which ones are the input attributes and which one is the response attribute. This description is interpreted and the two basic aspects, input and response, are defined. Of course, a lot of other aspects of the data could be interesting, e.g. the design matrix, all attributes of nominal type or a scaled view of data, thus the `DataSet` implementation of the `IDataSet` interface allows to set own aspects.

In short, a dataset must allow access to all data (`dataset()`) and to different aspects (`get(...)`). Furthermore, there must be some basic operations, like `dimension()` and `completeCases()`, which provide some information about the encapsulated data. Figure 1.7 shows the `R` related interface definition in simple UML notation:

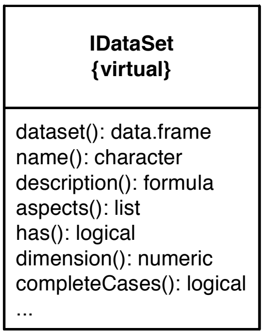

Figure 1.7: The `IDataSet` interface definition.

As said above, the `DataSet` class is an implementation of the interface which encapsulates data of the type `data.frame`. The concrete `R` implementation uses ideas from the `modeltools` package from Torsten Hothorn, Friedrich Leisch, and Achim Zeileis.

1.4.2 Machine Learning related Extensions

Because of the topic of this thesis, I thought about an optimisation of the dataset concept for benchmark experiments and machine learning methods.

Simplified, the benchmark process splits the dataset into pieces, e.g. learning and testing set, and passes them to the candidate algorithms. From the point of view of the benchmark process the splitting is a restriction of the view of the dataset with respect to the samples (rows). The candidate algorithms do not know anything about the restriction, for them this is the whole information about the problem.

On the other hand, a candidate algorithm has to ensure that it can work with the data. This means for example, that there are no incomplete cases, or all attribute types are processable. The first one is again a restriction of the samples view (rows) and the second one is a change in the view of an attribute (column).

In short, a dataset optimised for machine learning must allow to change the view of the data in respect to the samples and attributes without alter the original data, and look like a normal dataset from outside. The following figure 1.8 shows two implementations of the `IDataSet` interface, whereby the first one allows sample and the second one column related view changes:

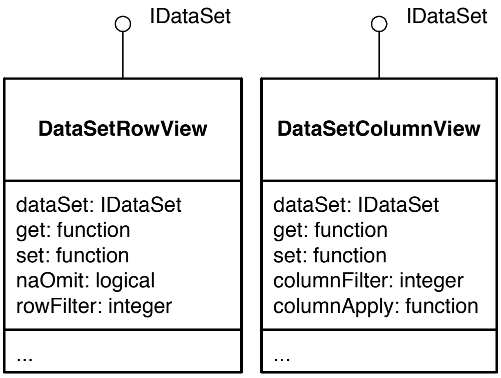

Figure 1.8: Two implementations of the `IDataSet` interface, `DataSetRowView` allows changes in the view of rows and `DataSetColumnView` allows changes in the view of columns.

Since this implementations encapsulate a `IDataSet` and both views implement this interface, it is possible to create a chain of views and this is exactly what happens in the above described benchmark process.

1.4.3 Usage Example

```
> library(dataset)
```

To demonstrate the usage of this package, I use the `Feldmaus` dataset. This is the result of an analysis of two kinds of field mice. The dataset consists of 5 input attributes (`length`, `width`, `height`, `weightclass`, `species`) and the response `sex`.

```
> data(Feldmaus2)
> str(Feldmaus2)
```

```
`data.frame':        40 obs. of  6 variables:
 $ length : num  13 14 14 16 15 13 13 15 14 12 ...
 $ width  : num  9 NA 7 8 7 8 9 7 10 9 ...
 $ height : num  3 2 4 5 3 3 5 3 3 4 ...
 $ weightc: Ord.factor w/ 3 levels "low"<"medium"<..: 1 1 3 3 1 2 2 2 3 3 ...
 $ species: Factor w/ 2 levels "californicus",..: 2 2 2 2 2 2 2 2 2 2 ...
 $ sex    : Factor w/ 2 levels "man","woman": 1 1 1 1 1 1 1 1 1 1 ...
```

Create a `DataSet` object which encapsulates the original data.frame. Additionally to the data, it knows some semantic of the data (through the formula) and its name:

```
> ds = dataSet(sex ~ ., data = Feldmaus2, name = "Feldmaus2")
```

```
Feldmaus2: DataSet
        sex ~ length + width + height + weightc + species
        with 40 samples,
        and the aspects input, response.
```

As you can see, the DataSet constructor interprets the formula and prepares two aspects input and response. One can access aspects with the get method. The result is the aspect definition interpreted on the encapsulated data:

```
> str(ds@get("input"))
```

```
`data.frame':        40 obs. of  5 variables:
 $ length : num  13 14 14 16 15 13 13 15 14 12 ...
 $ width  : num  9 NA 7 8 7 8 9 7 10 9 ...
 $ height : num  3 2 4 5 3 3 5 3 3 4 ...
 $ weightc: Ord.factor w/ 3 levels "low"<"medium"<..: 1 1 3 3 1 2 2 2 3 3 ...
 $ species: Factor w/ 2 levels "californicus",..: 2 2 2 2 2 2 2 2 2 2 ...
```

With the set method, one can define one's own aspects. At this time the definitions can be of the type formula or function.

Suppose we want to benchmark some machine learning algorithms A_1 and A_2 with this dataset. First we have to split the dataset into a training and testing set. The split is done using various methods (e.g. cross-validation), the sample indices for the training set are given in train.in. In the sense of the dataset package, this is a restriction of the row view:

```
> ds.train = dataSetRowView(ds, rowFilter = train.in)
```

```
Feldmaus2: DataSetRowView -> DataSet
        sex ~ length + width + height + weightc + species
        with 30 samples,
        and the aspects input, response.
```

The benchmark process hands over the training dataset to the algorithms. Each algorithm can have its own view of the data. A_1, for example, cannot handle NA values. Hence, it has to ensure that the data does not contain incomplete samples. Again, in the sense of the package, this is a restriction of the row view. A_1 encapsulates the training data and omits incomplete samples:

```
> A1.ds = dataSetRowView(ds.train, naOmit = TRUE)
```

```
Feldmaus2: DataSetRowView -> DataSetRowView -> DataSet
        sex ~ length + width + height + weightc + species
        with 29 samples,
        and the aspects input, response.
```

Algorithm A_2 cannot handle ordinal attributes. In this case, A_2 has to ensure that all ordinal attributes are converted to, for example, a numeric representation. In terms of the dataset package, this means that we change the view of a column:

```
> A2.ds = dataSetColumnView(ds.train, columnApply = function(x) {
+     if (is.ordered(x))
+         return(as.numeric(x))
+     else return(x)
+ })
```

```
Feldmaus2: DataSetColumnView -> DataSetRowView -> DataSet
        sex ~ length + width + height + weightc + species
        with 30 samples,
        and the aspects input, response.
```

If we request for data, the function is applied to all columns, and so we achieve that all columns of type `ordered` are converted to a numeric representation, i.e. columns of type `numeric`:

```
> str(A2.ds@get("input"))
```

```
`data.frame':        30 obs. of  5 variables:
 $ length : num  13 14 14 16 15 13 13 15 14 12 ...
 $ width  : num  9 NA 7 8 7 8 9 7 10 9 ...
 $ height : num  3 2 4 5 3 3 5 3 3 4 ...
 $ weightc: num  1 1 3 3 1 2 2 2 3 3 ...
 $ species: Factor w/ 2 levels "californicus",..: 2 2 2 2 2 2 2 2 2 2 ...
```

Case Study: Starting Shot

The case study is based on the article *The support vector machine under test* by Meyer et al. (2003). In this article, the authors benchmark a popular support vector machine implementation (libsvm) to 16 classification methods on 21 datasets and 9 regression methods on 12 datasets. The datasets are available from `http://www.ci.tuwien.ac.at/~meyer/benchdata/`.

I use the classification datasets to successively explain each part of the meta-learning framework. The datasets contain real and artificial problems of all sizes, table 1.1 shows a short summary .

Problem	#Attributes		#Samples		Class distribution
	nominal	continuous	complete	incomplete	(%)
promotergene	57		106		50.00/50.00
hepatitis	13	6	80	75	20.65/79.35
Sonar		60	208		53.37/46.63
Heart1	8	5	296	7	54.46/45.54
liver		6	345		42.03/57.97
Ionosphere	1	32	351		35.90/64.10
HouseVotes84	16		232	203	61.38/38.62
musk		166	476		56.51/43.49
monks3	6		554		48.01/51.99
Cards	9	6	653	37	44.49/55.51
BreastCancer	9		683	16	65.52/34.48
PimaIndiansDiabetes		8	768		65.10/34.90
tictactoe	9		958		34.66/65.34
credit		24	1000		70.00/30.00
Circle (*)		2	1200		50.67/49.33
ringnorm (*)		20	1200		50.00/50.00
Spirals (*)		2	1200		50.00/50.00
threenorm (*)		20	1200		50.00/50.00
twonorm (*)		20	1200		50.00/50.00
titanic	3		2201		67.70/32.30
chess	36		3196		47.78/52.22

Table 1.1: The 21 problems used in the case study. Problems marked with (*) are artificially created. The list ist sorted by the number of samples.

I decided to use six different classification algorithms, namely Linear Discriminant Analysis, Naive Bayes Classifier, K-Nearest Neighbour Classifier, Classification Trees, Support Vector Machines and Neural Networks. Although this selection defines the methodology some details of the algorithms are specific to their implementation, therefore table 1.2 lists the `R` functions and the corresponding package with the version number. Chapter 2 explains each of the methods theoretically and the usage example of the “The `bench` Package” section shows how they are integrated into my framework.

Algorithm		R Function	R Package	Version
Linear Discriminant Analysis	🟥	`lda`	MASS	7.2-27.1
Naive Bayes Classifier	🟨	`naiveBayes`	e1071	1.5-13
K-Nearest Neighbour Classifier	🟩	`knn`	class	7.2-27.1
Classification Trees	🟦	`rpart`	rpart	3.1-29
Support Vector Machines	🟦	`svm`	e1071	1.5-13
Neural Networks	🟪	`nnet`	nnet	7.2-27.1

Table 1.2: Classification methods used in the case study. The coloured rectangles are the colour codes for plots which deal with the algorithms. The `R` function names are also used as abbreviations for the algorithms.

My packages are available in version 1.0. `R` has version number 2.3.1. The experiments run on a workstation with a AMD Sempron 3400+ (2.00 gigahertz) processor and 1 gigabyte main memory.

The complete case study with all algorithm and benchmark definitions and their results are available in the `cs621` package, also at `http://mjae.net/mlnoi/`.

Chapter 2

Machine Learning

In this chapter, I explain the machine learning parts I need for my thesis in detail. I discuss the basic ideas of the six classification methods and the benchmarking of them. Then I outline my implementation of this machine learning part in the R implementation of the S4-class system and use it to determine the “true” ranking of the methods on each dataset of the case study.

2.1 Classification Algorithms

Classification algorithms predict the qualitative value of an outcome attribute given a number of input attributes. Formally, the problem can be stated as follows: given is a finite set of training samples $D = \{(x_1, y_1), \ldots, (x_n, y_n) | x_i \in X, y_i \in Y\}$, X is the set of all possible input tuples defined by the attributes $\langle X_1, \ldots, X_{attr} \rangle$ and Y is the set of all possible nominal outputs $\{C_1, \ldots, C_{cl}\}$. For simplification we assume X_i of continuous nature.

Decision theory for classification tells us that we need to know the class posteriors $p(y|x)$ for optimal classification (see Hastie et al., 2001, section 2.4). The posterior distribution of the classes after observing x is defined by the Bayes rule

$$p(y|x) = \frac{\pi_y p(x|y)}{p(x)},$$

where π_y denotes the prior probabilities of the classes, $p(x|y)$ the densities of distributions of the observations for each class and $p(x)$ the probability of the observations. We classify a observation by the class with maximal $p(y|x)$.

Classification algorithms now model the posterior with different assumptions or approximate the posterior directly.

2.1.1 Linear Discriminant Analysis

Linear discriminant analysis divides the problem space with linear decision functions into seperate regions that are classified as classes $C_1, C_2, \ldots, C_{cl}$. The decision boundaries are hyperplanes and one familiar possibility to get them is by modeling the class posterior $p(y|x)$, i.e. by modeling each class density $p(x|y)$ with the assumptions of multivariate normal distributions and equal covariance matrices Σ:

$$\begin{aligned} C_1 &\sim N(\mu_1, \Sigma) \\ &\vdots \\ C_{cl} &\sim N(\mu_{cl}, \Sigma) \end{aligned}$$

With this assumptions, we can simplify the comparison between two classes using the Bayes rule and define an equivalent description of this decision boundary, the linear discriminant function for each class:

$$\delta_k(x) = x^T \Sigma^{-1} \mu_k - \frac{1}{2} \mu_k^T \Sigma^{-1} \mu_k + \log \pi_k$$

μ_k and Σ are estimated with the sample mean and the within-class covariance matrix.

This term measures the distance of the observation to the class mean according to the Mahalanobis distance, which is basically proportional (not exactly) to the probability that a observation belongs to a particular class. The observation x is classified with the class with the smallest distance.

An example of a linear discriminant analysis solution for a two-class problem is given in section 1.1, figure 1.1 (left panel) shows the problem and figure 1.3 (left panel) is the solution, which shows the linear decision boundary through the problem space.

2.1.2 Naive Bayes Classifiers

The naive Bayes classifier models the class posterior with the assumption that the attributes are conditionally independent in each class. This means for the class densities that:

$$p(x|y = k) = \prod_{i=1}^{attr} p(x^i | y = k)$$

Although this "naive" assumption is not always accurate, it does simplify the classification task, since it allows the class conditional densities to be calculated separately for each attribute, i.e. it reduces a multidimensional density estimation problem to a number of one-dimensional kernel density estimation ones.

2.1.3 k-Nearest Neighbour Classifiers

The k-nearest neighbour classifier is memory-based (or instance-based) and requires no model to be fit. Given an observation x, the classifier finds the k nearest samples in the training set according to a distance measure d and takes a majority vote among the classes of these samples. This is equivalent to estimating the posterior probabilities $p(y|x)$ by the proportions of the classes in the neighbourhood.

The best choice of k depends upon the data. Generally, larger values of k reduce the effect of noise on the classification, but make boundaries between classes less distinct. The parameter k is determined with cross-validation between 1 and $\sqrt{n}$.

The k-nearest neighbour solution of the classification problem in figure 1.1 is shown in figure 1.2 (left panel) with 15 neighbours.

2.1.4 Recursive Partitioning Trees

Tree-based methods partition the problem space into a set of rectangles, and then fit a simple model in each one. In the task of classification each rectangle is a probability distribution over the classes and the assigned class is the one with the highest probability, thus a classification tree directly models the posterior $p(y|x)$.

The decision tree is created with a recursive procedure of binary splits of the form $X_j \leq s$ versus $X_j > s$. The algorithm starts with all data and has to find the best splitting variable j and split point s. This is done by calculating a node impurity measure and taking the split with the minimum impurity over all allowed splits. The split partitions the data into two resulting regions and the algorithm repeats the splitting process on each of the two regions. Then this process is repeated on all of the resulting regions until some stopping rule is applied.

The node impurity is defined with the probability distributions p_{ik} over the classes k at each node i. As a node represents a rectangle R_i of the partitioned problem space containing n_i samples (n_{ik} random samples from the multinomial distribution of each class), the proportion of class k at node i can be estimated by

$$p_{ik} = \frac{1}{n_i} \sum_{x_j \in R_i} I(y_j = k)$$

and the observations are classified to class $k(i) = argmax_k(p_{ik})$. Then, we can define different measures of node impurity:

$$\text{Misclassification error: } \frac{1}{n_i} \sum_{x_j \in R_i} I(y_j \neq k(i)) = 1 - p_{ik(i)}$$

$$\text{Gini index: } \sum_{k \neq k'} p_{ik} p_{ik'} = \sum_{k=1}^{cl} p_{ik}(1 - p_{ik})$$

$$\text{Cross-entropy or deviance: } -\sum_{k=1}^{cl} p_{ik} \log p_{ik}$$

The tree construction takes the split with the minimum impurity and continues until the number of samples reaching each leaf is small or the leaf is homogeneous enough, but one should see the effective tree size as the tuning parameter of this algorithm. A large tree might overfit the data, while a small tree might not capture the important structure. Thus the gained tree T_0 is pruned.

The established methodology is cost complexity pruning (Hastie et al., 2001). Pruning considers rooted subtrees of the tree T_0. The cost complexity criterion for a given α, a node impurity measure C (typically the misclassification error) and a tree T with size $|T|$ is

$$C_\alpha(T) = C(T) + \alpha|T|.$$

It can be shown that for each tuning parameter $\alpha \geq 0$ there is a smallest subtree T_α that minimizes the cost complexity criterion $C_\alpha(T)$. To find this tree, the internal nodes that produce the smallest per-node increase in $C(T)$ are successively collapsed, until the single root node is gained. This sequence of subtrees contains the tree T_α. The parameter α is the tradeoff between tree size and its goodness off fit to the data and the best value is achieved by k-fold cross-validation.

The classification tree solution of the classification problem in figure 1.1 is shown in figure 1.2 (right panel) where we see the decision boundaries and the indicated rectangles.

2.1.5 Support Vector Machines

Support vector machines are a generalization of linear decision boundaries, they produce nonlinear boundaries by constructing a linear boundary in a high-dimensional, transformed version of the problem space. The following introduction reduces the classification problem to a two-class problem with $y_i \in \{-1, 1\}$, the generalization is done with solving a two-class problem for each pair of classes and the majority vote amongst these results.

Assume the linear separability of the training data. The data are separable by a hyperplane

$$\{x | f(x) = w \cdot x + b = 0\}$$

with $b \in \mathbb{R}$ and $w \in \mathbb{R}^{attr}$ a unit vector, thus $\|w\| = 1$. An observation is classified with $k(x) = sign(w \cdot x + b)$. Normally there are infinitely many separating hyperplanes, so which one is the best? The idea is to use that one that separates the data with equal distance from both classes and has a maximal margin. The margin is the distance between the closest points of both classes. This hyperplane is called the maximum-margin hyperplane.

Since $f(x) = w \cdot x + b$ returns the distance from the hyperplane $f(x) = w \cdot x + b = 0$ the maximum margin search can be defined as optimization problem:

$$\max_{w,b,\|w\|=1} C$$

$$\text{subject to } y_i(x_i \cdot w + b) \geq C,\ i = 1, \ldots, n$$

C is the distance from the hyperplane to the first sample on each side, thus the margin is $2C$ wide.

Suppose now the classes overlap in the problem space. To tackle this problem, we allow some points to be on the wrong side of the margin, i.e. we introduce slack variables $\xi = (\xi_1, \ldots, \xi_n)$. The constraints then are modified to

$$y_i(x_i \cdot w + b) \geq C(1 - \xi_i),\ \forall i$$

with $\xi_i \geq 0$ and $\sum_{i=1}^n \xi_i \leq$ constant. Is a sample x_i on the wrong side or within the distance C, then the slack variable ξ_i has an amount proportional to the distance to the right side. Through the constraint $\sum_{i=1}^n \xi \leq$ constant the number of misclassifications is bounded. To simplify the computation, it is possible to re-formulate the problem as equivalent minimization problem:

$$
\begin{aligned}
&\min_{w,b} \frac{1}{2}\|w\|^2 + \gamma \sum_{i=1}^{n} \xi_i \\
\text{subject to } &\xi_i \geq 0 \\
&y_i(x_i \cdot w + b) \geq 1 - \xi_i,\ i = 1, \ldots, n
\end{aligned}
$$

Where the constant for the bounding of the misclassifications is replaced by the tuning parameter γ.

The solution of this quadratic programming problem is gained using Lagrange multipliers and the Karush-Kuhn-Tucker conditions (see Hastie et al., 2001, page 374). It provides an estimator for w and b. The w estimator has the form

$$\hat{w} = \sum_{i=1}^{n} \alpha_i y_i x_i,$$

with $0 \leq \alpha_i \leq \gamma$. For the solution only the samples with nonzero α_i contribute to, hence they are called the support vectors. All in all the solution of the optimization problem is $\hat{f}(x) = \sum_{i=1}^{n} \alpha_i y_i x_i + \hat{b}$ and the decision function is $k(x) = sign(\hat{f}(x))$.

So far the decision boundaries are linear in the problem space. The generalization to non-linear decision boundaries follows the idea to map the samples to a higher dimensional space, linearly separate them, and translate the solution back to non-linear boundaries in the original space.

The enlargement of the problem space is done by applying basis expansions $h_m(x), m = 1, \ldots, M$ on the samples $h(x_i) = (h_1(x_i), h_2(x_i), \ldots, h_M(x_i))$. This results in the adapted non-linear solution function

$$\hat{f}(x) = (\hat{w} \cdot h(x)) + \hat{b},$$

which we can rewrite using inner products:

$$\hat{f}(x) = \sum_{i=1}^{n} \alpha_i y_i \langle h(x), h(x_i) \rangle + \hat{b}$$

In fact, the transformation $h(x)$ is not specified at all, the only requirement is the knowledge of the kernel function

$$K(x, x') = \langle h(x), h(x') \rangle$$

that computes the inner product in the transformed space (what is called the kernel trick). A popular kernel for support vector machines is the radial basis kernel:

$$K(x, x') = \exp \frac{-\|x - x'\|^2}{c}$$

This approach has two tuning parameters, the cost of constraints violation γ and the kernel parameter c. Following Meyer et al. (2003) the best choices are determined with a grid search over the two-dimensional parameter space (γ, c), γ ranges from 2^{-5} to 2^{12} and c from 2^{-10} to 2^{5}.

2.1.6 Neural Networks

The central idea of neural network classifiers is to extract linear combinations of the problem attributes as derived features, and then model the posterior $p(c|x)$ as a non-linear function of these features.

A typical (feed-forward) neural network consists of three layers: input x, hidden h and output y. In each layer are a number of units, the input layer has as much units as the problem attributes ($attr$), the number of hidden units M is the tuning parameter and the number of output units is the number of different classes cl. Each input unit is weighted connected with each unit of the hidden layer and each hidden unit is weighted connected with each output unit. Sometimes there is an additional bias unit feeding into every unit in the hidden and output layer with constant "1" as input which captures the intercepts of the statistical model.

Statistically neural networks are modeled in the following way: the hidden units h_m are created from linear combinations of the inputs x,

$$h_m = \sigma(\alpha_{0m} + \alpha_m^T(x^1, \ldots, x^{attr})),\ m = 1, \ldots, M,$$

then the target y_k is modeled as a function of linear combinations of the h_m,

$$\begin{aligned} t_k &= \beta_{0k} + \beta_k^T(h_1, \ldots, h_M),\ k = 1, \ldots, cl \\ y_k(x) &= g_k(t_1, \ldots, t_k),\ k = 1, \ldots, cl. \end{aligned}$$

α_{0m} and β_{0k} are the intercepts and represent the bias. α_m is the vector of the weights between the hidden unit h_m and the input units, respective β_k is the vector of the weights between the output unit t_k and all hidden units. The activation function $\sigma(v)$ is usually chosen to be the sigmoid

$$\sigma(v) = \frac{1}{1 + e^{-v}}$$

and the output function $g_k(t_1, \ldots, t_k)$ is the softmax function

$$g_k(t_1, \ldots, t_k) = \frac{e^{t_k}}{\sum_{i=1}^{cl} e^{t_i}}.$$

To make a neural network model fit the training data well, the parameters $\{\alpha_{0m}, \alpha_m | m = 1, \ldots, M\}$ and $\{\beta_{0k}, \beta_k | k = 1, \ldots, cl\}$, often called the weights, need to be determined with the two-pass back-propagation procedure. The idea of this algorithm is that in the forward pass the current weights are fixed, the predicted values are computed and the error of the network is calculated. And in the backward pass each weight is adapted with respect to the current effect of the unit on the error. The concrete derivation can be found in Hastie et al. (2001, section 11.4).

As mentioned above the tuning parameter of this classifier is the number of hidden units. The best value is searched between 1 and $log(n)$ (again following Meyer et al., 2003).

2.2 Comparison and Selection of Methods

What we now need is the possibility to check how good an algorithm is on a specific learning problem and which algorithm is the best amongst a set of candidate algorithms.

Commonly used techniques are cross-validation or resampling to derive point estimates of the performances which are compared to identify algorithms with good properities. However, the problem of identifying a superior algorithm is structurally different from the performance assessment task, because the comparison of raw point estimates in finite sample situations does not take their variability into account, thus leading to uncertain decisions without controlling any error probability.

A framework which takes this variability into account is introduced in the paper *The Design and Analysis of Benchmark Experiments* by Hothorn et al. (2005) . The authors show "how one can sample from a well defined distribution of a certain performance measure, conditional on a data generating process, in an independent way". Thus, standard statistical test procedures can be used to test many hypotheses of interest without any restrictions or additional assumptions, neither to the candidate algorithms nor to the data generating process.

The benchmark situation is the following: given is a data generating process DGP from which B independent and identically distributed training datasets haven been drawn:

$$D^b = \{(x_1, y_1)^b, \ldots, (x_n, y_n)^b\} \sim DGP,\, b = 1, \ldots, B$$

There are $K > 1$ two-step candidate algorithms, whereas for each algorithm the function $l_k(\cdot|D^b)$ is the fitted model based on the observations D^b. This function itself has a distribution $\mathcal{L}_k$ as it is a random variable depending on D^b:

$$l_k(\cdot|D^b) \sim \mathcal{L}_k(DGP),\, k = 1, \ldots, K$$

The performance of the candidate algorithm l_k when provided with the training data D^b is measured by a scalar function p:

$$p_{kb} = p(a_k, D^b) \sim \mathcal{P}_k = \mathcal{P}_k(DGP)$$

p_{kb} is a random variable and follows a distribution function $\mathcal{P}_k$ which again depends on the data generating process. These performance distributions of the candidate algorithms can be used to compare the algorithms by formulating a hypothesis test:

$$\begin{aligned} &H_0 : P_1 = \cdots = P_K \\ &H_A : \exists i, j \in \{1, \ldots, K\} : P_i \neq P_j \end{aligned}$$

Often in practical situations (real-world problems) the data generating process is unknown, but a single training dataset $D \sim \mathcal{Z}_n$ of n observations from some distribution function $\mathcal{Z}$ is available. The data generating process then is mimicked by the empirical distribution function of the training dataset: $DGB = \mathcal{Z}_n$. B independent training datasets are drawn by bootstrapping $D^1, \ldots, D^B \sim \mathcal{Z}_n$, and the test dataset T is defined in terms of the out-of-bootstrap observations:

$$T^b = D \setminus D^b,\, b = 1, \ldots, B$$

Thus the models are fitted on the training dataset D^b and their performance is evaluated on the corresponding test dataset T^b.

The above described benchmark experiments are block-design procedures and a global test for the null hypothesis $H_0 : P_i \neq P_j$ for any i, j can be done with the Friedman test. To discover the algorithms i, j which are responsible for the rejection of the global null hypothesis one can use the the Wilcoxon-Nemenyi-McDonald-Thompson test (see Hollander and Wolfe, 1999).

2.3 The bench Package

The bench package implements the benchmark process described in the last section. For this purpose, I first define my view of a machine learning algorithm and then explain the implementation of the benchmarking.

2.3.1 Machine Learning Algorithms

My point of view of a machine learning algorithm in one sentence: it is a template, that, instantiated with data, creates a model after a specific method. Therefor, the algorithm object has to know how to create a model, how to predict new data, how to adapt hyperparameters, and what the data the algorithm can work with are. Figure 2.1 shows the corresponding class:

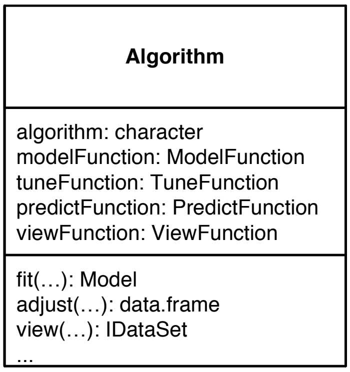
Algorithm
algorithm: character
modelFunction: ModelFunction
tuneFunction: TuneFunction
predictFunction: PredictFunction
viewFunction: ViewFunction
fit(...): Model
adjust(...): data.frame
view(...): IDataSet
...

Figure 2.1: The Algorithm class with all necessary functions to be an machine learning algorithm in my point of view.

The modelFunction is the implementation of the "how to create a model for specific data" method (e.g. support vector machine classifier, neural network classifier, ...). It is used by the fit(...) method and returns an object of the class Model. The Model knows for which data it is model, based on which method, and it knows how to predict new data. Figure 2.2 shows the class.

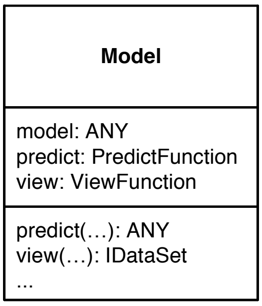

Figure 2.2: The Model class with all necessary functions to be a machine learning model in my point of view.

The `tuneFunction` knows for concrete data how to adjust hyperparameters if the method has one. It is used by the function `adjust(...)`. For example, for support vector machines this could be the implementation of a grid search over (C, γ).

The `predictFunction` knows how to predict the response for new data. Since the `Algorithm` is just the template for new models it cannot predict, so this function is passed to the `Model` object by the `fit` method.

The `viewFunction` implements the algorithm's view of the data. The function gets an `IDataSet` object and can encapsulate it in more dataset views (see section 1.4 about the `DataSet` package) to ensure that the algorithm only sees data it can work with. The important thing is, that this function is used every time the other functions are called, hence the model, tune and predict function only see data through this specified view.

The `algorithm` property specifies the name of the algorithm.

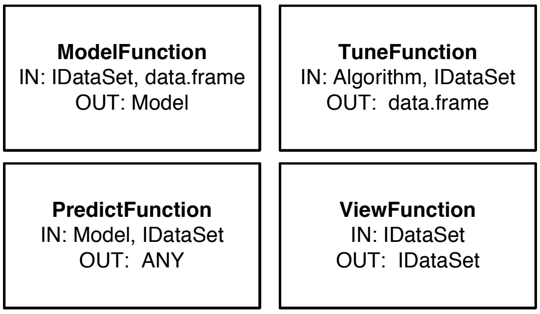

Figure 2.3: The signatures of the functions of an `Algorithm` object. IN specifies the type of the function arguments, and OUT the type of the return value.

2.3.2 Benchmark Experiments

The implementation of the benchmark experiment framework is straight after the section 2.2 and the therein named paper *The Design and Analysis of Benchmark Experiments* from Hothorn et al. (2005).

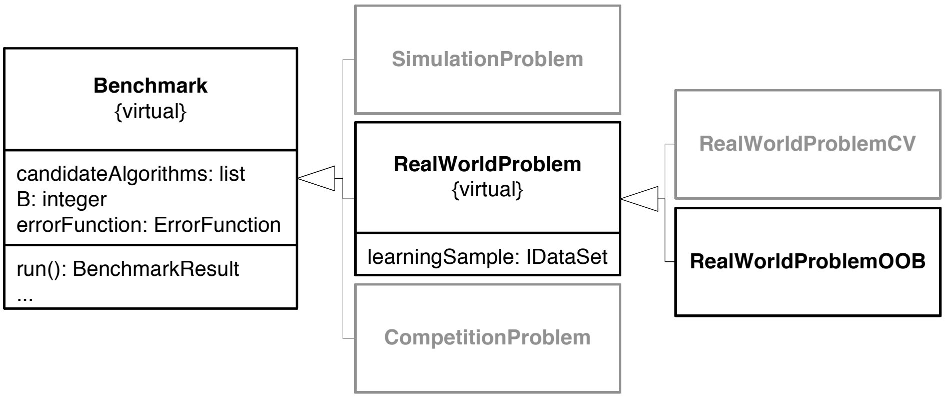

Figure 2.4: The class hierarchy of possible kinds of benchmark experiments. The gray ones are not implemented yet, because this thesis treats all learning problems as real-world problems.

A benchmark experiment is basically described by a `Benchmark` object. It consists of a list of candidate algorithms of the type `Algorithm`, a number `B` which specifies the drawn learning samples from the data generating process, and the error function.

The next hierarchy layer describes the kind of the data generating process for training and testing samples. Since my case study only deals with real-world problems, I only implemented this class and the "out-of-bootstrap strategy" specialisation of it.

An experiment is started with the `run` method which returns a `BenchmarkResult` object. This contains the error values, and various time measures of the `B` passes (see figure 2.5 for class diagram). Based on this result the ranking of the candidate algorithms can be determined. Additionally, I developed the `benchplot` and `pooledbenchplot` to visualise the results of one and many benchmark experiments. They are explained in the usage example and in the case study.

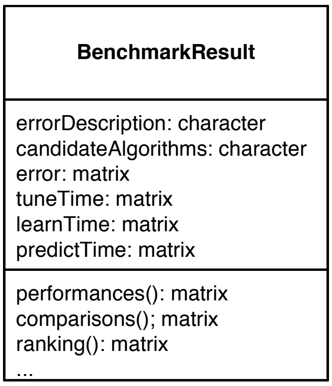

Figure 2.5: The `BenchmarkResult` class diagram.

2.3.3 Usage Example

```
> library(bench)
```

Algorithms

I demonstrate the `R` implementation of my algorithm idea on the basis of the support vector machine classifier and the classification tree method. As I want to use as much existing software as possible, both `Algorithm` objects wrap the corresponding functions. A standalone implementation of an algorithm can be found in the package (`knnS4`).

Let's start with the support vector machines. I use the `svm` function from the `e1071` package. In point of view of the `Algorithm` object, this is the `ModelFunction`. The `svm` function, and most of the other existing machine learning functions, have the signature `(formula, data=..., list of parameters, ...)`. So, if you want to wrap a existing function you do not have to implement the `ModelFunction`, just set the `algorithm` property to the name of the existing function and the `fit` method calls it:

```
> svm_algorithm = "svm"
```

The package also defines a `predict` function, which I want to use, so the `PredictFunction` implementation looks like:

```
> svm_predict = predictFunction("svm_predict",
+     function(model, data) {
+         return(predict(model, newdata = data))
+     })
```

The method cannot work with NA values, hence I fade them out:

```
> svm_view = viewFunction("svm_view",
+     function(dataSet) {
+         return(dataSetRowView(dataSet,
+             naOmit = T))
+     })
```

And the tuning of the hyperparemeter C and γ is done with the tune function from the same package:

```
> svm_tune = tuneFunction("svm_tune",
+     function(algorithm, dataSet) {
+         tuned = tune.svm(description(dataSet),
+             data = dataset(dataSet),
+             gamma = 2^(-10:5),
+             cost = 2^(-5:12), tunecontrol = tune.control(sampling = "cross",
+                 cross = 10, best.model = F,
+                 performances = F))
+         return(tuned$best.parameters)
+     })
```

Now, putting all together, the definition of the support vector machine algorithm is the following:

```
> svm_algorithm = algorithm(svm_algorithm,
+     tuneFunction = svm_tune, predictFunction = svm_predict,
+     viewFunction = svm_view)
```

```
svm: Algorithm
        with no specific model function,
        hyperparameter tuning with the svm_tune function,
        the predict function svm_predict, and
        restrictions of the data through the view function svm_view.
```

An example of where to implement a ModelFunction is the classification tree method because I want to prune a fitted tree with the 1-SE rule (see Venables and Ripley, 2002, page 260). The basic model function is rpart from the rpart package:

```
> rpart_model = modelFunction("1-SE-rule-model",
+     function(dataSet, parameters) {
+         m = rpart(description(dataSet),
+             data = dataset(dataSet),
+             cp = 0, method = "class")
+         p = m$cptable
+         xstd = p[, 5]
+         xerror = p[, 4]
+         cp0 = p[, 1]
+         minpos = min(seq(along = xerror)[xerror ==
+             min(xerror)])
+         rule = (xerror + xstd)[minpos]
+         cp = sqrt(cp0 * c(Inf,
+             cp0[-length(cp0)]))
+         pcp = cp[which(xerror <
+             rule)[1]]
+         return(prune(m, cp = pcp))
+     })
```

Again, I use the predict function from the package, but I'm only interested in the class labels:

```
> rpart_predict = predictFunction("rpart_predict",
+     function(model, data) {
+         return(predict(model, newdata = data,
+             type = "class"))
+     })
```

There are no hyperparameters to tune and the method can handle all types of data, so the classification tree algorithm definition is:

```
> rpart_algorithm = algorithm("rpart",
+     modelFunction = rpart_model,
+     predictFunction = rpart_predict)

rpart: Algorithm
        with 1-SE-rule-model as model function,
        no hyperparameter tuning,
        the predict function rpart_predict, and
        no restrictions of the data.
```

Now, I use the Feldmaus dataset from the dataset package usage example to demonstrate the working with the Algorithm objects:

```
> data(Feldmaus)
> ds = dataSet(Feldmaus, sex ~ .,
+     name = "Feldmaus")
```

Using the view method we can see the data through the "algorithms eyes". For the support vector machine you see that there is one sample fewer because it is incomplete, the classification tree sees all samples:

```
> view(svm_algorithm, ds)

Feldmaus: DataSetRowView -> DataSet
        sex ~ length + width + height + species
        with 39 samples,
        and the aspects input, response.

> view(rpart_algorithm, ds)

Feldmaus: DataSet
        sex ~ length + width + height + species
        with 40 samples,
        and the aspects input, response.
```

The fit method creates a model for the data. The support vector machine model function has hyperparameters, hence we first adjust them to the dataset and then fit the model using them:

```
> svm_hp = adjust(svm_algorithm,
+     ds)

  gamma cost
1   0.5   16
```

```
> svm_model = fit(svm_algorithm,
+     ds, svm_hp)

svm model of the Feldmaus dataset:
        with the the hyperparameters gamma=0.5, cost=16.
```

And the classification tree model of the `Feldmaus` data:

```
> rpart_model = fit(rpart_algorithm,
+     ds)

rpart model of the Feldmaus dataset:
        with no hyperparameters.
```

The prediction of a class for new input data is done with the `predikt` method (no, the `k` is no mistake, it is written this way because I don't want to overwrite the `S3 predict` function). As an example, the prediction of the sample `s1`:

```
  length width height     species sex
1     13     9      3 ochrogaster man

> predikt(svm_model, s1)

[1] man
Levels: man woman

> predikt(rpart_model, s1)

[1] man
Levels: man woman
```

Benchmark Experiments

I use the tictactoe benchmark experiment from the case study to show the usage. The candidate algorithms are `svm_algorithm` and `rpart_algorithm` from above and the following four:

```
> lda_algorithm

lda: Algorithm
        with no specific model function,
        no hyperparameter tuning,
        the predict function lda_predict, and
        restrictions of the data through the view function lda_view.

> naiveBayes_algorithm

naiveBayes: Algorithm
        with no specific model function,
        no hyperparameter tuning,
        the predict function naiveBayes_predict, and
        no restrictions of the data.

> knn_algorithm
```

```
knn: Algorithm
        with knn_model as model function,
        hyperparameter tuning with the knn_tune function,
        the predict function knn_predict, and
        restrictions of the data through the view function knn_view.
```

```
> nnet_algorithm
```

```
nnet: Algorithm
        with nnet_model as model function,
        hyperparameter tuning with the nnet_tune function,
        the predict function nnet_predict, and
        restrictions of the data through the view function nnet_view.
```

The benchmark experiment is formulated as real-world problem with the out-of-bootstrap strategy to draw a learning and testing sample and 250 passes, the error function is misclassification:

```
> bench = rw_oob(candidateAlgorithms = list(lda_algorithm,
+     naiveBayes_algorithm, knn_algorithm,
+     rpart_algorithm, svm_algorithm,
+     nnet_algorithm), B = 250, errorFunction = misclassification,
+     learningSample = tictactoe)
```

```
RealWorldProblemOOB:
        with 250 passes,
        and the candidate algorithms lda, naiveBayes, knn, rpart, svm, nnet.
```

And GO:

```
> bench_result = run(bench)
```

The result is an object of the class `BenchmarkResult` which contains all time measures and error values per pass:

```
> bench_result
```

```
Result of an benchmark experiment (RealWorldProblemOOB):
        with the candidate algorithms lda, naiveBayes, knn, rpart, svm, nnet,
        the error function misclassification,
        and 250 passes, whereof 0 failed.
```

And with the `ranking` method, on can estimate the ranking of the algorithms according to a performance measure and a p value:

```
> ranking(bench_result, p = 0.05,
+     y = "error")
```

```
                 error ranking
svm        0.006493132       1
lda        0.016518527       2
nnet       0.069185581       3
rpart      0.097929701       4
knn        0.171104030       5
naiveBayes 0.294405950       6
```

To visualize the benchmark result, I developed a plot called benchplot:

```
> benchplot(bench_result, y = "error")
```

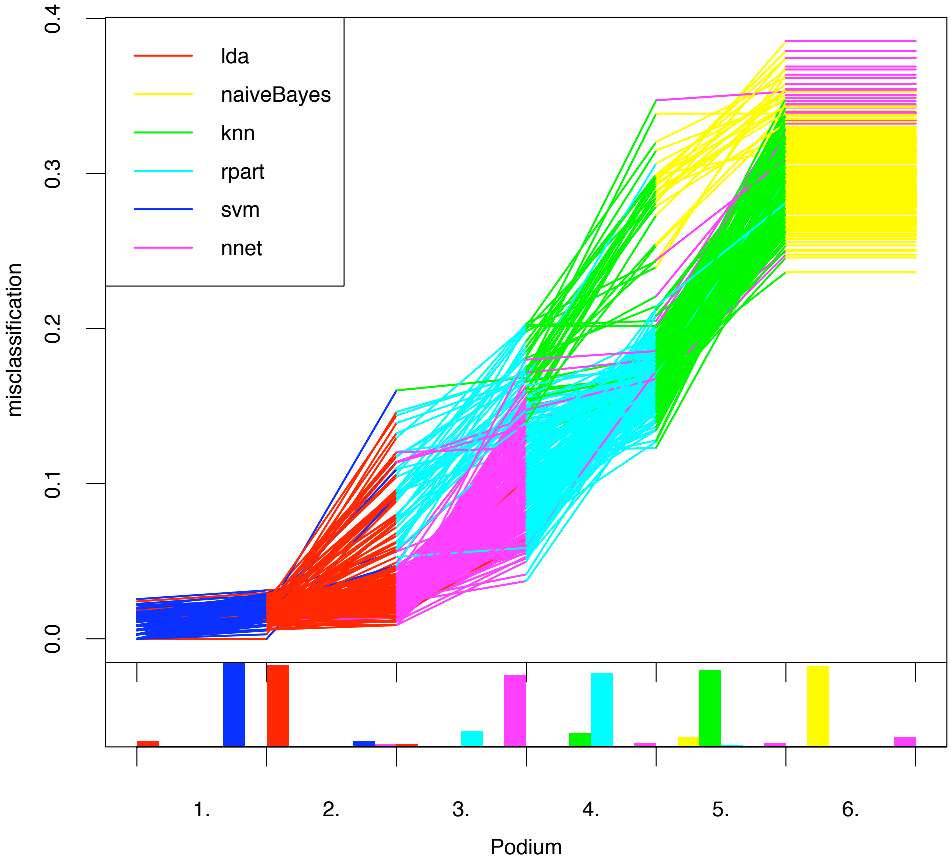

Figure 2.6: Benchplot of the benchmark result of the six candidate algorithms on the tictactoe dataset.

The idea is based upon the parallel coordinate plot. The x-axis is the podium, with places from 1 to the number of candidate algorithms (6), a place is always between two ticks. The y-axis is the performance measure, in this case the misclassification, and each pass of the benchmark experiment is on line. The performance measures per pass are sorted and printed on the left tick of each podium place, then they are linked together and the colour of the line from the left tick to the right tick is the algorithm's colour. Additionally, the barplots at the bottom show the appearance of each algorithm at the corresponding podium place. Some other benchplots can be seen in the appendix 4.3.3 and all benchplots of the case study experiments can be seen in `demo(benchplot)`.

Case Study: “True” Ranking

This part of the case study defines the “true” ranking of the candidate algorithms on each dataset concerning some performance measures. As one goal of meta-learning is the estimation of the “true” ranking, these results are the reference results in later case study parts.

For each dataset a real-world benchmark experiment with 250 (bootstrap) runs is specified. The observed performance measures are misclassification, time of tuning hyperparameters for a dataset, time of learning a model for a specific dataset with the tuned hyperparameter and prediction time of a complete dataset. As one can imagine, the misclassification is the most interesting one, but there may be situations where the creation time of a model (the sum of all three time measures) is of importance too. Moreover, the accumulated creation time, the benchmark time, is interesting, because this is the time meta-learning should shorten (see the case study section about suggestions).

The hyperparamter estimation for (`knn`, `svm`, and `nnet`) and the `rpart` algorithm are as defined in the corresponding sections above. Some definitions of the algorithms in my framework are shown in the usage example of the `bench` package, the others are analogous.

All individual benchmark results are available in the in the `cs621` package. I present some pooled results which are interesting with respect to to meta-learning.

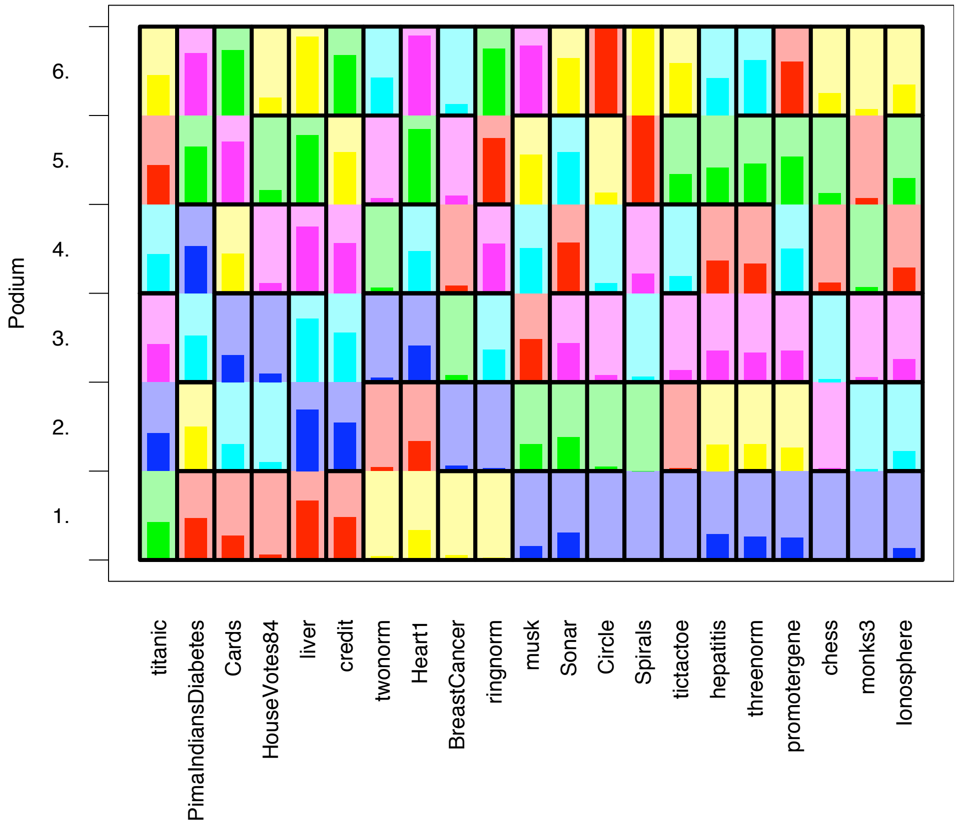

Figure 2.7: Pooled benchplot showing the ranking corresponding to the misclassification with p-value 0.05. The colour codes are 🟥 lda, 🟨 naiveBayes, 🟩 knn, 🟦 rpart, 🟦 svm, and 🟪 nnet.

Figure 2.7 shows the answer to the most interesting question, the ranking regarding to the misclassification. For each dataset the algorithm performances are sorted in an ascending way. The corresponding podium place is stained with the algorithms colour and the performance value is expressed as a percentage with the darker bar.

Furthermore, the results of the significance test (with p-value 0.05) of the performance differences per dataset are shown as pooling of podium places with borders. Pooled podium places indicate that the performance value is different but not significantly different, hence the corresponding algorithms have the same podium place.

As an example, the algorithms performances on the monks3 dataset are:

```
       svm      rpart       nnet        knn        lda naiveBayes
0.01095148 0.01163782 0.02932404 0.03444507 0.03522340 0.03526528
```

But the significance test says, that the difference is not significant for some places and the correct ranking is:

```
     rpart        svm       nnet        lda naiveBayes        knn
         1          1          2          3          3          3
```

Thus, the first two and the last three places are pooled.

To visualize some trends, the datasets in the benchplot are sorted after their 1., 2., . . . algorithm. One question is, if the clusters of the winner algorithms are also seen in the clustering of the characteristics of the datasets (see the next case study part, 3.4.2).

Additional to the above plot, the plot 2.8 about the benchmark time per algorithm and dataset is interesting.

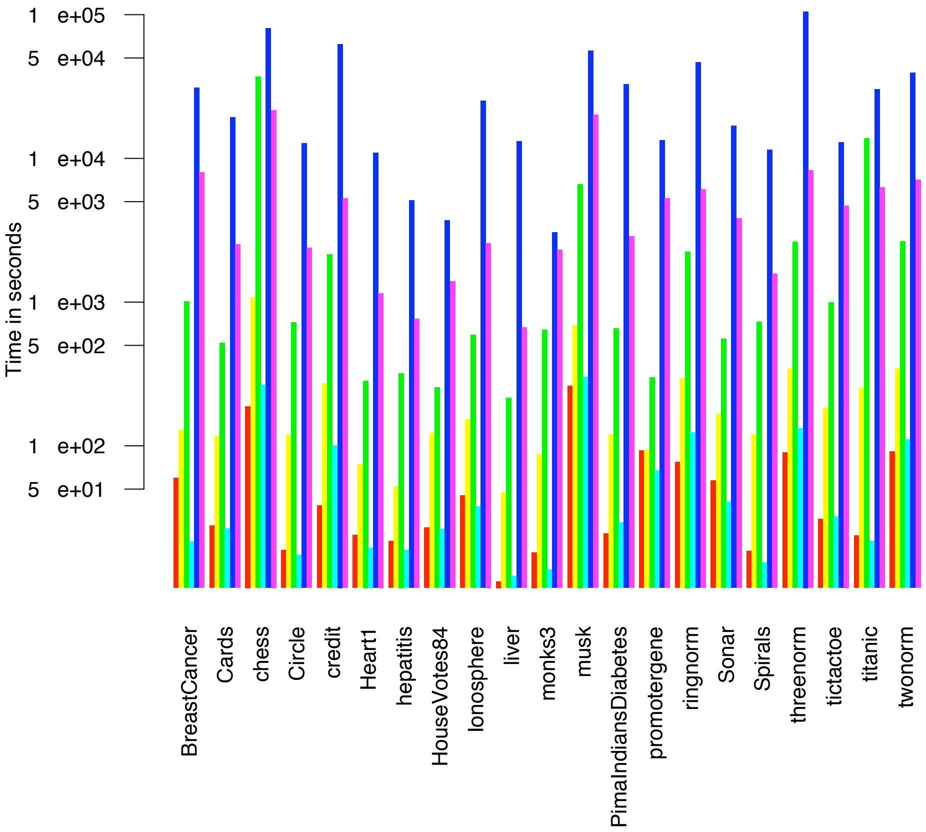

Figure 2.8: Benchmark time per algorithm and dataset. The y-axis scale is logarithmic. The colour codes are the same as in the figure above.

Figure 2.7 shows that support vector machines often are the best algorithm. But the additional time information in figure 2.8 shows that they need a lot of time (because of the huge hyperparameter search space). Now, using the information of the two plots for a specific dataset, say monks3, svm has the lowest performance value but the difference is not significant to the second algorithm rpart. However, the benchmark time differences are significant, svm needs 3056 seconds and rpart 14 seconds. So, if we just wanted the best algorithm, we could say, that we have wasted a lot of time with benchmarking the support vector machines on this dataset. To overcome this problem, some meta-learners use both, time and accuracy information to generate a suggestion.

To complete this case study part and to demonstrate how much time these benchmark experiments took, I present figure 2.9, which shows the total time in hours per dataset.

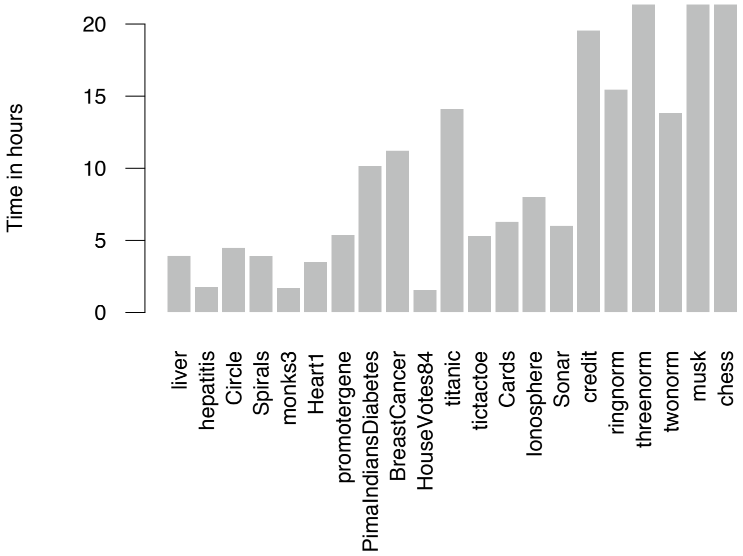

Figure 2.9: Distribution of the 231 hours of benchmarking. The datasets are sorted after their size, i.e. the product of the number of samples and the number of attributes.

Chapter 3

Characterisation of Learning Problems

This chapter explains the statistical and information-theoretic measures I use for the characterisation of a problem. To use them efficiently, I introduce a method to aggregate part results. Then I explain the distance measure and outline the corresponding part of the `latem` package. The case study shows some clustering aspects of the characterisations of the problems and relates them to the performance measures gained in the last case study.

3.1 Introduction

In the past, there were some projects which defined problem characteristics and related them with machine learning methods. One very important one was the StatLog project and their resulting book Michie et al. (1994). Another interesting one is the METAL project. In my thesis I refer to the PhD thesis by Alexandros Kalausis (Kalousis, 2002). He extends the characteristics from the StatLog project and defines an aggregation for one kind of these characteristics. I introduce this representation in the next section and then explain each characteristic theoretically and with some examples. But first, we have to make clear what a dataset in this case is.

In case of classification, a dataset D consists of attributes $X_1, \ldots, X_{attr}$ and one nominal class attribute C with cl different class labels (levels). The nature of the attributes X_i can be continuous or nominal, other levels of measurement are not supported by now. A nominal attribute X_i has I categories. Having another nominal attribute X_j with J categories, one can represent the joint distribution with a contingency table. π_{i+} describes the marginal distribution $p(X_i = x_k)$ of X_i, and analogy π_{+j} describes the marginal distribution $p(X_j = x_k)$ of X_j. The range of a continuous attribute X is from $-\infty$ to ∞.

Hence, a characterisation of a dataset has to describe nominal and continuous attributes and their relation to each other. If a characteristic is not applicable to a dataset, the return value is “not available”.

3.1.1 Aggregation of Characteristics

Some of the later defined characteristics can only be calculated for single attributes, but the result of a characteristic must have a fixed number of values for all datasets. Hence, we have to aggregate these values. One possibility to aggregate is to calculate the mean, thereby we often lose a lot of information. To overcome this problem, Kalousis uses a so called histogram representation for characteristics whose return values are in a fixed range. The fixed range is split into n bins and each bin contains the number of the attributes with values in this bin (or the percentage amount).

Using this kind of representation, we do not lose as much information as if using the mean. The next example shows the advantages of this method.

Example (histogram representation) This example shows the difference between the aggregation with mean and with the histogram method by means of the correlation coefficient ρ which ranges between -1 and 1. The theoretical explanation of this measure follows in the next chapter.

Given are two datasets `red`, with three attributes, and `green` with two attributes. Figure 3.1 shows the coherence of the attributes for each dataset.

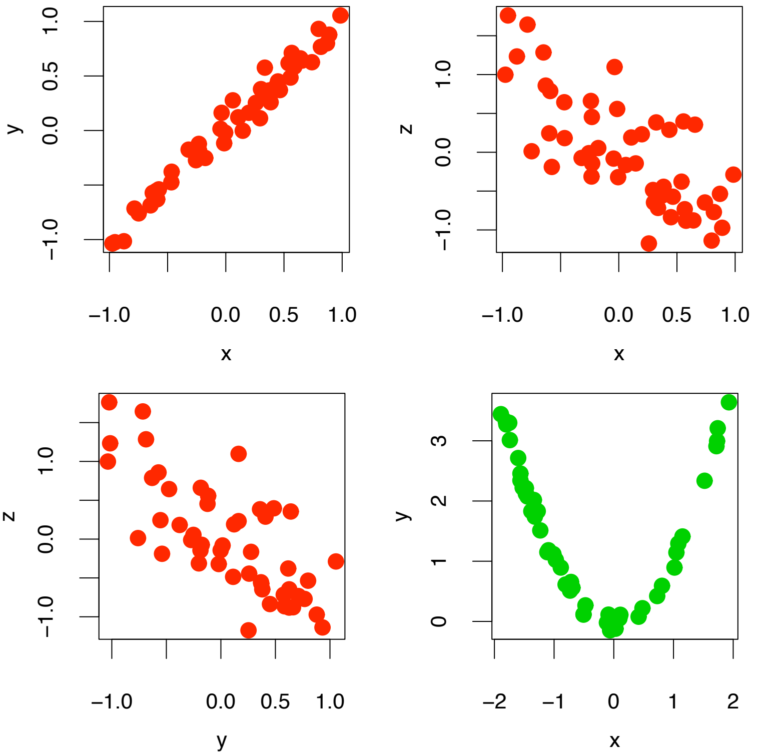

Figure 3.1: Scatterplots to show the coherences of datasets `red` and `green` attributes.

The correlation coefficients of the `red` dataset are

$$\rho_{xy} = 0.986,\ \rho_{xz} = -0.748,\ \rho_{yz} = -0.751$$

and of the `green` dataset

$$\rho_{xy} = -0.183.$$

If we now calculate the mean $\hat{\rho}$ of both datasets, we see that they have similar mean correlation coefficients:

$$\hat{\rho}_{red} = -0.171 \text{ and } \hat{\rho}_{green} = -0.183$$

The difference between the two numbers is not high, so based on the correlation coefficients, one could say that the two datasets are similar.

Using the histogram representation with $n = 3$ bins the aggregated ρ values of the dataset looks as follows. For the `red` dataset

$$\bar{\rho}_{1,red} = 0.667,\ \bar{\rho}_{2,red} = 0.000,\ \bar{\rho}_{3,red} = 0.333$$

and for the `green` dataset

$$\bar{\rho}_{1,green} = 0.000,\ \bar{\rho}_{2,green} = 1.000,\ \bar{\rho}_{3,green} = 0.000.$$

Figure 3.2 is the visualisation of them.

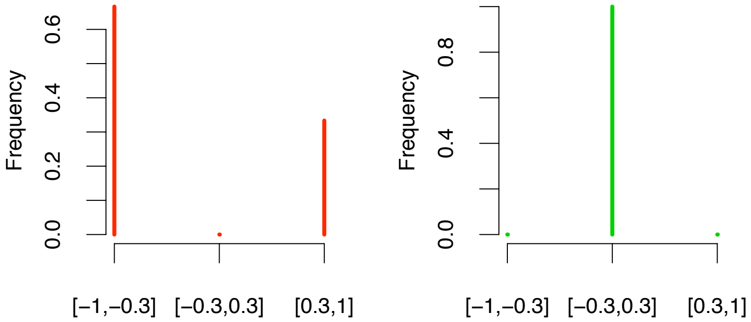

Figure 3.2: Histogram representation of the correlation coefficients of dataset `red` and `green`.

Based on this knowledge we would say that the datasets are totally different. Of course, if we choose a higher value for n, we get a better view of the distribution of the correlation coefficients of a dataset, up to the exact values for $n = \infty$.

3.2 Problem Characteristics

3.2.1 General Description

The most general characteristics are the number of attributes $attr$, the number of samples n and the number of class labels cl.

One important factor in learning problems is the dimensionality. If the number of attributes increases, the number of samples must increase exponentially. The problem is known as the *curse of dimensionality* and is described in detail in Hastie et al. (2001). As a measure of the dimensionality we define

$$dim = \frac{attr}{n}.$$

Another measure of the dimensionality could be $\frac{attr}{\log n}$. The idea is based on the Bayesian information criterion (BIC) (see Hastie et al., 2001, section 7.7), where the number of samples contributes in a logarithmic way.

The performance of a learning method is influenced by incomplete samples (or cases). As different learning methods handle missing values differently, we also measure this characteristic. The characteristics are the total number of missing values $mvals$ and the percentage

$$pmvals = \frac{mvals}{attr * n}.$$

Additionally, the distribution of the missing values among the attributes is important for some learning methods (see Kalousis and Hilario, 2000). Hence, Kalousis proposes to calculate the percentage of missing values for each attribute and then create a histogram with 10 bins. The first bin contains the number (or frequency) of attributes with less than 10%, the next with 10%...20%, and so on, missing values.

3.2.2 Description of the Attributes

A dataset consists of nominal and continuous attributes. Again, different learning methods can handle these types of attributes differently well. To consider this, we measure the number of nominal (nom) and continuous (con) attributes and their percentage from the total number of attributes

$$\%nom = \frac{nom}{attr} \qquad \text{respectivly} \qquad \%con = \frac{con}{attr}.$$

Nominal Attributes

bin is the number of binary-coded attributes when we represent all nominal attributes with “local binary encoding”. Many machine learning methods handle nominal attributes in this way. The problem thereby is, that the dimension increases but the number of samples is the same.

Some general characteristics are calculated from the integer representation of the nominal attributes. There are the maximum $max.nom$, the minimum $min.nom$, the mean value $mean.nom$ and the standard deviation $std.nom$.

The entropy $H(X)$ of a nominal attribute is a measure for the amount of randomness of the categories . It is defined by

$$H(X) = -\sum_i \pi_{i+} \log_2 \pi_{i+}.$$

The value of the entropy is always greater equal than 0. It is maximal if all categories are equally likely, and zero if the categories are certainty. The maximal value depends on the number of categories. The interpretation of the entropy is, that the higher the value, the more uniform is the distribution of the categories. Because of the possible different ranges no histogram representation is possible, hence we use the mean entropy:

$$\widehat{H(X)} = \frac{1}{nom} * \sum_{i=1}^{nom} H(X_i)$$

Example (Entropy) Given is the `titanic` dataset, it consists of three nominal attributes. The following figure shows the distribution of categories per attribute:

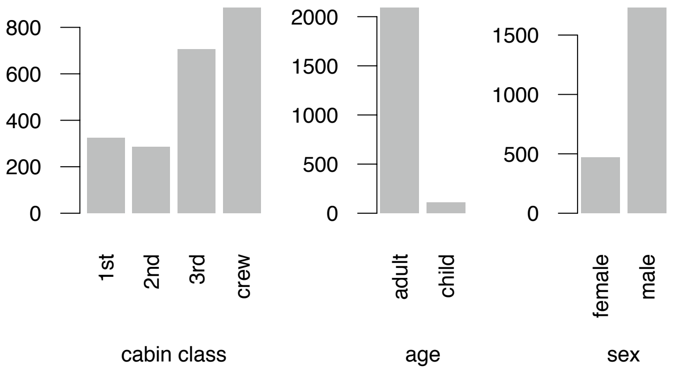

Figure 3.3: Nominal attributes of the `titanic` dataset.

The entropy of the single attributes are

$$H(\texttt{socialclass}) = 1.844,\ H(\texttt{age}) = 0.284,\ H(\texttt{sex}) = 0.748$$

and the characterisation of the dataset through the mean entropy is

$$\widehat{H} = 0.959.$$

Continuous Attributes

To describe continuous attributes, we use skewness and kurtosis from the descriptive statistics. Both measures describe deviances from the normal distribution. We characterise a dataset with these measures because some learning methods are based upon the assumption of normally distributed samples per class. So, based on the response classes, the attributes are split and the kurtosis and skewness are calculated.

The skewness describes the missing symmetry of a distribution. A positive skewness means that there is an asymmetry tail towards the positive numbers and if the value is negative, the asymmetry is towards the negative numbers. It is defined as

$$\gamma = \frac{E(X - \mu_X)^3}{\sigma_X^3}.$$

The kurtosis describes how “fat” the tails are in comparison with a normal distribution with the same standard deviation. It is defined as

$$\beta = \frac{E(X - \mu_X)^4}{\sigma_X^4}.$$

As both measures are unbounded, we use the mean values $\hat{\gamma}$ and $\hat{\beta}$ for the characterisation.

Example (Skewness and Kurtosis) Given is an artificially created attribute `attr` with values from two classes `red` and `green`. The values of the class `red` are normal distributed with $N(0,1)$ and the values of the `green` class are drawn from a exponential distribution with $\lambda = 1$. Figure 3.4 shows a jitter plot.

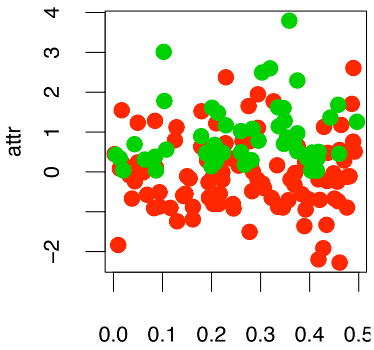

Figure 3.4: Jitter plot of the attribute `attr` with the separation into the two classes `red` and `green`.

The skewness and kurtosis per classes are:

$$\gamma(\texttt{red}) = 0.383,\ \beta(\texttt{red}) = 3.278$$

$$\gamma(\texttt{green}) = 1.443,\ \beta(\texttt{green}) = 4.956$$

The resulting mean values are

$$\hat{\gamma}(\texttt{attr}) = 0.913 \text{ and } \hat{\beta}(\texttt{attr}) = 4.117.$$

If one uses learning methods with discriminant functions for classification, the equality of the covariance matrices of the different class samples play an important role. A measure, which can be used to test equality is the Box's M statistic . M is zero if all covariance matrices S_{c_i} are equal to the pooled covariance matrix $S = \frac{1}{n-cl}\sum_{i=1}^{cl} S_{c_i}$. We use this statistic M to calculate the quotient $SD.ratio$ of the pooled standard deviation and the standard deviations of the classes:

$$SD.ratio = \exp\left(\frac{M}{con * \sum_{i=1}^{cl}(n_i - 1)}\right)$$

$SD.ratio$ is 1 if $M = 0$, otherwise it is strict greater than 1.

Example ($SD.ratio$) Given are the two datasets `a` and `b` from figure 3.5 with each two classes. The samples from both datasets are drawn from a multivariate normal distribution. In case of dataset `a`, the covariance matrices of both classes are equal:

$$\Sigma_1 = \Sigma_2 = \begin{pmatrix} 1 & 0 \\ cr0 & 1 \end{pmatrix}$$

For dataset `b` the covariance matrices are different:

$$\Sigma_1 = \left(1 \quad 0\}0 \quad 1\right), \Sigma_2 = \begin{pmatrix} 10 & 3 \\ 3 & 2 \end{pmatrix}$$

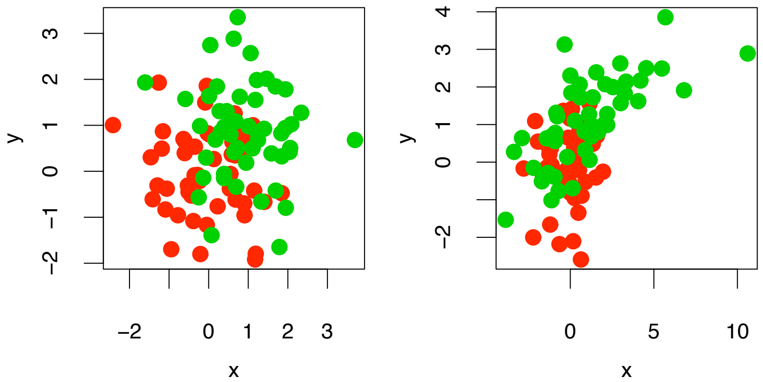

Figure 3.5: Datasets with same respective different covariance matrices.

The values of the Box's M statistic are

$$M_a = 0.764 \text{ and } M_b = 39.040,$$

hence the corresponding quotients are

$$SD.ratio_a = 1.004 \text{ and } SD.ratio_b = 1.220.$$

Class Attribute

An interesting characteristic of the class attribute is the distribution of the samples in respect to the categories. One possibility to measure this, is the entropy $H(C)$. The theoretical explanation is the same as above, and the interpretation is: the higher the value, the more uniform is the distribution of the samples.

3.2.3 Attribute Associations

To measure associations between two nominal attributes X, Y with I respectively J different categories, one can use the concentration coefficient (or Goodman and Kruskal's τ)

$$\tau_{XY} = \frac{\sum_{i=1}^{I} \sum_{j=1}^{J} \frac{\pi_{ij}^2}{\pi_{i+}} - \sum_{j=1}^{J} \pi_{+j}^2}{1 - \sum_{j=1}^{J} \pi_{+j}^2}.$$

This coefficient describes "the proportional reduction in the probability of an incorrect guess predicting Y using X" (Kalousis, 2002). It ranges between $[0, 1]$ and the higher the value the better we can predict Y, if we know X. Therefore X is assumed as independent and Y as dependent. Hence the concentration coefficient is not symmetric and we have to calculate it for all possible combinations of the nominal attributes of a dataset. The results are reported in a histogram.

Another, more popular, measure is the chi-squared statistic (for the Definition see Venables and Ripley, 2002, page 64) from the chi-squared test of independence. But as Kalousis does not use it in his dataset characterisation, I won't use it too.

Example (concentration coefficient) Given are data from a survey. The interviewer asked 2619 persons about their favorite fruit (`apple`, `banana`, `cherry` or `orange`) and their favorite color (`blue`, `green` or `red`). The following table lists the absolute frequency of the survey:

Fruit	Color		
	blue	green	red
apple	42	694	748
banana	66	474	87
cherry	123	266	21
orange	21	67	10

Table 3.1: Contingency table of the survey.

Now, is there an effect in predicting the favorite color (Y) if we know the favorite fruit (X)? And in the other direction?

In the first case, the concentration coefficient τ_{XY} is 0.118. This means that the percentage rate of a wrong favorite color prediction is reduced by 11.818%, if we additionally know the distribution of the favorite fruits. In the other direction, the coefficient τ_{YX} is 0.130.

Associations between two continuous attributes X and Y are measured with the correlation coefficient ρ_{XY} . It is a measure for the linear dependence of the two attributes and defined through:

$$\rho_{XY} = \frac{Cov(X,Y)}{\sqrt{Var(X)Var(Y)}}$$

The correlation coefficient ranges between $[-1, 1]$ and is symmetric. 1 indicates a perfectly positive, and -1 a perfectly negative linear dependence. 0 means that there is no linear dependence. The characterisation is done with a histogram.

Example (correlation coefficient) The following figures show two dependent variables x and y and their correlation coefficient.

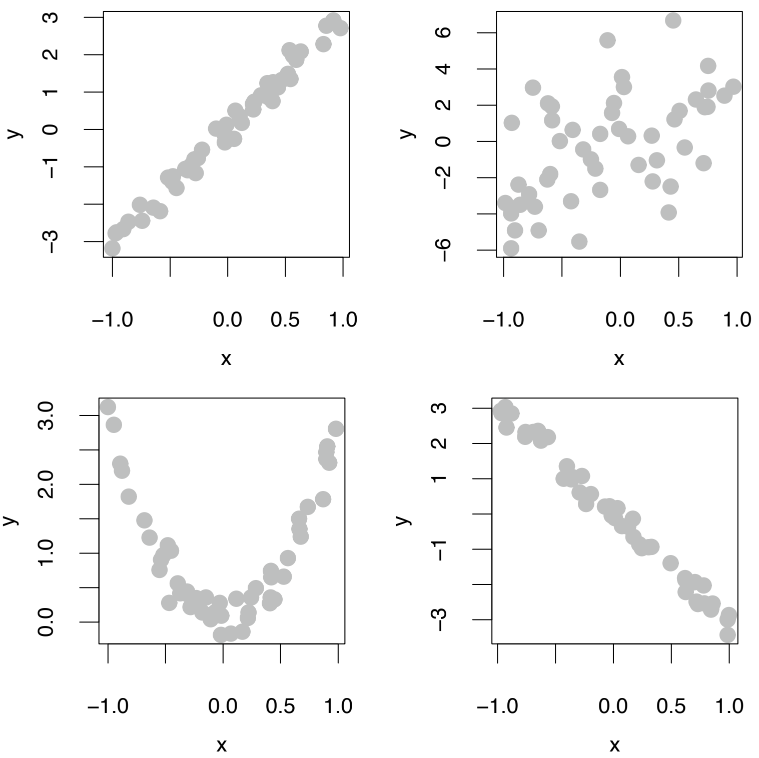

Figure 3.6: The correlation coefficients are $\rho = 0.992$, $\rho = 0.513$, $\rho = 0.0519$ and $\rho = -0.994$.

Another description of associations between continuous attributes is the multiple correlation coefficient . This coefficient describes the correlation between each attribute and the linear combination of the remaining ones. Are $X_1, X_2, \ldots, X_{con}$ continuous attributes, then the multiple correlation coefficient R_i between X_i and the multivariate variable $Z_i = (X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_{con})$ is given through the maximal correlation coefficient between X_i and a linear function (from Z_i) $Z_i * \alpha$. Hence, R_i is given through:

$$R_i = argmax_{\alpha \neq 0} \frac{Cov(X_i, Z_i\alpha)}{\sqrt{Var(X_i)Var(Z_i\alpha)}}$$

R_i ranges between $[0, 1]$. 1 means, that X_i is a linear combination of Z_i and 0 means, that both variables are independent. So, this coefficient answers the question, if the prediction of an attribute X_i improves, when X_i does not only depend on one other attribute X_j but is seen as dependent attribute of $X_{j_1}, \ldots X_{j_k}$.

What we need yet, is a possibility to describe associations between continuous and nominal attributes. For this case, Kalousis uses the F-distribution, similar to the ANOVA. He uses the *p-value* as indicator, whether a continuous attribute affects a nominal attribute. A *p-value* near 0 rejects the assumption of the equality of the group mean values, a value near 1 accepts this assumption. Since the *p-value* is a probability it ranges between $[0, 1]$.

The problem with this proposed usage of the *p-values* is, that they are equally distributed in the interval $[0, 1]$ under the null hypothesis, i.e. we can not really talk about good or bad values, but the distance function uses them in a linear way. A better way is to report the rejection of the null hypothesis for different significance levels and use that for the distance calculation between two datasets.

Example (*p-value*) The following figure 3.7 shows a box plot of two datasets `a` and `b`. Both have one attribute and three classes.

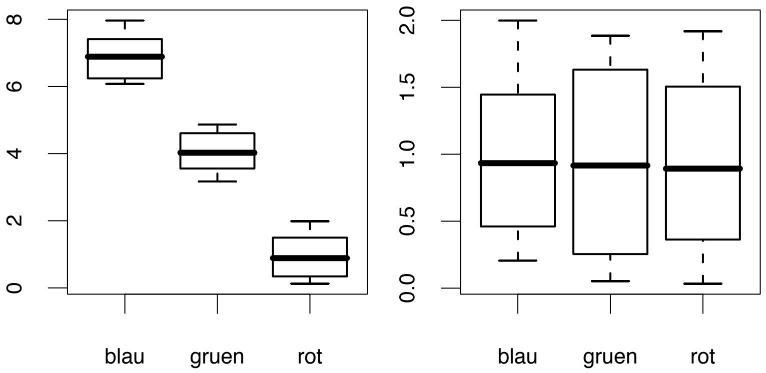

Figure 3.7: Boxplots of the classes for each dataset `a` and `b`. Dataset `a` is produced with different class means, `b` with equal class means.

The *p-value* for dataset `a` is 4.75e-36 (that is numerically 0) and the *p-value* for dataset `b` is 0.989.

3.2.4 Associations with the Class Attribute

Probably the most important aspect in classification is the amount of information an attribute provides about the class. It seems reasonable, that the higher an attribute's information content about the class is, the easier is the classification. Different learning methods are differently robust against irrelevant attributes.

A frequently used measure for the mutual dependency of two attributes X and Y, is the transinformation or mutual information . It measures how much knowing one of these attributes reduces the uncertainty about the other. It is defined through

$$MI(X, Y) = H(Y) - H(Y|X).$$

$H(Y|X)$ is the conditional entropy of Y given X. The mutual information is symmetric and ranges between $[0, min(H(X), H(Y))]$. The value 0 indicates, that the attributes are independent. If one attribute totally explains the other then the maximum is taken. This measure is also known as “information gain” and plays an important role on decision trees.

To characterise a dataset, we calculate the mutual information for each attribute X_i together with the class attribute C and average out:

$$\widehat{MI}(C, X) = \frac{1}{nom} * \sum_{i=1}^{nom} MI(C, X_i)$$

We can further use this value to give an estimation about the common number of needed attributes to describe the class attribute. This is known as equivalent number of attributes:

$$EN.attr = \frac{H(C)}{\widehat{MI}(C, X)}$$

And we can estimate the amount of non-useful information of a dataset, the noise-to-signal ratio

$$NS.ratio = \frac{\widehat{H}(X) - \widehat{MI}(C, X)}{\widehat{MI}(C, X)}.$$

Example (mutual information) Figure 3.8 completes the dataset `titanic` (see figure 3.3) with the corresponding class attribute.

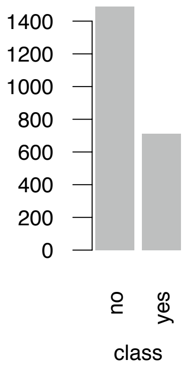

Figure 3.8: Class attributes of the `titanic` dataset.

The mutual information between this class attribute and each nominal attribute is

$$MI(C, \texttt{socialclass}) = 0.0593,\ MI(C, \texttt{age}) = 0.00641 \text{ and } MI(C, \texttt{sex}) = 0.142,$$

with the corresponding entropies

$$H(\texttt{socialclass}) = 1.844,\ H(\texttt{age}) = 0.284,\ H(\texttt{sex}) = 0.748 \text{ and } H(C) = 0.908.$$

The resulting mean transformation is

$$\widehat{MI}(C, X) = 0.0694,$$

and furthermore:

$$EN.attr = 13.085 \text{ and } NS.ratio = 12.824$$

As another characterisation, the concentration coefficient τ_{X_iC} between the class attribute and each nominal attribute is used. This value ranges between $[0, 1]$, hence we can use the histogram representation.

Let us turn to continuous attributes. Again, an indication of associations between continuous attributes and the class attribute is described with the *p-value* of the F-distribution. We determine for each continuous attribute X_i, whether the class C affects a grouping of the attribute or not. Is this the case, the attribute is good to describe differences between categories. As mentioned above, the *p-value* is a probability and ranges between $[0, 1]$. So, we can us the histogram representation for the characterisation.

To get a real description of associations of continuous attributes, we have to use the theory of discriminant analysis and especially the canonical correlation analysis . The canonical correlation analysis tries to find and quantify associations between groups of attributes. Assume, A and B are distinct sets of attributes, B is seen as the set of dependent ones. Now, the analysis compares the correlation between linear combinations of both sets.

The calculation is based on the calculation of the eigenvalue λ_i for each linear discriminant function. A high eigenvalue signifies, that the corresponding discriminant function explains a lot of the variance. We can use the eigenvalue as quality measure, but it is not bounded to range, so, for this reason, we use the canonical correlation coefficient:

$$\rho_i = \sqrt{\frac{\lambda_i}{1 + \lambda_i}}$$

Since the eigenvalues are non-negative and sorted ($\lambda_1 \leq \ldots \leq \lambda_{attr-1}$), it is imperative that $0 \leq \rho_i \leq 1$ and the ρ_i are sorted too.

For the characterisation of a dataset, Kalousis only uses the first canoncial correlation coefficient between all continuous attributes (the set A) and the local-binary encoded class attribute (the set B):

$$\rho_{max} = \sqrt{\frac{\lambda_1}{1 + \lambda_1}}$$

Back to the linear discriminant functions. The total variation of them is the sum of the eigenvalues. The proportion of variation explained by the first linear discriminant, and hence the fraction of the first canonical correlation coefficient, is computed as

$$frac1 = \frac{\lambda_1}{\sum_i \lambda_i}.$$

Example (first canonical correlation coefficient) Given are the following two two-dimensional datasets from figure 3.9:

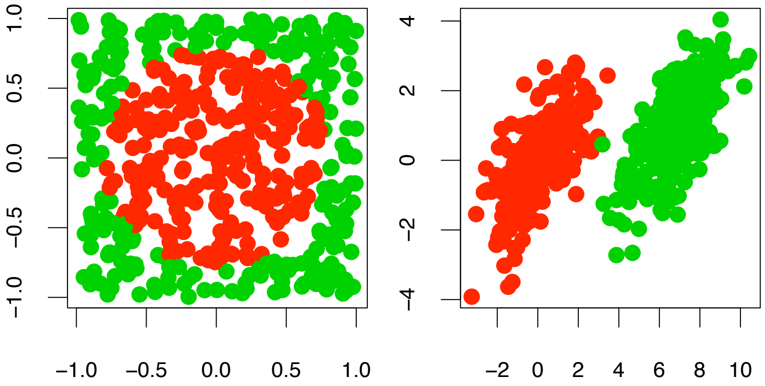

Figure 3.9: Datasets a and b, whereby b is and a is not linear separable.

Both datasets only have one linear discriminant function, the following figure shows the transformed attributes with a jitter plot:

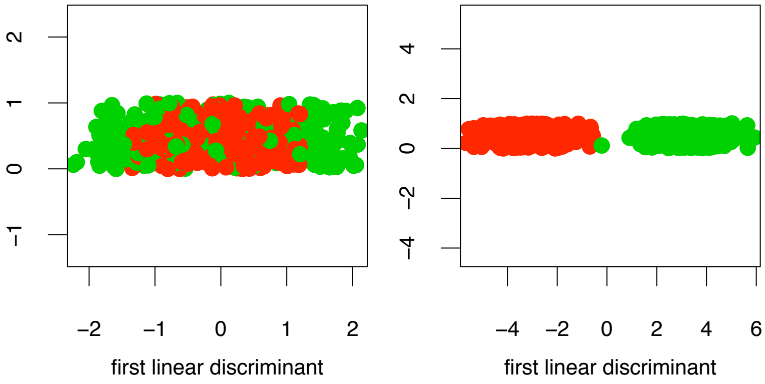

Figure 3.10: Jitter plots of first linear discriminants of datasets a and b.

As we can see, the first linear discriminant of dataset a is not linear separable, hence the eigenvalue and the first linear correlation coefficient are near 0:

$$\lambda_a = 0.00196 \text{ and } \rho_{max_a} = 0.0442$$

On the other hand, the first linear discriminant of dataset `b` is linear separable, the eigenvalue is high and the first linear correlation coefficient is near 1:

$$\lambda_b = 10.382 \text{ and } \rho_{max_b} = 0.955$$

3.3 Distance Measure

With the above defined measures we can characterise all datasets and get a “normalized” form of them. Now, to put them in relation, we have to define similarity. Kalousis (2002) and Soares and Brazdil (2000) define distance measures in their papers. The idea behind both definitions is the same, they just normalize the characterisation values differently. I explain and use the one from Kalousis.

As mentioned in the introduction of this chapter, if a characteristic is not applicable to a dataset, the return value is “not available” (`NA`). But in this special case, this does not mean that there is no information available, in comparison with other datasets, this `NA` is really important.

Considering, there are two datasets D and D'. If we characterise both datasets and one specific character measure m_i is not applicable on both, then the two are equal at this measure m_i. Hence the distance between them should be zero. On the other hand, if m_i is applicable on dataset D and not on dataset D' then the distance should be as large as possible.

With this consideration, we define the distance measure d: D and D' are datasets, $\langle m_1, \ldots, m_c \rangle$ and $\langle m'_1, \ldots, m'_c \rangle$ the corresponding sorted and normalized to $[0, 1]$ characterisations. The distance measure d is defined as:

$$d = \sum_{i=1}^{c} d_{ii}^2$$

with

$$d_{ii} = \begin{cases} m_i - m'_i, & \text{if } m_i,\, m'_i \in \mathbf{R} \\ 1, & \text{if one of } m_i \text{ or } m'_i \text{ is } \texttt{NA} \\ 0, & \text{if both } m_i \text{ and } m'_i \text{ are } \texttt{NA} \end{cases}$$

3.4 The latem Package, Part 1

This first part of the latem package description shows the implementation and usage of the theoreticaly defined characteristics and distance measures.

3.4.1 Characterisation

A character measure is implemented with the CharacteristicFunction class. The function has one argument of type IDataSet and the return value is a numeric vector. One must specify the length of the return vector, i.e. one value or the number of histogram bins.

A CharacteristicList contains the wanted characteristics. The characterisation of a datasets is done with the characterise(...) method. This method takes a list of DataSet objects and returns a Characterisation object. This object extends the matrix class and provides some methods like add(...), which adds a new dataset, and distance(...), which calculates the distance matrix with respect to a distance function.

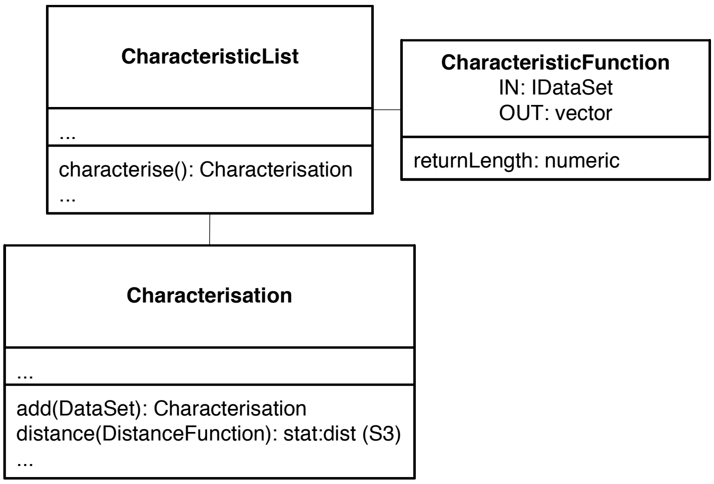

Figure 3.11: A CharacteristicList contains various CharacteristicFunctions. The result of a characterisation is a Characterisation object.

A distance measure is implemented with the class DistanceFunction. The input arguments are two numeric vectors of the same length. Normally they are the result vectors of the characterise(...) method. The DistanceFunction return value is a dist class from the package stat (this is a S3 package).

Additional, the distance measure needs normalized values, hence we need the possibility to process the Characterisation object. This offers the CharacterisationProcessingFunction class. The input argument and the return value are Characterisation objects.

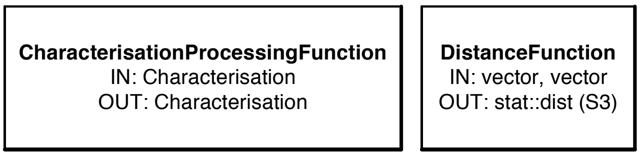

Figure 3.12: The CharacterisationProcessingFunction and DistanceFunction classes.

3.4.2 Usage Example

```
> library(latem)
```

I use two datasets from the case study to show some aspects of the dataset characterisation. Heart1 is a real-world problem and Spirals a artificially created one:

```
> Heart1

Heart1: DataSet
        Class ~ age + sex + pain + trestbps + cholesteral + sugar + ecg +
    rate + angina + oldpeak + slope + vessels + thal
        with 303 samples,
        and the aspects input, response.
```

```
> Spirals

Spirals: DataSet
        Class ~ X1 + X2
        with 1200 samples,
        and the aspects input, response.
```

The characters are the one from Kalousis, there is a function which returns the corresponding list with these characteristics:

```
> chars = kalousisList()

Kalousis character list: CharacteristicList
        with 32 characters and 95 values.
```

```
> names(chars)

 [1] "n"        "attr"     "cl"       "dim"      "nas"      "pnas"
 [7] "nom"      "max.nom"  "min.nom"  "mean.nom" "sd.nom"   "con"
[13] "pcon"     "pnom"     "bin"      "skew"     "kurt"     "hc"
[19] "hx"       "mi"       "en.attr"  "ns.ratio" "tauxy"    "tauxc"
[25] "rhoxy"    "nasx"     "r"        "pvalsxy"  "pvalsxc"  "rho.max"
[31] "frac1"    "sd.ratio"
```

As an example of an implementation of such a character, I show the "number of samples" character:

```
> n.char = characteristicFunction("n", 1, function(dataSet) {
+     n = nrow(dataSet@get("input"))
+     names(n) = "n"
+     return(n)
+ })
```

Characterise both datasets, the return value is a `Characterisation` object:

```
> c = characterise(chars, list(Heart1, Spirals), "usex")

usex: Characterisation
         2  datasets characterised
        with the Kalousis character list characteristics.
```

To calculate the distance matrix between the datasets, we have to say, how to calculate the distance between two of them:

```
> kalousis.dist = distanceFunction("kalousis distance", function(x,
+     y) {
+     nas = union(which(is.na(x)), which(is.na(y)))
+     if (length(nas) != 0) {
+         x = x[-nas]
+         y = y[-nas]
+     }
+     d = sum((x - y)^2)
+     n = union(setdiff(xnas, ynas), setdiff(ynas, xnas))
+     return(d + length(n))
+ })
```

This distance measure needs a normalized form (to range $[0, 1]$) of the characterisation values, hence we have to implement such a preprocessing function:

```
> normalize.cproc = characterisationProcessingFunction("normalize [0,1]",
+     function(characterisation) {
+         r = range(characterisation, na.rm = T)
+         return((characterisation - r[1])/sum(abs(r)))
+     })
```

Now we can calculate the distance:

```
> distance(normalize.cproc(c), kalousis.dist)

[1] 39.55892
```

Let us change the Heart1 dataset and watch the distance. First modification is the simulation of a measurement error in the attribute `cholesteral`:

```
> h2 = dataset(Heart1)
> h2$cholesteral = 0.1 * h2$cholesteral + 17
> c = add(c, dataSet(h2, Class ~ ., "Heart1-2"))

> distance(normalize.cproc(c), kalousis.dist)

                Heart1      Spirals
Spirals  3.955892e+01
Heart1-2 4.159648e-08 3.955892e+01
```

The distance grows just a little bit. Second modification is deletion of the attributes `age` and `sex`:

```
> h3 = h2[, -(1:2)]
> c = add(c, dataSet(h3, Class ~ ., "Heart1-3"))

> distance(normalize.cproc(c), kalousis.dist)

                Heart1      Spirals     Heart1-2
Spirals  3.955892e+01
Heart1-2 4.159648e-08 3.955892e+01
Heart1-3 7.527692e-06 3.955888e+01 7.487502e-06
```

Now the modified dataset has moved more away from the original one. The last modification is the generation of incomplete cases, therefor I insert `NA` values at various positions:

```
> h4 = h3
> h4[sample(1:303, 10), sample(1:12, 10)] = NA
> c = add(c, dataSet(h4, Class ~ ., "Heart1-4"))
```

```
> distance(normalize.cproc(c), kalousis.dist)
```

```
               Heart1      Spirals     Heart1-2     Heart1-3
Spirals  3.955892e+01
Heart1-2 4.159648e-08 3.955892e+01
Heart1-3 7.527692e-06 3.955888e+01 7.487502e-06
Heart1-4 5.640068e-03 3.956538e+01 5.640028e-03 5.630328e-03
```

Case Study: The Meta-Knowledge Base

The idea of this case study is, that maybe I could find clusters in the characterisation of the datasets and these clusters would be similar to the indicated clusters in the pooled benchplot from figure 2.7.

The characterisation is done with all characteristics described in the section 3.2 above . I call this the Kalousis' list, because these are the ones he defined in his paper (Kalousis, 2002). Other characterisation lists are the METAL and StatLog lists, both are also defined in the last named paper. They are subsets of the Kalousis list and without the histogram representation of some results. I do not use them for this case study. The distance function is the one I defined in the section 3.3.

I take a look at the hierarchical cluster dendrogram with complete linkage (figure 3.13). In association with multidimensional scaling (see Venables and Ripley, 2002, pages 302-306) of the distance matrix to the two-dimensinal space, I decide to use 7 different clusters.

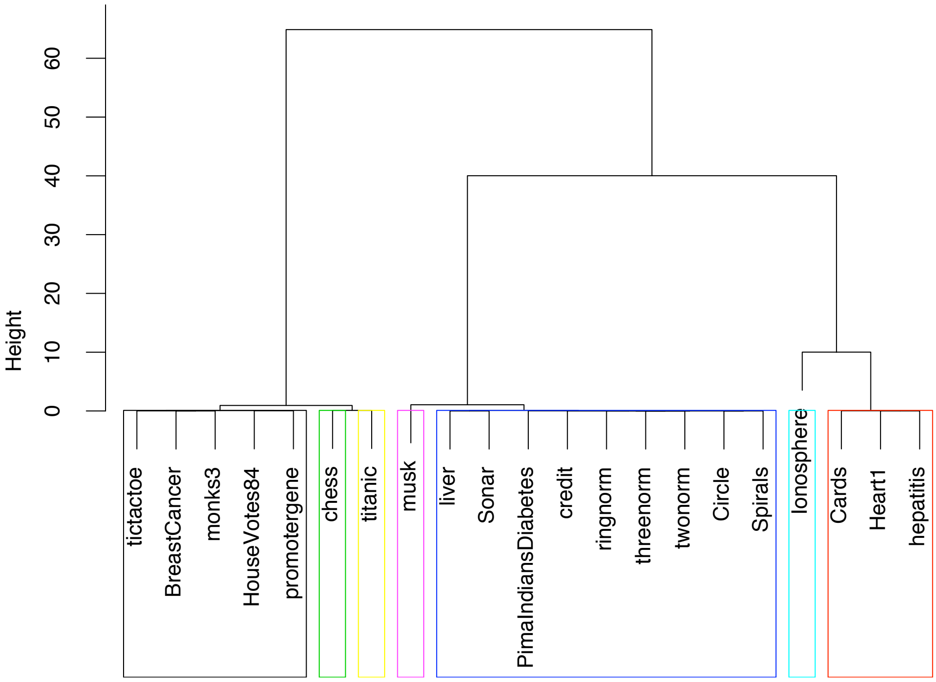

Figure 3.13: Hierarchical cluster dendrogram of the dataset characterisations. The coloured borders indicate the different clusters.

The visualisation of the multidimensional scaling should explain why I use this clustering: As we can see in figure 3.14 three big clusters exist, but if we take a closer look we see that in each of these clusters most datasets are exactly on the same position and only one or two are not. With regard to the relation with the benchmark results from the last case study part, I want to be as sensitive as possible concerning dissimilarities between the characterisations and take a look at their impact on the methods performances. Thus I treat them at the moment as own clusters and compare them with their corresponding big cluster.

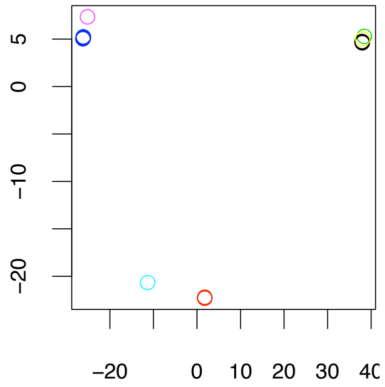

Figure 3.14: Plot of the distance matrix of the characterisations, scaled into the two-dimensional space. The colours indicate the same clusters as in the figure above.

As we can see, there are three clearly separated clusters, but each of these three clusters has some one-member clusters. Now I want to bring in the benchmark experiment results. My idea is to split the pooled benchplot from 2.7 into pieces, whereas each plot contains the datasets corresponding to one cluster. The following figures show the clusters with more than one member together with their nearest one-member clusters.

The top-right cluster: The support vector machines (blue) dominate this cluster, four times in first and two times in second place.

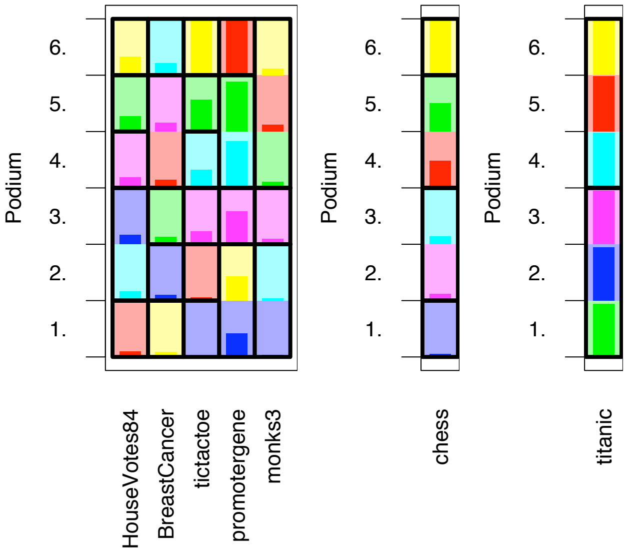

Figure 3.15: Rankings of the top-right cluster: the big black cluster with the green and yellow one-member clusters.

The top-left cluster: With regard to the benchmark results, this cluster splits into two parts. Again, support vector machines (blue) dominate one part, but k-nearest neighbours classification (green) is also not so bad. In the second part, linear discriminant analysis (red) is in front.

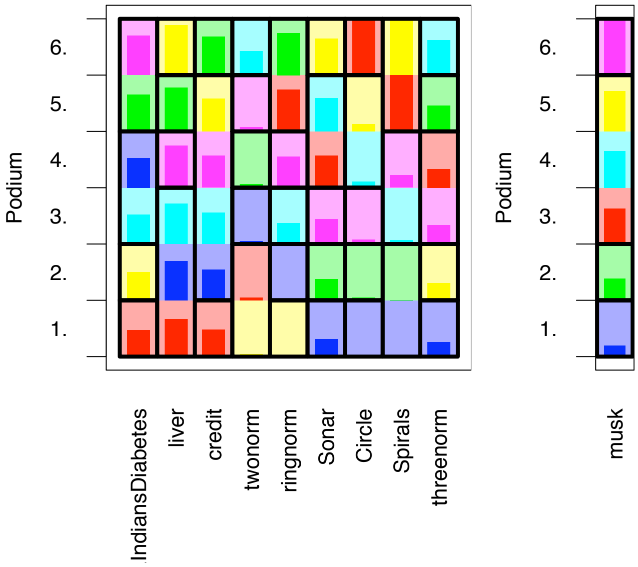

Figure 3.16: Rankings of the top-left cluster: the big blue cluster with the yellow one-member cluster.

The bottom cluster: In this cluster, three algorithms are equally often in front, linear discriminant analysis (red), support vector machines (blue) and naive bayes (yellow).

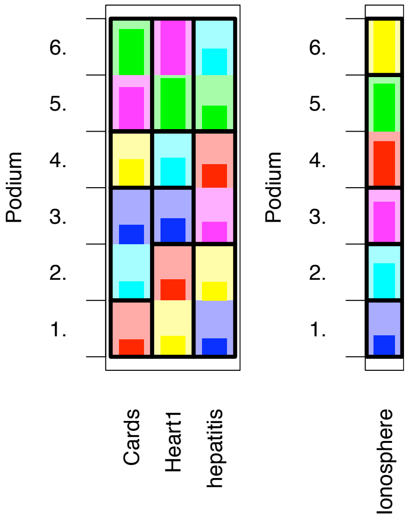

Figure 3.17: Rankings of the bottom cluster: the big red cluster with the cyan one-member cluster.

All in all, the result is not as clear as I thought, but I think some trends are visible. One cause may be the small number of datasets. Kalousis, for example, used 47 initial datasets and then generated 1035 new datasets by deleting a given percentage of attribute values and creating new attributes whose values were generated in a purely random fashion.

Chapter 4

Meta-Learner

In this chapter, I explain the last part of the framework, the meta-learner. There are different approaches, I introduce all ideas and explain their differences, then I point out two meta-learner in detail and outline their implementation in the corresponding second part of the `latem` package. The case study shows their practical usage.

4.1 Introduction

By now we have the possibilities to benchmark a set of candidate algorithms on a set of datasets, characterise these datasets and relate them using a distance measure. As shown in the last case study, if we associate both data, some trends come to the fore concerning the clustering of the datasets and the ranking of the candidate algorithms. Now we want to use this knowledge and apply it to new problems . There exist three different kinds of methods (Kalousis, 2002). The two main approaches see meta-learning as classification respective as regression problem, the third approach includes methods which produce rankings but do not fit into the classification or regression attempt.

In the following the meta-knowledge base $kb = (C_{D_j}, ER_{l_i}^{D_j}, T_{l_i}^{D_j})$ is given. It contains information about M datasets ($j = 1, \ldots, M$) and N candidate algorithms ($i = 1, \ldots, N$). C_{D_j} is the characterisation $\langle m_1, \ldots, m_c \rangle$ of the dataset D_j, $ER_{l_i}^{D_j}$ and $T_{l_i}^{D_j}$ are the error rate and the benchmark time of algorithm l_i on dataset D_j.

4.1.1 Classification-Based

In this approach meta-learning problems are formulated as classification problems. The instances of these classification problems consist of the characterisation of the datasets and the class label. Depending on the class label, different suggestions are possible.

The class label can take, for example, one of the values *appl* and *non-appl*, depending whether the algorithm exhibits high or low performance on the specific dataset (Michie et al., 1994). In this case the meta-learner constructs N classification problems, one for each candidate algorithm. The suggestion is a prediction of possible algorithms.

Another possibility is to create pairwise comparisons of the algorithms for each dataset, the class label indicates the winner of the pair (Pfahringer et al., 2000). This results in $\binom{N}{2}$ different classification problems, and the suggestion is a partial ordering.

The simplest formulation of a meta-learner is to construct one classification problem with a class label which indicates the best algorithm for the corresponding problem (Bensusan and Giraud-Carrier, 2000). The suggestion contains the best algorithm.

4.1.2 Regression-Based

In this approach, the goal is to predict some performance measure of an algorithm from the characterisation of the dataset. A regression problem is formulated for each algorithm, hence the meta-learner constructs N regression problems. To solve the regression problems, one can use every existing algorithm. To make better use of the semantics of the "not-available" values that occur in the datasets characterisation, one has to change the distance measure or to recode the values to new ones that lie outside the domain (Bensusan and Kalousis, 2001). The suggestion consists of the prediction of the performance measure for each algorithm. Based on this, a ranking can be established or the best algorithm determined. The visualisation of this idea is, in principle, figure 1.5 about the "no free-lunch theorem"

4.1.3 Other Local Methods

There are some methods which cannot be classified in the above approaches. These methods use a combination of the nearest neighbour method and some ranking schemas. With the first part a set of similar datasets (based on the characterisation) is established, then the performance information of these datasets are used to construct a relative ranking of the algorithms.

4.2 Local Methods

The assumption that I take is that no global model exists for that kind of problem, Thus I only consider local methods. Both methods follow a similar idea, but the the first one, the zoomed ranking meta-learner, generates a relative ranking and the second one, the Nadaraya-Watson-Epanechnikov meta-learner, estimates a classifier's performance measure.

4.2.1 Zoomed Ranking

The idea behind this method, introduced by Soares and Brazdil (2000), is that algorithms perform similar on similar datasets. The method consists of two distinct phases. The first phase is called zooming and identifies a subset of similar datasets. This subset is used to construct a relative ranking of the algorithms on the basis of the performance information in the second phase.

Zooming

The selection of relevant datasets is based upon the k-nearest neighbour method (introduced in section 2.1.3) . This method is employed on the characterisation of the datasets, using the distance measure defined in section 3.3 and returns a list of k datasets which are in the neighbourhood of a new problem .

Ranking

Now, we want to use the performance information from the neighbourhood to generate a ranking of the algorithms for a new problem. Several schemata can be used for that purpose (see Soares and Brazdil, 2000; Brazdil and Soares, 2000), the following sections explain some of them.

Average Ranks (AR): This is a simple ranking method which only uses the information about the error rate . For each dataset the algorithms are ordered according to the misclassification error and ranks are assigned. The best algorithm has rank 1, the second has rank 2, and so on. Generally, $R_{l_i}^{D_j}$ is the rank of algorithm l_i on dataset D_j, hence the average rank for each algorithm is

$$AR_{l_i} = \frac{1}{N} \sum_{D_j} R_{l_i}^{D_j}.$$

The final ranking is obtained by sorting AR_{l_i} ascending, whereas the algorithm with the smallest value is the suggestion for the best.

Success Rate Ratios (SRR): Again, this method only uses the error information . First step is to calculate the success rate ratio for each pair of algorithms l_i, $l_{i'}$ on each dataset D_j:

$$SRR_{l_i,l_{i'}}^{D_j} = \frac{1 - ER_{l_i}^{D_j}}{1 - ER_{l_{i'}}^{D_j}}$$

Second step is to aggregate the values and calculate the pairwise mean success rate ratio

$$SRR_{l_i,l_{i'}} = \frac{1}{M} \sum_{D_j} SRR_{l_i,l_{i'}}^{D_j},$$

for each pair of algorithms. This represents an estimate of the general advantage/disadvantage of algorithm l_i over algorithm l_j. Last step is the calculation of the overall mean success rate ratio for each algorithm,

$$SRR_{l_i} = \frac{1}{N-1} \sum_{l_{j'}} SRR_{l_i, l_{i'}}$$

The ranking of the algorithms is derived directly from this measure, the higher the value the higher the corresponding rank.

Adjusted Ratio of Ratios Ranking (ARR): The steps of this method are equal to the steps of the last method, the difference is the calculation of the first measure . This one uses the information about the error and the total execution time. The adjusted ratio of ratios for each pair of algorithms l_i, $l_{i'}$ on each dataset D_j is defined as

$$ARR^{D_j}_{l_i, l_{i'}} = \left(\frac{SR^{D_j}_{l_i}}{SR^{D_j}_{l_{i'}}} \right) * \left(1 + \frac{\log(\frac{T^{D_j}_{l_i}}{T^{D_j}_{l_{i'}}})}{K_T} \right)^{-1} .$$

$SR^{D_k}_{l_i}$ is the success rate and is calculated by $1 - ER^{D_k}_{l_i}$, K_T is a user-defined value that determines the relative importance of time. The ratio of the success rates can be seen as advantage of algorithm l_i relative to algorithm $l_{i'}$, hence it can be considered as benefit. Respectively, the ratio of times can be seen as disadvantage of algorithm l_i relative to $l_{i'}$ and hence as cost.

The logarithm of the time ratio is used, because the range of this ratio is much wider then the range of the success rate ratio and it would dominate the result. Using the logarithm results in values around 1, as happens with the success rate ratio.

The K_T parameter is interpreted as an estimate of how much accuracy (in percent) one is willing to trade for a 10 times speedup or slowdown, i.e. $10x$ speedup/slowdown $\cong$ $X\%$ accuracy. For example, the user is willing to trade 10% of accuracy for a 10 times speedup/slowdown, then $K_T = 1/10\% = 1/0.1 = 10$.

The second step aggregates the values to the pairwise mean adjusted ratio of ratios

$$ARR_{l_i, l_{i'}} = \frac{1}{M} \sum_{D_k} ARR^{D_k}_{l_i, l_{i'}}$$

And the ranking ist obtained by calculating the the overall mean adjusted ratio of ratios

$$ARR_{l_i} = \frac{1}{N-1} \sum_{l_{i'}} ARR_{l_i, l_{i'}}$$

for each algorithm and sort them in ascending way.

4.2.2 Nadaraya-Watson-Epanechnikov Ranking

To gain an estimation of a performance measure of a classifier, we can use a similar idea to the zoomed ranking method. Again we look at the neighbourhood, but instead of using a schema to calculate a ranking score for each method, we perform regression on the data and obtain the ranking according to these values.

I decided to use the Nadaraya-Watson regression method with the Epanechnikov quadratic kernel because this is a simple method and easy to implement.

Nadaraya-Watson Regression

The Nadaraya-Watson estimator is member of a class of nonparametric regression techniques that gain flexibility in estimating the continuous regression function by fitting a different but simple model separately at each query point. This is done by using only those observations close to the query point, which is achieved by using a kernel function to weight the training samples based on its distance from the query point. As the model is the entire training dataset, this method is known as memory-based, other synonyms are kernel-based or instance-based method.

The estimator is defined (following Hastie et al., 2001) as

$$\hat{f}(x_0) = \frac{\sum_{j=1}^{M} K_\lambda(x_0, x_j) * y_j}{\sum_{j=1}^{M} K_\lambda(x_0, x_j)},$$

with the Epanechnikov quadratic kernel

$$K_\lambda(x_0, x) = D\left(\frac{d(x_0, x)}{\lambda}\right),$$

with

$$D(t) = \begin{cases} \frac{3}{4}(1 - t^2), & \text{if } |t| \leq 1 \\ 0, & \text{otherwise} \end{cases}$$

and λ is the width of the local neighbourhood.

A generalisation of the kernel and its fixed neighbourhood width is to use a function $h_\lambda(x_0)$ and determine the width for each x_0:

$$K_\lambda(x_0, x) = D\left(\frac{d(x_0, x)}{h_\bullet(x_0)}\right)$$

Examples for $h_\bullet(x_0)$ are the constant function $h_\lambda(x_0) = \lambda$ or the k-nearest neighbourhoods function $h_k(x_0) = d(x_0, x_{[k]})$ where $x_{[k]}$ is the kth closest x_i to x_0.

In all cases the selection of the width is subject to the bias-variance tradeoff. If the width is too small, the estimator is an average of a small number of y_i close to x_0, hence its variance is relatively large and the bias tend to be small. On the other hand, is the width large, the variance of the estimator is small but the bias is high, because it uses x_i with the corresponding y_i further from x_0. A lot of methods exist to overcome this problem, examples are cross-validation or so-called plug-in methods (see Loader, 1999, section 6 about local regression).

Ranking

In our case, d is the distance measure defined in section 3.3. x_0 is the characterisation of the new dataset and x_j is the characterisation C_{D_j} of the dataset D_j. The corresponding y_i is, for a fixed candidate algorithm l_i, the error rate $ER_{l_i}^{D_j}$ or the benchmark time $T_{l_i}^{D_j}$ of the this algorithm on the dataset. The ranking then is obtained by estimating the performance measure NWE_{l_i} for each algorithm l_i and sort them in ascending order.

4.3 The latem Package, Part 2

This second part of the latem package description shows the classes and methods of the meta-knowledge base and of the meta-learner.

4.3.1 Meta-Knowledge Base

A meta-knowledge base consists of the Characterisation object and a list of arbitrary knowledge about some performance measures of each algorithm on each dataset. The structure of the knowledge is not defined, because different meta-learners need different structures.

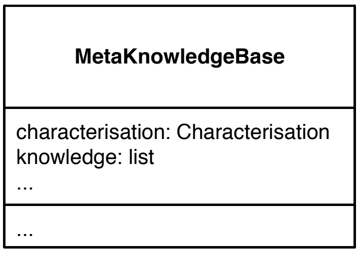

Figure 4.1: The MetaKnowledgeBase class.

4.3.2 Meta-Learner

As there are a lot of possibilities to implement such a meta-learner, this is just an interface definition. An implementation of a meta-learner has a method suggest which takes a MetaKnowledgeBase and a DataSet object and returns a Suggestion. I introduced the Suggestion class to present the results of a meta-learner in a correct context, e.g. relative ranking or prediction of performance measure.

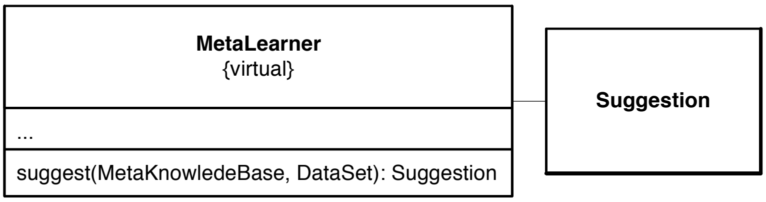

Figure 4.2: The MetaLearner and Suggestion interfaces.

The package contains the concrete implementations of the meta-learners introduced in the last section, MetaLearningZooming and MetaLearnerNwreg. Both need a DistanceFunction and a CharacterisationProcessingFunction. The implementation of the Nadaraya-Watson-Epanechnikov ranking is working with the k-nearest neighbourhood method for determining the kernel width, hence one have to define the number of neighbours k. Also, the zoomed ranking method needs the number of neighbours neighbours to zoom into and a ZoomingRankingFunction with its parameters.

The following example shows the usage of both meta-learners on the basis of the usage example from part one of the latem package description.

4.3.3 Usage Example

```
> library(latem)
```

Meta-Knowledge Base

This usage example is the continuation of the latter one. First, we build up the meta-knowledge base, and collect everything we know about the two datasets Heart1 and Spirals. The main part is the characterisation `c`:

```
usex: Characterisation
         2  datasets characterised
        with the Kalousis character list characteristics.
```

In this case, our knowledge consists of the performance measures `p` of the six candidate algorithms:

```
              lda naiveBayes          knn     rpart         svm      nnet
Heart1  0.1676137  0.1668039 0.4247871106 0.2367657 0.207225334 0.4503889
Spirals 0.4992615  0.5005232 0.0009292269 0.0319588 0.000734557 0.1108002
```

And the mean benchmarking time `t` in seconds for each experiment:

```
            lda naiveBayes     knn   rpart      svm    nnet
Heart1  0.09564    0.29588 1.13440 0.07792 43.72036 4.61660
Spirals 0.07424    0.48108 2.90628 0.06148 46.01156 6.27024
```

```
> kb = metaKnowledgeBase(c, list(error = p, time = t), "usex")
```

```
usex: MetaKnowledgeBase
        with knowledge of error, time
        about 2 datasets.
```

Meta-Learner

To show the usage of the different meta-learners and their settings, I use the most modified Heart1 dataset from the last usage example:

```
Heart1-4: DataSet
        Class ~ pain + trestbps + cholesteral + sugar + ecg + rate +
    angina + oldpeak + slope + vessels + thal
        with 303 samples,
        and the aspects input, response.
```

Zoomed Ranking: A meta-learner of this type is created using the `zooming` function. We have to define the number of neighbours, the distance with its preprocessing function and the ranking function with its parameters. In this usage example I use one neighbour and the previous defined `kalousis.dist` and `normalize.cproc` functions. A zoomed ranking meta-learner with the adjusted ratio of ratios ranking function is created in the following way:

```
> z.arr1 = zooming(kalousis.dist,
+     1, normalize.cproc, rankingFunction = arr.zoom,
+     rankingFunctionParams = list(K = 1))
```

```
Zoomed Ranking: MetaLearner
        Suggests ranking based on 1 neighbours,
        the distance function kalousis distance
        with normalize characterisation,
        and the ranking function adjusted ratio of ratios (K=1).
```

The meta-learners `z.ar` and `z.srr` with the ranking functions `ar.zoom` (average ranks) and `srr.zoom` (success rate ratios) are created in the same way.

Of course the Heart1 dataset is the nearest neighbour (independently from the ranking function)

```
> neighbours(z.arr1, kb@characterisation, h4)

     Heart1     Spirals
 0.00564081 39.56538189
```

and the different suggestions for the ranking of the candidate algorithms are:

```
> suggest(z.arr1, kb, h4)

lda < naiveBayes < rpart < knn < nnet < svm

> suggest(z.ar, kb, h4)

naiveBayes < lda < svm < rpart < knn < nnet

> suggest(z.srr, kb, h4)

naiveBayes < lda < svm < rpart < knn < nnet
```

When we compare these suggestions with our knowledge about the Heart1 dataset, they absolutely make sense.

Nadaraya-Watson-Epanechnikov Ranking: To create a meta-learner of this type one can use the function `nwe`. Again, the distance with its preprocessing function must be defined and the number of neighbours to determine the kernel width:

```
> n = nwe(kalousis.dist, normalize.cproc, k = 1)

Nadaraya-Watson-Epanechnikov Ranking: MetaLearner
        Suggests ranking based on 1 neighbours
        and the distance function kalousis distance
        with normalize characterisation
```

The kernel width for the Heart1-4 dataset is:

```
> b = kbandwidth(n, kb@characterisation, h4)

bandwidth
 19.78551
```

The resulting kernel weights for each dataset from the meta-knowledge base are:

```
> kweights(n, kb@characterisation, h4, b)

 Heart1 Spirals
   0.75    0.00
```

The suggestions for the ranking of the candidate algorithms based on the misclassification error and the benchmark time are:

```
> suggest(n, kb, h4, knowledge = "error")

naiveBayes        lda        svm      rpart        knn       nnet
 0.1668039  0.1676137  0.2072253  0.2367657  0.4247871  0.4503889

> suggest(n, kb, h4, knowledge = "time")

    rpart        lda naiveBayes        knn       nnet        svm
   0.07792    0.09564    0.29588    1.13440    4.61660   43.72036
```

Case Study: Suggestions

The idea of this case study is to use the results of the previous case studies and show the practical usage of meta-learning.

The setup is the following: The meta-knowledge base consists of the characterisation of the 21 datasets and the misclassification error and benchmark time of each of the 6 candidate algorithms on the datasets. The meta-learners are the zoomed ranking and the Nadaraya-Watson-Epanechnikov ranking methods. From the class of zoomed ranking methods I use the ones with the adjusted ratio of ratios ranking method (ARR) with the relative importance of time of 1, the success rate ratios method (SRR) and the average ranks method (AR). In all cases I use 3 as the number of neighbours, because the last case study showed that the smallest cluster with more than one member has this size.

Using this setup I predict for each of these datasets their rankings with the rest of the meta-knowledge base. The following are some selected cases by which I explain the gained results and then I show that the individual ranking suggestions are better than a global ranking.

Selected Suggestions

Circle dataset: This dataset is member of the top-left cluster (see figure 3.16), the candidate algorithms' true misclassification errors and benchmark times are the following:

Rank	Error		Time	
1	svm	0.007443	rpart	0.06952
2	knn	0.02458	lda	0.07548
3	nnet	0.03869	naiveBayes	0.4771
4	rpart	0.05626	knn	2.894
5	naiveBayes	0.0667	nnet	9.533
6	lda	0.4902	svm	51.05

Table 4.1: True rankings of the algorithms on the Circle dataset.

The three neighbours of this dataset are Spirals, ringnorm and threenorm, in ascending order considering their distances. If we focus on the ranking with respect to the misclassification error and compare their true rankings with the algorithm ranking on the Circle dataset, we see (figure 3.16) that the Spiral dataset has nearly the same ranking, but the two others are more differently. The suggestions of both meta-learners are given in the tables 4.2 and 4.3.

Rank	Error		Time	
1	svm	0.0009418	rpart	0.06554
2	knn	0.003749	lda	0.0765
3	rpart	0.03324	naiveBayes	0.4882
4	nnet	0.1120	knn	2.966
5	naiveBayes	0.4968	nnet	6.45
6	lda	0.4981	svm	47.54

Table 4.2: Nadaraya-Watson-Epanechnikov ranking suggestions of the algorithms on the Circle dataset.

The ranking suggestions of the Nadaraya-Watson-Epanechnikov meta-learner are almost correct, Spearman's rank correlation coefficient is 0.9429 for the error ranking and 1 for the time ranking. The concrete time value estimation is also closely to the true values, the absolute deviation is 6.677, but the error value estimation is poor, the absolute deviation is 0.5265. The reason is the big impact of the Spiral dataset, with kernel weight 0.75, and totally different error values. The ringnorm dataset has kernel weight 0.005337 and the threenorm dataset kernel weight 0.000588.

Rank	ARR_1	AR	SRR
1	knn	svm	svm
2	lda	nnet	rpart
3	naiveBayes	naiveBayes	nnet
4	nnet	lda	knn
5	svm	rpart	naiveBayes
6	rpart	knn	lda

Table 4.3: Zoomed ranking suggestions of the algorithms on the Circle dataset.

The comparison of the zoomed ranking meta-learner suggestions shows that from the error-only based ranking methods the success rate ratios ranking is better than the average ranks ranking, Spearman's rank correlation coefficients are 0.7714 (SRR) and 0.2571 (AR). The comparison of the adjusted ratio of ratios ranking method is not so easy, but if we do it in an informal way, we see that the ranking definitely makes sense.

BreastCancer dataset: This is an example where we get no useful suggestions concerning the misclassification error. The dataset is member of the top-right cluster (see figure 3.15) and the true rankings are:

Rank	Error		Time	
1	naiveBayes	0.027	rpart	0.09266
2	svm	0.03101	lda	0.2555
3	knn	0.04005	naiveBayes	0.5515
4	lda	0.04442	knn	4.34
5	nnet	0.04835	nnet	34.23
6	rpart	0.06405	svm	133.4

Table 4.4: True rankings of the algorithms on the BreastCancer dataset.

The three nearest neighbours are monks3, tictactoe and HouseVotes84, but their distances to the BreastCancer dataset are high, 0.002611 (monks3), 0.007781 (tictactoe), 0.020999 (HouseVotes84), in contrast with the latter example where the distance of the farthest neighbour is 0.00006406. And if we compare the algorithm rankings (figure 3.15) we see that the one on the BreastCancer dataset is quite unique.

Rank	Error		Time	
1	svm	0.01687	rpart	0.09356
2	lda	0.02686	lda	0.09953
3	nnet	0.05063	naiveBayes	0.5262
4	rpart	0.0526	knn	2.808
5	knn	0.09663	nnet	12.20
6	naiveBayes	0.1483	svm	28.11

Table 4.5: Nadaraya-Watson-Epanechnikov ranking suggestions of the algorithms on the BreastCancer dataset.

Rank	ARR_1	AR	SRR
1	knn	nnet	svm
2	lda	svm	lda
3	nnet	naiveBayes	nnet
4	rpart	lda	rpart
5	naiveBayes	rpart	knn
6	svm	knn	naiveBayes

Table 4.6: Zoomed ranking suggestions of the algorithms on the BreastCancer dataset.

Both meta-learners provide bad suggestions. The Nadaraya-Watson-Epanechnikov meta-learner ranking has a Spearman's rank correlation coefficient of -0.2 and an absolute deviation of 0.1655. The zoomed ranking provides suggestions with a coefficient of -0.2 (SRR) and 0.1429 (AR) and if we compare the ARR_1 suggestion we see that for example the naiveBayes algorithm is in the true ranking first (error) and third (time) but in the suggestion next to the last.

Suggestions vs. Global Ranking

A ranking method is then interesting if it performs better than a general global ranking. This global ranking is computed on the basis of the mean misclassification error over all the datasets and similar for the benchmark time:

Rank	Error	Time
1	svm	lda
2	rpart	rpart
3	nnet	naiveBayes
4	naiveBayes	knn
5	knn	nnet
6	lda	svm

Table 4.7: Global ranking.

For the benchmark time the global ranking is the correct one but also almost all meta-learners provide the right ranking, because the benchmark time induced ranking does not relatively vary a lot over the different datasets. For the error-induced rankings I take the Spearman's rank correlation coefficient as index for the quality of the ranking, the nearer at 1 the better the ranking. Using this interpretation, I can determine the best and worst ranking method for each dataset:

Dataset	Best Method	Worst Method
BreastCancer	AR	NWE
Cards	AR	*
chess	NWE	AR
Circle	NWE	AR
credit	AR	NWE
Heart1	NWE	*
hepatitis	NWE	AR
HouseVotes84	AR	NWE
Ionosphere	*	AR
liver	SRR	*
monks3	*	AR
musk	NWE	AR
PimaIndiansDiabetes	NWE	SRR
promotergene	*	AR
ringnorm	AR	SRR
Sonar	AR	NWE
Spirals	NWE	AR
threenorm	AR	SRR
tictactoe	SRR	*
titanic	NWE	AR
twonorm	NWE	SRR

Table 4.8: The best and worst ranking method considering the error for each dataset, whereas * denotes the global ranking.

We see that the global ranking is only three times the best suggestion. Overall the Nadaraya-Watson-Epanechnikov meta-learner provides the most usable suggestions. Thus, if we believe in the Nadaraya-Watson-Epanechnikov meta-learner and use its ranking suggestions to benchmark the methods in this order, the truly best method is benchmarked as the second one on average:

Suggested Rank	Number of Suggestions
1	9
2	6
3	1
4	2
5	1
6	2

Table 4.9: Nadaraya-Watson-Epanechnikov rank suggestion for the truly best method per dataset. Calculated with 10-fold cross-validation and majority vote.

Summary & Conclusion

In this thesis I investigated the topic of meta-learning, especially the meta-learning for machine learning approach. I explained this approach theoretically, showed an implementation for each part using the R system for statistical computation and used a case study with 21 classification learning problems and 6 classification methods to show the concrete usage.

Following the case study, I first benchmarked each method on all learning problems. As quality measures I used the misclassification rate and the benchmark time. Firstly, these experiments provided one part of the meta-knowledge base, and secondly, I used them to relate the problems in a clustering based on the rankings of the methods. The clustering pointed out that there are groups of problems with different best methods.

The second part of the meta-knowledge base, the characterisation of the problems, was obtained with various statistical and information-theoretical measures. Again, I clustered the problems based on their characterisation and this showed the separation of the 21 problems into three big clusters. The relation of these three clusters with the benchmarking results denoted some trends, but was not as clear as I assumed. The most likely reason is the small number of problems.

Each dataset characterisation and its according benchmark result form the basis for the meta-learning for machine learning approach. I applied two meta-learners, the Nadaraya-Watson-Epanechnikov meta-learner as regression-based approach and zoomed ranking with the average ranks, success rate ratios and adjusted ratio of ratios schemata. I compared the suggestions with the "true" rankings using the Spearman's rank correlation coefficient, the Nadaraya-Watson-Epanechnikov meta-learner provided the best suggestions.

The introduced methods produced workable suggestions, but of course there are a lot of possibilities to improve. Simplest improvements are the usage of more learning problems and more sophisticated meta-learners. The first one could be done either by altering the existing ones or by collecting real new data, but one should do both to cover most of the problem space. Then, the Nadaraya-Watson regression estimator is a really simple one, there exist a lot of more sophisticated methods, particularly for such a high-dimensional problem. Using such methods and some dimension reduction techniques may improve the suggestions.

Improvements also can be made in the meta-knowledge base. Maybe there are better ways to describe a problem and more information from the benchmark process should be used. In this thesis I simply aggregated the performance measures and did not use the produced empirical performance distributions or the significance results from the rankings.

Another interesting problem, in my opinion, is the estimation of hyperparameters. In mostly all cases the search for the best hyperparameters caused the long benchmark times, and maybe it is possible to use meta-learning to reduce the method's hyperparameter search space for a new problem based on familiar problems.

The R framework can also be extended, especially the meta-learner part could be extended to a better Lego-like framework.

Additional Benchplot Examples

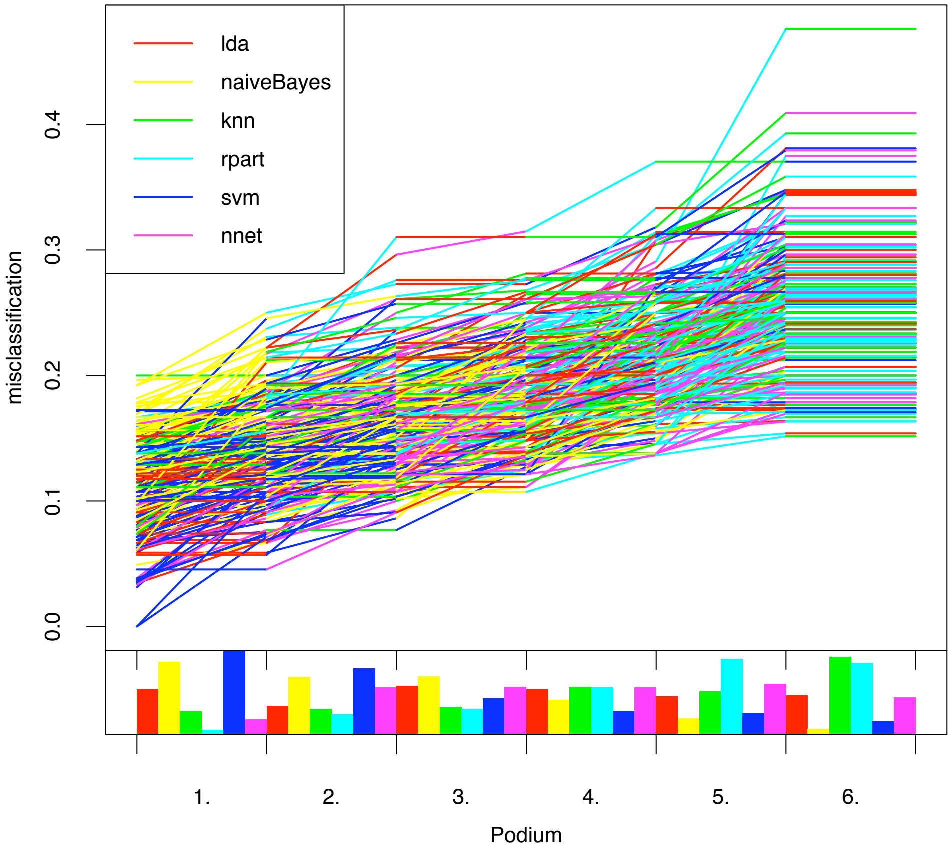

Figure 1: Benchplot of the benchmark result of the six candidate algorithms on the hepatitis dataset.

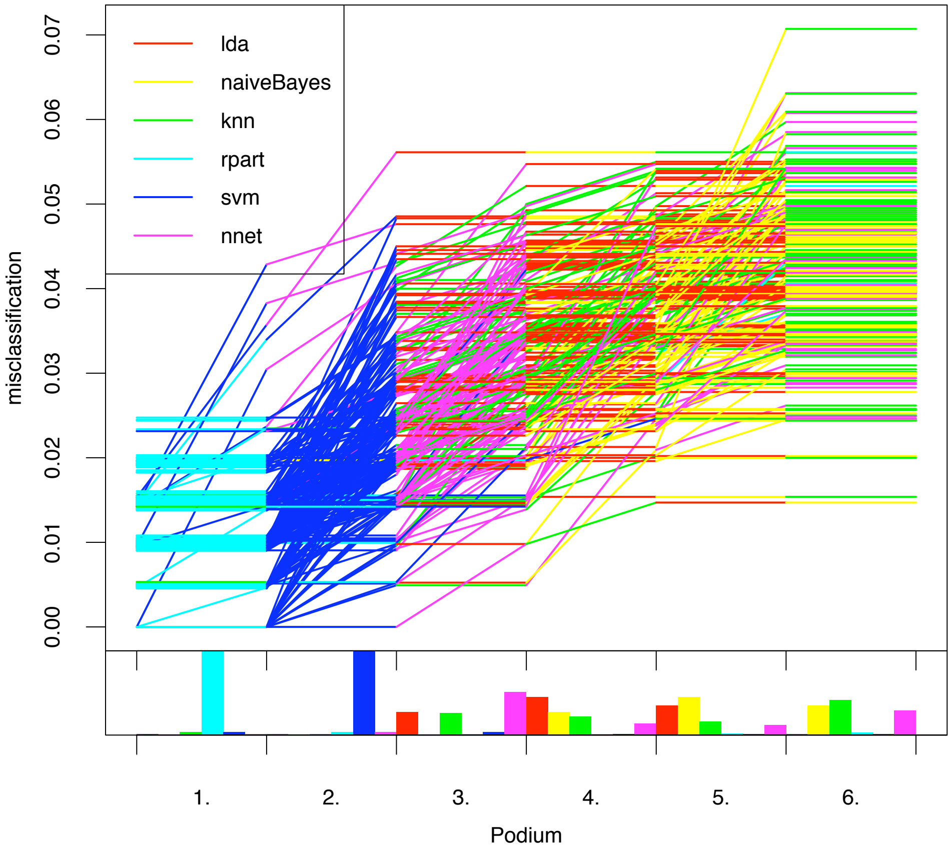

Figure 2: Benchplot of the benchmark result of the six candidate algorithms on the monks3 dataset.

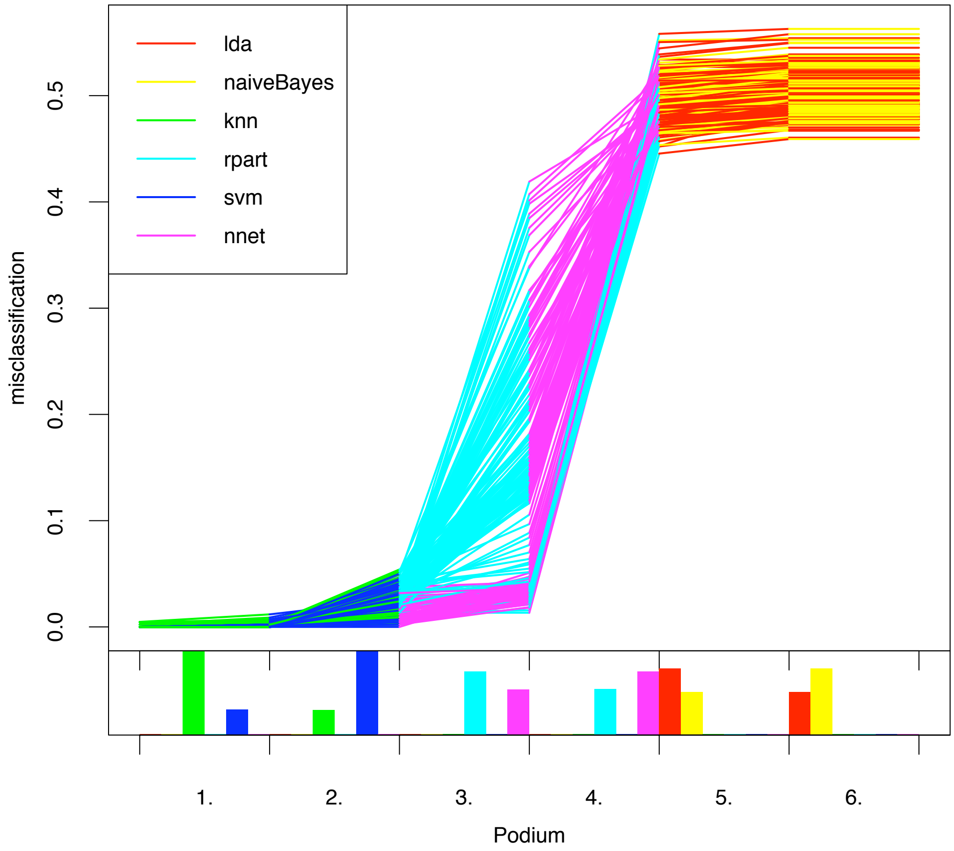

Figure 3: Benchplot of the benchmark result of the six candidate algorithms on the Spirals dataset.

Bibliography

Dana Angluin. Computational learning theory: survey and selected bibliography. In *STOC '92: Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 351–369. ACM Press, 1992.

Hilan Bensusan and Christophe Giraud-Carrier. Discovering task neighbourhoods through landmark learning performances. In *Proceedings of the 4th European Conference on Principles and Practice of Knowledge Discovery in Databases*. Springer-Verlag, 2000.

Hilan Bensusan and Alexandros Kalousis. Estimating the predictive accuracy of a classifier. In *EMCL '01: Proceedings of the 12th European Conference on Machine Learning*, pages 25–36. Springer-Verlag, 2001.

Pavel Brazdil and Carlos Soares. A comparison of ranking methods for classification algorithm selection. In *Machine Learning: ECML 2000, 11th European Conference on Machine Learning, Barcelona, Catalonia, Spain, May 31 - June 2, 2000, Proceedings*, volume 1810, pages 63–74. Springer-Verlag, 2000.

John M. Chambers. *Programming with Data*. Springer-Verlag, 1998.

Christophe Giraud-Carrier, Ricardo Vilalta, and Pavel Brazdil. Introduction to the special issue on meta-learning. *Machine Learning*, 54(3):187–193, 2004.

Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Element of Statistical Learning*. Springer-Verlag, 2001.

Myles Hollander and Doublas Wolfe. *Nonparametric Statistical Methods*. Wiley, 1999.

Torsten Hothorn, Friedrich Leisch, Achim Zeileis, and Kurt Hornik. The design and analysis of benchmark experiments. *Journal of Computational and Graphical Statistics*, 14(3):675–699, 2005.

Alexandros Kalousis. *Algorithm Selection via Meta-Learning*. PhD thesis, University Of Geneva, 2002. Thesis Number 3337.

Alexandros Kalousis and Melanie Hilario. Supervised knowledge discovery from incomplete data. In *Proceedings of the second International Conference on Data Mining 2000*. WIT Press, 2000.

Christian Köpf, Charles Taylor, and Jörg Keller. Meta-analysis: From data characterisation for meta-learning to meta-regression. In *International Symposium on Data Mining and Statistics*, 2000.

Clive Loader. Bandwidth selection: classical or plug-in? *Annals of Statistics*, 27(2):415–438, 1999.

David Meyer, Friedrich Leisch, and Kurt Hornik. The support vector machine under test. *Neurocomputing*, 55:169–186, September 2003.

Donald Michie, David Spiegelhalter, Charles Taylor, and John Campell, editors. *Machine learning, neural and statistical classification*. Ellis Horwood, 1994. URL `http://www.amsta.leeds.ac.uk/~charles/statlog/`.

Tom Mitchell. *Machine Learning*. McGraw Hill, 1997.

Bernhard Pfahringer, Hilan Bensusan, and Christophe Giraud-Carrier. Tell me who can learn you and i can tell you who you are: Landmarking various learning algorithms. In *Proceedings of the 17th International Conference on Machine Learning*. Morgan Kaufman, 2000.

Carlos Soares and Pavel Brazdil. Zoomed ranking: Selection of classification algorithms based on relevant performance information. In *Principles of Data Mining and Knowledge Discovery*, pages 126–135, 2000.

William Venables and Brian Ripley. *Modern Applied Statistics with S*. Springer-Verlag, fourth edition, 2002.

Ricardo Vilalta and Youssef Drissi. A perspective view and survey of meta-learning. *Artificial Intelligence Review*, 18(2):77–95, 2002.

Ricardo Vilalta, Christophe Giraud-Carrier, Pavel Brazdil, and Carlos Soares. Using meta-learning to support data mining. *International Journal of Computer Science and Applications*, 1(1):31–45, 2004.

David Wolpert. The supervised learning no-free-lunch theorems. In *Proceedings of the 6th Online World Conference on Soft Computing in Industrial Applications*, 2001.

List of Figures

List of Tables

Index