Efficient tuning in supervised machine learning
Koch, P.

Citation
Koch, P. (2013, October 29). *Efficient tuning in supervised machine learning*. Retrieved from https://hdl.handle.net/1887/22055

Version: Corrected Publisher's Version
License: Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden
Downloaded from: https://hdl.handle.net/1887/22055

Note: To cite this publication please use the final published version (if applicable).

Cover Page

The handle http://hdl.handle.net/1887/22055 holds various files of this Leiden University dissertation.

Author: Koch, Patrick
Title: Efficient tuning in supervised machine learning
Issue Date: 2013-10-29

Chapter 2

Methods

In this chapter we introduce methods and operators which are necessary elements for the experiments in this thesis. We start with a brief introduction to machine learning in Sec. 2.1 and proceed with more detailed descriptions of feature selection and feature construction in Sec. 2.2 and Sec. 2.3 respectively. Finally we give an overview about methods of Computational Intelligence in Sec. 2.4 and present the state-of-the-art of model-assisted optimization in Sec. 2.5.

2.1 Machine learning

Machine learning is the process of gaining knowledge using computers without programming the knowledge explicitly. In general two different types of learning can be distinguished:

- *Supervised learning*: direct training feedback is available in form of examples for which the output is known.
- *Unsupervised learning*: training is indirectly done, e.g., by performing self-learning methods such as trial-and-error.

In this thesis the focus lies on supervised learning, nevertheless parts of the research can also be applied to unsupervised learning.

2.1.1 Supervised learning

Supervised learning belongs to the most widely researched fields in artificial intelligence. The goal is to train a computer model which is capable to predict new data.

In supervised learning a mapping from input data to a target variable is trained by using examples. The target variable can be represented by discrete values (or classes), whereas we call this a classification task. The target can also be a continuous variable, and in this case we call the learning task *regression*. As a third special case we will also investigate learning tasks that have a time dependency, which we refer to as *time series* problems.

2.1.1.1 Learning from data

The data for machine learning is usually stored inside a file or a database. The entries of this file or database are called *instances* or *patterns*. A dataset for ML can be written as a sequence of pairs $D = (\vec{x}_1, y_1), (\vec{x}_2, y_2), \ldots, (\vec{x}_n, y_n)$, consisting of input data or feature vectors $\vec{x}^{(i)}$ and their corresponding outputs $y^{(i)}$. Each input $\vec{x}^{(i)}$ consists of N attributes describing the characteristics of the instance. Later we will also denote the input attributes as *features*. In supervised learning, the algorithms require outputs to train a prediction model. E.g., in classification they require at least one instance of each class in the training data. For *classification* tasks the target variables y_i are defined by a finite set with a fixed number of elements, the corresponding classes. In *regression*, the values y_i are defined by a continuous space, e.g. $y_i \subseteq \mathbb{R}$.

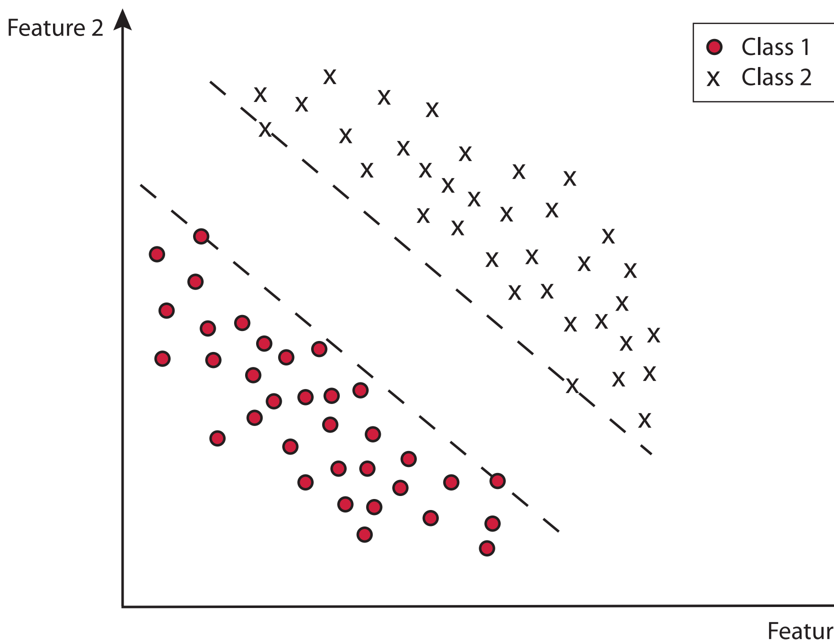

Figure 2.1: Linear separability of a binary classification problem. The figure shows data points (instances or pattern) in a two-dimensional feature space. The instances are marked by red points and black crosses for class 1 and 2 respectively. The two dashed lines illustrate the margin of possible separations of the classes. A sophisticated learning algorithm for classification would try to find a separating line with maximum margin to both class instances.

Figure 2.2: Poisonous or edible? The mushroom Amanita muscaria or “Fly agaric toadstool”. Picture by Tony Wills, License: Creative Commons Attribution 3.0 Unported

2.1.1.2 Classification

In classification we are interested in finding a separating hyperplane in the N-dimensional feature space, which is able to classify each pattern to its belonging target label. E.g., in Fig. 2.1 a two-class example is shown, where the class patterns are displayed as crosses and circles. The two dashed lines represent possible margins for each class. Every hyperplane between these two margin lines can separate the class patterns by classifying all points to the left side as circles and the points on the right side as crosses. An early learning algorithm that makes use of a separating hyperplane is the perceptron invented by Rosenblatt [207], which later led to the concept of Artificial Neural Networks (ANN) [109].

An intuitive example for classification is to classify mushrooms. Mushrooms can be either *edible* or *poisonous*. If we write down examples for mushrooms, including attributes like colour, size, points, etc., and denote the class label for each example, we can train a learning algorithm which is capable to classify mushrooms. The learning algorithm would use the known examples for predicting new unknown examples into edible or poisonous. Of course, the probability of classifying new samples, increases, if a certain number of training patterns is available, and under the assumption that the training data is correct.

2.1.1.3 Regression

In classification the output was considered as a discrete and finite variable. However, in many applications it is desired to predict numerical values like real numbers, e.g., to predict sensor measurements in an engineering process. In this case we perform *regression*, instead of classification.

Although regression is a different approach, real-valued output functions can also be learned by most algorithms which are proposed for classification. An easy to understand

model for regression is the *linear model*:

$$y_i = w_0 + w_1 x_{i,1} + w_2 x_{i,2} + \ldots + w_N x_{i,N} + \varepsilon_i \tag{2.1}$$

The linear model predicts the target by learning the optimal weighted sum of the input features. The weights $\vec{w} = w_1, ..., w_N$ are determined by an optimization procedure minimizing the error of the training set. The weight w_0 is just an offset and the term ε_i acts as *slack variable*. The weights for the model are based on the training data and can be determined by minimizing a loss function like the mean squared error (MSE). For predictions $\vec{\hat{y}}$ and real observations $\vec{y}$ the MSE for a training set of size n is defined as follows:

$$\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \tag{2.2}$$

In practice often the root mean squared error is considered:

$$\sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2} \tag{2.3}$$

It is possible to use interpretable models like the linear model in Eq. 2.1 to reveal feature importance. The disadvantage is that linear models are limited to easier data. More powerful models with the opportunity for interpretability are, e.g., multivariate regression splines (MARS) [87], or other techniques for symbolic regression like Genetic Programming [158].

2.1.1.4 Time series analysis

In many engineering applications data is dependent on a *time* label, which represents the information when the data was recorded. In history time series have been known at least since the 10th century (cf. Fig. 2.3). A well-known example of a time series application is the prediction of stock market prices. Here, the prices have an additional time information. Thus, the price of a stock is defined by its value at time t. Outgoing from time t the price of a stock can rise (hausse) or fall (baisse). It is important to note, that the price in the future $t + k$ is often dependent on time t and also on outer events from the environment which occurred in the period from $t + k$ up to some earlier time $t - \ell$. Of course the predictability in the future is usually limited to a certain k, because the stock market situation of today will only have a marginal influence to the situation in the next day or hour. But it is assumed that the time series can be predicted in the future up to a certain k. The learning algorithms can gain from adding influences from outside, e.g., actual news. The price of the stock can change according to the information given in such additional reports. The other assumption in time series analysis is *autoregression*, e.g., that the price will depend on the change of earlier values. This corresponds with time series like weather forecasts, where it can be assumed, that the weather in the next minute will be probably similar as before. Thus,

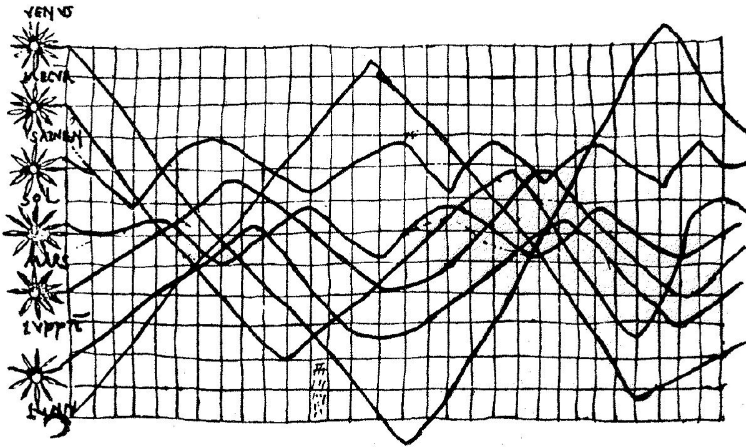

Figure 2.3: The figure shows a historic drawing of an early time series in the 10th century. Drawing found in [170, 241].

output values of time series data are correlated to a) the values of the time in the past and b) changes or actions in the underlying domain.

Usually a time series is recorded as a discrete stochastic process from time $T = 0$ to t:

$$(\vec{x}^{(0)}, y^{(0)}), (\vec{x}^{(1)}, y^{(1)}), \ldots, (\vec{x}^{(t)}, y^{(t)}) \tag{2.4}$$

In the simplest case the output is equal to the input $\vec{x}^{(i)} = y^{(i)}$, for all $i = 0, \ldots, t$. Then the time series can be simply written as

$$\vec{y} = y^{(0)}, y^{(1)}, \ldots, y^{(t)} \tag{2.5}$$

The time series in Eq. Eq. 2.5 is called a univariate time series. Here, the time series is solely defined by the probabilistic structure (the mean and variance) of the samples of y. In this case, classical time series analysis methods like ARIMA or GARCH processes can be used [31]. However, these methods are limited, because they are suited for linear processes and also assume a certain autocorrelation of the target. Instead, regression techniques from ML can be applied, to detect more difficult structures in the time series.

Today there is a large demand for analyzing multivariate time series. Here, at any time point t more than one variable is recorded. All variables have a certain influence on the target variable, which is in general unknown for arbitrary tasks.

2.1.2 Learning algorithms

The state-of-the-art in ML is difficult to grasp. Many algorithms exist, capable for many purposes, so that a certain favourite can hardly be named here. However, at least two algorithms showed good performances on a variety of tasks, and they can even be improved

by setting their parameters: random forest [26] as representative of decision tree-based methods and Support Vector Machines (SVMs) [245]. In this section we define the learning algorithms, a more detailed description on the underlying hyperparameters is then given in Sec. 4.2.

2.1.2.1 Random forest

Random forest (RF) [26] is a learning algorithm based on classification and regression trees (CART) [28]. RF uses an ensemble of decision or regression trees for the prediction. For ensemble methods we use a number of independent learning algorithms, which are later combined by an aggregation procedure. The goal is to get a better result using the ensemble, compared with possible weak results of single learning algorithms itself [102]. E.g., the CART approach by Breiman [28] has weaknesses, because the single trees are prone to overfitting. With the aggregation of many trees in an ensemble, such weaknesses of single learning algorithms are compensated. The most prominent two methods for ensemble learning are boosting [212] and bagging [24], which are described in the following sections.

RF uses the bagging approach (see Sec. 2.1.2.2) to create the tree ensemble. Additionally, a random feature selection for each node of the trees is taken from Ho [112, 113] and Amit and Geman [3]. These advantages of tree learners leads to a very robust algorithm, which is very stable to issues like overfitting or noise in the data.

Although RF is almost parameter-free, and already works well with default settings, it is sometimes advantageous, to optimize certain parameters [218]. As important parameters for RF, the number of trees (*ntree*) in the ensemble can be mentioned [232], as well as the number of splits performed in each tree node (*mtry*). Instead of performing a tuning of *mtry*, Liaw and Wiener [166] suggest the following defaults based on the number of features p:

$$mtry_{default} = \begin{cases} \max(\lfloor \frac{p}{3} \rfloor, 1), & \text{if task is regression} \\ \sqrt{p}, & \text{else} \end{cases} \tag{2.6}$$

It has to be noted, that this formula is only a rule of thumb. In our experiments we will also see advantages of tuning *mtry*. Other tuning options include the computation times of the algorithm, e.g., by setting the *sampsize* parameter (sample size used for the RF trees).

2.1.2.2 Ensemble approaches

Bagging

Bagging [24] is an acronym for *bootstrap aggregating*, where bootstrapping refers to a resampling method [67]. In bootstrapping k learning or training subsets of the training set size n are drawn repeatedly random with replacement from the complete training data. An ensemble learning algorithm would now train k prediction models (e.g., CART) for each

subset. For the prediction all models in the ensemble predict the test pattern. For the final output, the k predicted outputs of the models are *aggregated* by an aggregation function. As aggregation function, e.g., majority voting [190] can be used, or one can take the mean of all ensemble learners in regression.

Boosting

Boosting was presented by Schapire [211] and has been later improved by Freund and Schapire [86]. In boosting, the complete training data is used for the training, but a weight is assigned for each training pattern. At first the weights are uniformly distributed for all patterns, and then updated during the algorithm run. The weighting scheme is defined by variables $w_t(i)$, which denote the weight of pattern $\vec{x}^{(i)}$ in iteration t of the algorithm. In each iteration training patterns are sampled using $w_t(i)$ as probability distribution for the sampling with replacement. After having drawn the training data according to the weighting scheme all ensemble learners are trained using these sets. Then the weights are updated, assigning a smaller weight for correctly predicted patterns, and an increased weight for wrongly predicted patterns respectively. In the next round the sampling and training is applied again until a termination condition holds. Possible termination criteria are, e.g., a limited number of model trainings or a time limit.

When the target variable has a certain number of wrong values, the prediction accuracy degenerates for boosting, as shown by Dietterich [58]. For this reason bagging is used within RF.

2.1.2.3 Support Vector Machines

Support Vector Machines (SVMs) have been originally introduced as learning algorithms for binary classification and regression tasks [215], but they can also be used to solve multi-class problems, e.g., by incorporating multiple "one-against-all classifiers" (see [23] for a comprehensive overview).

Our main intention for using SVMs — besides that SVMs are known as a powerful ML tool — is that SVMs tend to be very sensitive to parameter settings, which makes them interesting for parameter tuning tasks. In binary classification, SVMs seek the maximal margin classifier, which best separates the two classes. In the simplest case this means to search for the optimal separating hyperplane by minimizing the empirical risk. However, if SVMs could only classify separable data, the applicability would be very limited, since most real-world problems are not linearly separable. Therefore, SVMs perform classification with the following extensions:

(1) Kernel-induced feature space: the input space is implicitly mapped to a higher-

dimensional space using a kernel function

$$K(\vec{X}, \vec{Z}) = \langle \phi(\vec{X}) \cdot \phi(\vec{Z}) \rangle \tag{2.7}$$

where ϕ defines a mapping from the input space. A simple example for such a transformation is to calculate the monomials of the input features and to consider them for the mapping. The kernel function denotes the similarity of two observations $\vec{X}$ and $\vec{Z}$. Because the kernel function can be interpreted as a dot product in a high-dimensional space, the computation is feasible also for very high dimensions. When the patterns cannot be separated by a linear classifier in the original feature space, this can be still possible in the kernel-induced feature space. The kernel function can be defined anew for each task, or pre-defined kernel functions can be chosen. It is important that kernel functions fulfill the Mercer theorem [180], as they have to be positive definite and symmetric (PSD property).

The most frequently used kernel functions for SVMs include the following functions:

– Linear

$$K(\vec{X}, \vec{Z}) = \langle \vec{X}, \vec{Z} \rangle \tag{2.8}$$

– Polynomial kernel:

$$K(\vec{X}, \vec{Z}) = \langle \vec{X}, \vec{Z} \rangle^d \tag{2.9}$$

– Radial basis function (RBF):

$$K(\vec{X}, \vec{Z}) = \exp\left(-\frac{||\vec{X} - \vec{Z}||^2}{2\gamma^2} \right) \tag{2.10}$$

where γ and d are parameters for the corresponding kernel functions.

(2) Regularized risk minimization: If some observations are still not classified correct in the kernel-induced feature space, SVMs can make use of the soft-margin concept by Cortes and Vapnik [51]. In soft-margin SVMs a regularization term is introduced, which penalizes wrongly classified patterns. Making this more rigorous, we can write the optimal SVMs classification model in the unconstrained dual form, denoting H as the reproducing kernel Hilbert space for the kernel function $K(\cdot, \cdot)$, in the following optimization problem:

$$\hat{F} = \arg \inf_{F \in H, b \in \mathbb{R}} ||F||_H^2 + C \sum_{i=1}^{n} L(Y_i, F(\vec{X}_i) + b) \tag{2.11}$$

Here F is the real-valued target function, and $\hat{F}$ the regularized target function. In case

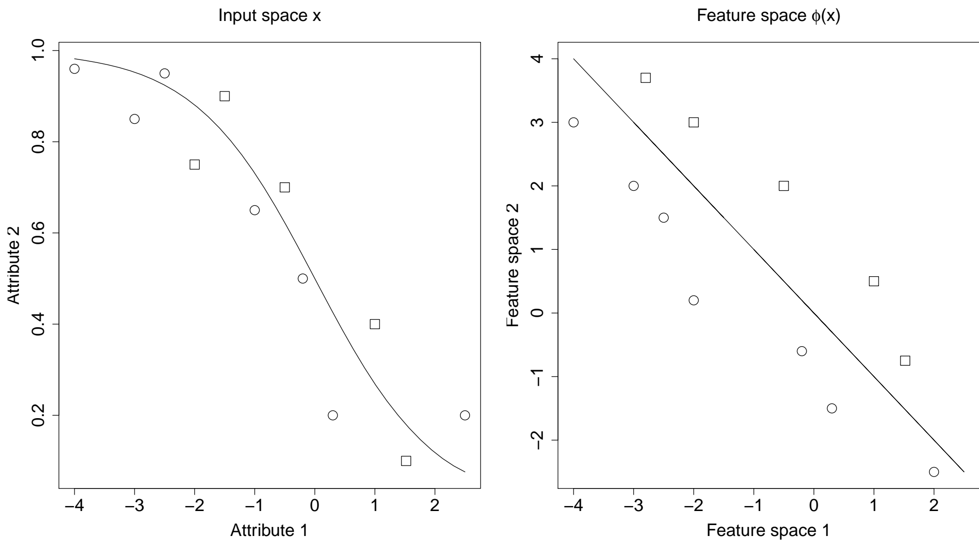

Figure 2.4: Mapping from input to feature space via kernel function ϕ. Before the mapping, the data is not separable, after the mapping a linear classifier can separate the class patterns.

of binary classification we would discretize the real-valued output of $\hat{F}$ by mapping it to $\{-1, 1\}$. The first term is called a smoothness penalty using a regulariser $||\cdot||_H^2$, where H is the Hilbert space of functions defined over the input domain [52, p. 40]. The 2-norm can be used to penalize non-smooth functions. The second term measures the closeness of our predictions to the true outputs by means of a loss function. In classification, we usually select the hinge loss $L(Y, t) = L_h(Y, t) = \max(0, 1 - Yt)$ for an intended output $t = \pm 1$ and a classifier score Y. For regression we often set $L(Y, t) = L_\epsilon(Y, t) = \max(0, |Y - t| - \epsilon)$ to the ϵ-insensitive loss. The hinge loss is a convex, upper surrogate loss for the 0/1-loss (which is of primary interest, but algorithmically intractable), while L_ϵ provides the estimation of the median of Y given $\vec{X}$. Both losses lead to quadratic programming problems for Eq. 2.11, which can be solved efficiently, and the non-differentiability of these two loss functions further provides for sparse solutions [43]. The two terms are balanced by the parameter C, sometimes also referred to as *Cost*. In recent SVM implementations, a value of $C = 1$ is taken as default, equally weighting loss function and smoothness penalty.[1]

The optimal kernel function may vary depending on the data. Without having any prior knowledge, the RBF kernel works well in most cases, presuming that the kernel parameters are set to good values. Of course the RBF and polynomial kernel functions are more complex

[1] In the *R* implementations contained in the *e1071* package [59] and in the *kernlab* package [136], the default value for C is set to 1.

and are able to classify non-separable data better than a linear kernel. But as a drawback they are more expensive to compute.

In regression, usually the Support Vector Regression (SVR) approach of Drucker *et al.* [63] is applied. For more information about SVR the interested reader is referred to the tutorial of Smola and Schlkopf [228]. For simplicity we do not distinguish between SVR and SVM in this thesis, but indicate which method is used where it is necessary.

A recent overview about the state-of-the-art in learning with kernels has been given by Signoretto and Suykens [221].

2.1.3 Generalization and benchmarking

In ML experiments a dataset

$$\mathcal{D} = \Big((\vec{x}^{(1)}, y^{(1)}), (\vec{x}^{(2)}, y^{(2)}), \ldots, (\vec{x}^{(m)}, y^{(m)}) \Big) \tag{2.12}$$

consists of m examples or instances and the corresponding output, e.g., a class label or real value.

If all training patterns in the data $\mathcal{D}$ are consistent, the optimal learning algorithm would always give the correct answer for every new example. Of course this is a very strong assumption for most datasets, and even when this holds, it remains unclear, if the learning algorithm also behaves optimal on future instances. Thus, in practice approximations of this optimal model are computed. In order to compare these models, we have to define a procedure, under which we can measure the model quality. This is known as benchmarking of ML algorithms.

Benchmarking of learning algorithms belongs to one of the most controversely discussed topics in ML and statistics: Hothorn *et al.* [116] describe a theoretical framework for benchmarking. Eugster and Leisch [73] and Eugster *et al.* [72] present techniques for visualizing benchmark results. Hornik and Meyer [114] propose to use consensus rankings for benchmarking ML algorithms. Bischl *et al.* [20] discuss resampling strategies for tuning of learning algorithms and describe common pitfalls and advantages of the methods.

In experiments it is important to take into account statistical analyses to provide sound results. Public benchmark sets like the UCI repository [84] are available and can be downloaded from the internet. There are at least two reasons for using public datasets: evaluation with public datasets makes the results of algorithms reproducible, because every researcher is able to test his method on these datasets. Second, researchers can help in improving methods when they are applied to common datasets.

However, we also think that public benchmark sets alone might be not sufficient for a sound benchmarking of algorithms and can result in overfitting to certain benchmark repositories itself. Besides that, the motivation behind developing new algorithms is to solve new problems. Here, public benchmark datasets might be too restrictive. Instead we think that with a combination of public benchmark sets and real-world data better progress can

be made. Learning algorithms are also thought to find hidden structures in unknown data, and if benchmarks are too restrictive, it can happen that in the sequel the development of learning algorithms stagnates.

In the following paragraphs, we introduce methods for evaluating ML algorithms. A desirable property of ML algorithms is, that they can generalize well and perform good on other unseen data. Therefore, special resampling strategies exist, which play a key role for all experiments made in this thesis.

2.1.3.1 Sampling strategies

ML models should be well generalizing, to learn a model for the dataset. The learning algorithm should be able to generate predictions for new patterns $\vec{x}^{(o)}$ with a maximal accuracy. In this section sampling methods are proposed, which aim at estimating the *generalization error*. These sampling methods include:

- Random sampling (holdout set)
- k-fold cross-validation
- Leave-one out cross-validation
- Sub-sampling
- Stratified sampling

Random sampling

For training and testing purposes the dataset $\mathcal{D}$ is usually split randomly into disjoint subsets

$$\mathcal{L} \subset \mathcal{D} \tag{2.13}$$

and

$$\mathcal{T} \subset \mathcal{D} \tag{2.14}$$

with $\mathcal{L} \cup \mathcal{T} = \mathcal{D}$. Now each learning algorithm can be trained on the training data $\mathcal{L}$, while it can be evaluated on test data $\mathcal{T}$. This accuracy estimation method is called *holdout method*, because the patterns of set $\mathcal{T}$ are held out during training.

Cross-validation

Another method for estimating the generalization performance is the k-fold cross validation (CV). Here the data is split into k disjoint subsets. For each subset $\mathcal{D}(i), i \in$

$\{1, \ldots, k\}$ a prediction model is trained, predicting the remaining patterns

$$\mathcal{D}(j) = \mathcal{D} \backslash \mathcal{D}(i) \tag{2.15}$$

Finally the errors of each learned model can be aggregated, e.g., by taking the mean of the errors.

Leave-one out cross-validation

Leave-one out cross-validation (LOOCV) is a special case of k-fold cross-validation, where the number k is equal to the number of patterns of the dataset ($k = |\mathcal{D}|$). LOOCV certainly gives the best estimate of the real generalization performance. Unfortunately LOOCV belongs to the most expensive methods to evaluate learning algorithms, and for this reason it is not well suited when many models must be built, as is the case in parameter tuning of learning algorithms.

Sub-sampling

Normally all available training patterns in the data set $\mathcal{D}$ are used for training. Sometimes $\mathcal{D}$ tends to be very large, and the training takes a lot of time. Additionally, outliers and redundant patterns can be misleading for learning the concept. In such cases, we can perform a training using only small subsets (subsamples) of the available training data. This sampling strategy is called *sub-sampling* and is very effective, when it is repeatedly performed, e.g., in a tuning process.

Learning with reduced training set sizes can give remarkable speed-ups for the model-building process, which is the main reason for using it in practice. Last [161] proposes partial learning by projective sampling to estimate the optimal training set size. For Artificial Neural Networks [38] and Support Vector Machines [220], intelligent sub-sampling near the decision boundary helps to speed-up the training process without loss of accuracy. In many studies it was observed that training with smaller sets does not necessarily lead to worse generalizing prediction models. Instead, the complexity of the trained models can be reduced as shown by Oates and Jensen [187] and Provost *et al.* [196]. Some people think that simpler hypotheses should be preferred, which was constituted in Occam's razor [22].

Stratified sampling

When performing sub-sampling, it can happen that the training set sizes tend to be very small, which is especially the case for smaller datasets. Another situation where this issue can occur is data, where the classes are imbalanced. This can lead to training sets, where whole classes are missing. Stratified sampling [186] is a sampling strategy for classification to solve this issue. Similar to other random sampling strategies, stratified sampling draws patterns

at random, but draws them from each *stratum* or class. By doing this the distribution of the class probabilities remains unchanged.

2.2 Feature selection

Feature processing probably belongs to one of the most important steps for obtaining better prediction models. In this section we introduce various methods for feature subset selection, which are capable to form better feature starting sets. It is shown in [255], that although the dataset itself does not contain more information with a reduced feature subset, the selection can be beneficial for the generalization performance. Although this does not seem to be very logical at first sight, the reason is that unimportant features in the dataset can deteriorate results. With a better feature set, noise is reduced in the data and a better classification or regression model can be obtained. Due to the high importance of feature selection, we include hyperparameters of feature processing algorithms in the tuning, in order to generate more powerful learning algorithms.

Feature subset selection can be of large importance, because the model benefits from a reduced feature subset, where non-informative features are excluded. Guyon and Elliseeff [101] present a comprehensive overview about this topic. In general we can distinguish between three different types of feature selection:

- *Filter approaches* give rankings of the features without training a learning algorithm

- *Wrapper approaches* return feature subsets by applying a specific learning algorithm in an iterated manner

- *Embedded methods* are integrated into some learning algorithms, which enables off-the-shelf feature rankings

All approaches have their advantages and disadvantages, which are briefly described in the following sections.

2.2.1 Filter approaches

Feature selection using filter methods is solely based on criteria without requiring to build and to evaluate a specific learning algorithm. For this reason these methods are very attractive, because building learning algorithms for large-scale data can become computationally expensive, and filter methods always provide a quick alternative for measuring the information content of a variable then.

2.2.1.1 Correlation-based filtering

A well-known measure to determine the similarity of two variables is the linear correlation. For two variables $\vec{x}$ and $\vec{y}$ the correlation is defined by

$$\rho = \frac{\sum_{i=1}^{m}(x_i - \mu(x_i)) \cdot (y_i - \mu(y_i))}{\sum_{i=1}^{m}\sqrt{(x_i - \mu(x_i))^2} \cdot \sqrt{(y_i - \mu(y_i))^2}} \tag{2.16}$$

where $\mu(x_i)$ and $\mu(y_i)$ are the mean values of variable $\vec{x}_i$ and $\vec{y}_i$ respectively.

The output ρ lies in the range between -1 and 1, indicating a negative or positive correlation between the two variables.

2.2.1.2 Entropy and information gain

Instead of the linear correlation measure, often the entropy is been used for measuring the information content of a variable $\vec{x}$. The entropy is defined by

$$H(\vec{x}) = \sum_{i=1}^{m} Prob(x_i)\log(Prob(1/x_i)) \tag{2.17}$$

Now we can define the following quantity for measuring the entropy of $\vec{x}$ after $\vec{y}$ was observed:

$$H(\vec{x}|\vec{y}) = -\sum_{j=1}^{m} Prob(y_j)\sum_{i=1}^{m} Prob(x_i|y_j)\log_2(Prob(x_i|y_j)) \tag{2.18}$$

where $Prob(x_i|y_j)$ is the posterior probability of $\vec{x}$ under $\vec{y}$.

As an alternative to the entropy, Quinlan [197] introduced the information gain, which is based on the entropy and became popular as splitting rule in the C4.5 learning algorithm [197]:

$$IG(\vec{x}|\vec{y}) = H(\vec{x}) - H(\vec{x}|\vec{y}) \tag{2.19}$$

2.2.1.3 Random forest importance

Random forest includes a method for ranking features, which is called *random forest importance* (RFI). RFI cannot be classified as a filter approach, because it exhibits several differences to the other approaches mentioned so far, but it is not a wrapper either. Instead it can be categorized as an *embedded method*, because although RF is used for determining a feature ranking, the ranking is rather *built-in* the learning algorithm [101]. For this reason a prediction model has to be learned only once to obtain a feature ranking and not multiple times like in wrapper approaches.

The advantage of the RFI method is that it can detect variable interactions, which the other filter approaches were not able to. The idea is as follows [27]: permute a variable or feature by looking how much the prediction error increases when all other variables remain the same. Two measures are considered for the RFI:

- the mean decrease in accuracy and

- the mean decrease in Gini index [92] describing the node impurity measure [25, 169] which is often used in tree-based classifiers.

These measures are obtained by performing the permutation test: When a certain feature V_j is permuted, the out-of-bag (OOB) performance or the Gini index changes. The difference between this performance and the prior performance is calculated for all trees. The average of the tree values then describes the RFI measure for feature V_j.

2.2.2 Wrapper approaches

In wrapper approaches the learning algorithm is used to perform a prediction of the task given a certain feature subset. In general words a search in the feature subset space is performed, using the prediction error as objective function value to be minimized. Well-known examples are exhaustive search (which is NP-hard, making the runtime intractable in high dimensions unless $P = NP$), Genetic Algorithms or feature forward selection and backward elimination. Kohavi and John [152] give a comprehensive survey of wrapper approaches. We describe the most often used approaches in the following sections.

2.2.2.1 Feature forward selection

Feature forward selection is an iterative procedure where one feature is selected in the beginning. Using this single feature a model training is performed. After evaluating the model, the accuracy on the evaluation set is taken as feature importance. In the first iteration the algorithm would use all features to build N prediction models. The feature with the best prediction accuracy is then selected as most important feature. Outgoing from this feature all other features are added and again N models are built. The procedure is iterated, until no more improvement with adding new features can be made.

The disadvantages of the forward selection method are obvious: first, if the number of features of the dataset is large, many model trainings must be performed to obtain a feature subset. This gets even more expensive, when a large fraction of the features is advantageous for the learning algorithm. Secondly, the method cannot detect complex variable interactions. Let us assume that features V_1 and V_2 together are perfectly suited to predict the target variable. However, feature V_1 and V_2 both do not give any improved prediction when considered exclusively. This means, that the method would converge and both features V_1 and V_2 would never be added to the feature subset.

2.2.2.2 Genetic feature selection

Another heuristic for feature selection is the classical Genetic Algorithm (GA) [98]. A set or population of binary strings of length N (corresponding to the number of features) is initialized first. A '1' at position i in the binary string indicates, that the feature is selected, while a '0' would mean to exclude the feature.

Note, that number of possible feature sets increases exponentially with the number of available features/attributes (2^N). For this reason a genetic search can become computationally expensive. Another point is that empty feature sets must be avoided, e.g., a string full of zeros would lead to crashes for most learning algorithms. Therefore in most implementations of genetic feature selection a parameter constant is added, denoting the number of features to be used.

2.3 Feature construction

In this section we describe the creation of new features by performing specific transformations or projections. We show how these transformations can be learned semi-automatically providing only a set of basis functions. Although datasets sometimes already contain attributes which are good descriptors of the target, it can be helpful to derive new features using the original attributes. These can be *projections* or other transformations performed on the data. Well-known data transformation methods are, e.g., the Principal Component Analysis (PCA), or logarithmic and Fourier (spectral) transformations. Another possibility is to calculate monomials of the input attributes of degree d.

In some cases features can be derived by recommendations of experts. However, this requires of course a detailed knowledge about the properties of the data. For this reason we propose to use methods, which require only little knowledge, and which can be applied to most numeric datasets without any prior information. In Sec. 3.1 we show how user-based feature construction can be combined with model-assisted tuning for improving prediction models. Another feature processing method which has received only little attention so far is the Slow Feature Analysis. Combined with a simple classifier it can be used as a state-of-the-art learning algorithm, producing almost as good results as a SVM and RF, but in much shorter time. Otherwise the main purpose is to use it as a pre-processing method for other learning algorithms. We applied SFA to a gesture recognition problem and achieved remarkable results.

2.3.1 Principal Component Analysis

Principal Component Analysis (PCA) was firstly proposed by Pearson [189] and later formalized and named by Hotelling [115]. The goal of PCA is to find the directions in the data which have the highest variance. It is a classical projection method for learning with reduced feature sets. PCA transforms the coordinate system by determining the maximum variance of the input dimensions. PCA is helpful when attributes are correlated. In such cases PCA transformations can help to reduce the total number of features, without losing too much information.

In the application of PCA, in a first step, the data is mean-centered. Afterwards, the covariance matrix is calculated for the points. The PCA then diagonalizes the covariance

matrix to obtain the eigenvectors. Then, the eigenvectors are sorted in order of decreasing eigenvalues. Note that the eigenvalues can represent the data, because they inherit the variances of the samples in the dataset. Eigenvalues are also often used in image calculations, since transformations can easily be made when the structure is available. Now we can use them for reducing the number of features of the data. With fewer features based on the largest eigenvalues of the covariance matrix we can often achieve better predictions for the task.

Formally PCA can be defined as follows: the covariance matrix of the data is denoted by $\vec{C}$. Now the eigenvectors of this matrix are determined, and sorted in order of decreasing eigenvalues. The matrix of the eigenvectors is written as $\vec{U}$. Now we simply calculate the product of $\vec{U}$ and $\vec{C}$:

$$\vec{Y} = \vec{U}^T \vec{C} \tag{2.20}$$

The output matrix $\vec{Y}$ comprises the transformed feature space, from which we can now select the first d rows, to reduce the original m-dimensional feature set to $d < m$ principal component features. The more components are selected, the less variance of the data can be observed in the transformed space. Because PCA is affected by scaling, all attributes are at first standardized to zero mean and unit variance. Instead of using PCA, sometimes the Singular Value Decomposition (SVD) proposed by Golub and Van Loan [100] is considered, since it is more robust in determining the eigenvectors, especially when the covariance matrix is singular.

The most relevant drawbacks of PCA and comparable methods are, that they are only suited to make transformations of real-valued attributes. If discrete attributes are present in the data, PCA is not applicable any more. A frequently used approach is then to remove these features from the dataset and to apply PCA only to the numerical part of the data.

From our perspective PCA is a part of the learning process and therefore should be also incorporated into the tuning. Although no direct parameter is required for PCA, the number of features d to be selected from the transformed feature space, can be seen as a hyperparameter.

2.3.2 Monomials

Monomials are products of attributes or features of degree d. Assume we have three features denoted by a, b, c in the dataset. The set of all possible monomials of degree 2 would then be

$$F := \{a^2, b^2, c^2, ab, ac, bc\} \tag{2.21}$$

It can be valuable to calculate such feature sets and use them instead of only the basis attributes. The reason is the higher-dimensional space the data is projected in. The main disadvantage is, that monomials of higher degrees, or monomials for a large-scale dataset are

expensive to calculate. For this reason monomials are usually only calculated for the most important features. The importance of the features can be determined by using a feature ranking, e.g., through a filter, or by PCA through the level of variance of the principal components.

2.3.3 Slow Feature Analysis

Slow Feature Analysis (SFA) is a learning algorithm from neuroscience which is capable of learning unsupervised new features or "concepts" from time series. SFA was originally developed in context of unsupervised learning of learning invariances in the visual system of vertebrates [252]. In [253] and [254] a detailed overview about the algorithm is given. Although SFA is inspired from neuroscience, it does not have the drawbacks of conventional Artificial Neural Networks (ANNs) such as long training times or strong dependencies on initial conditions. Instead, SFA is fast in training and it has the potential to find hidden features out of multidimensional signals, as shown by [16] for handwritten-digit recognition.

SFA is optimally suited to construct features for time series signals. The original SFA approach for time series analysis is defined as follows: For a (multivariate) time series signal $\vec{x}(t)$ where t indicates time, find the set of real-valued output functions $g_1(\vec{x}), g_2(\vec{x}), ..., g_M(\vec{x})$, such that each output function

$$y_j(t) = g_j(\vec{x}(t)) \tag{2.22}$$

minimally changes in time[2]:

$$\Delta y_j(t) = \langle \dot{y_j}^2 \rangle_t \text{ is minimal} \tag{2.23}$$

The Δ-value can be described by measuring the slowness of an output signal as the time average of its squared derivative [254]. To exclude trivial solutions we add some constraints:

$$\langle y_j \rangle_t = 0 \text{ (zero mean)} \tag{2.24}$$

$$\langle y_j^2 \rangle_t = 1 \text{ (unit variance)} \tag{2.25}$$

$$\langle y_k y_j \rangle_t = 0 \text{ (decorrelation for } k > j) \tag{2.26}$$

The third equation is only relevant from the second slow signal on to prevent higher signals from learning features already represented by slower signals.

For arbitrary functions this problem is difficult to solve, but SFA tries to find a solution by expanding the input signal into a nonlinear function space by applying certain basis functions, e.g., monomials of degree d. This expanded signal is sphered to fulfill the

[2] $\langle \cdot \rangle_t$ means average over time and $\dot{y}$ indicates the time derivative.

constraints of Eq. 2.24, Eq. 2.25 and Eq. 2.26. Then SFA calculates the time derivative of the sphered expanded signal and determines from its covariance matrix the normalized eigenvector with the smallest eigenvalue. Finally the sphered expanded signal is projected onto this eigenvector to obtain the slowest output signal $y_1(t)$.

Berkes [16] extended this approach to classify a set of handwritten digits. The main idea of this extension is to create many small time series out of the class patterns: let us assume that for a K-class problem each class $c_m \in \{c_1, ..., c_K\}$ has got N_m patterns. We then reformulate the Δ-objective function (2.23) for SFA with distinct indices k and l as the mean of the difference over all possible pairs:

$$\Delta(y_j) = \frac{1}{n_{pair}} \cdot \sum_{m=1}^{N_m} \sum_{\ell=k+1}^{N_m} \left(g_j(p_k^{(m)}) - g_j(p_\ell^{(m)}) \right)^2 \tag{2.27}$$

where n_{pair} denotes the total count of all pairs and $p_k^{(m)}$ and $p_l^{(m)}$ represent the k-th and l-th class pattern of class m. The constraints defined by Eq. 2.24, Eq. 2.25 and Eq. 2.26 can be reformulated then by substituting the average over time with the average over all patterns, such that the learned functions have a zero mean, unit variance and are decorrelated [16].

As shown by Berkes [16], the $(K-1)$ slowest SFA output signals are expected to have a low intra-class variation, but usually a high inter-class variation. Therefore Berkes [16] proposes to train a standard Gaussian classifier on the slowest $(K-1)$ SFA outputs produced from the training records. The Gaussian classifier will seek an optimal position and shape of a Gauss function for each class in this $(K-1)$-dimensional space. The class probabilities of patterns $\vec{x}$ are then defined by the posterior probabilities according to the Bayes decision rule.

2.3.4 Genetic Programming for feature processing

Genetic Programming (GP) [158] is a technique, which discloses a large variety of usage (see Sec. 2.4.4). GP can be applied to construct features, by learning non-linear combinations of the basis features. In contrast to monomials, GP has a much higher complexity of the search space. In earlier approaches, Krawiec [159] used GP for feature construction to build better features from the original feature set. Krawiec discovered that good features could be constructed using GP, but he also observed a remarkable overfitting to the training data. Likewise Smith and Bull [224] used GP for constructing new features, and combine the construction with a Genetic Algorithm (GA) for feature selection. They achieved improved results in 8 of 10 datasets compared with C4.5 [197], but also observed the problem of overfitting in some cases. Therefore they provided a reordering strategy, which enables better generalization performance again. We think that the large degree of freedom of GP can be an advantage, but can also be a great disadvantage at the same time. It is difficult to define good parameters for GP, and other issues like the large runtime caused by the wrapper approach and the large search space remain. Besides this, the overfitting problem must be handled. For this reason the search for better feature sets with GP is promising, but can become elusively slow, which can make it inefficient for practical use.

2.4 Stochastic optimization

Nonlinear optimization problems arise in many fields like engineering, mathematics or computer science. In contrast to linear optimization problems, nonlinear optimization problems are usually solved with heuristics.

2.4.1 Formulation of the problem

In a search space $S \subseteq \mathbb{R}^m$ we seek for the best solution $\vec{x}^* \in S$ of an evaluation function f:

$$f(\vec{x}) = f(x_1, x_2, ..., x_m) \tag{2.28}$$

The search space S can be of continuous type, that is $S \subseteq \mathbb{R}^m$, but can also contain discrete parts, or can be restricted by bounds.

In general, we can assume that the underlying problem is a minimization problem, that is we are seeking a point where the function value of f is minimal (if this exists):

$$f(\vec{x}) \to \min \tag{2.29}$$

Note, that maximization problems can be re-formulated to minimization problems, but without loss of generality minimization problems are considered in this thesis:

$$\max f(\vec{x}) = -\min(-f(\vec{x})) \tag{2.30}$$

Another point is that the parameters for learning algorithms are usually constrained, meaning that we need to respect at least box-contraints in the form of lower ($\vec{lB} \in S$) and upper bounds ($\vec{uB} \in S$), for the components of $\vec{x}$:

$$\begin{aligned} lB_1 \leq \quad & x_1 \quad \leq uB_1 \\ lB_2 \leq \quad & x_2 \quad \leq uB_2 \\ & \vdots \\ lB_m \leq \quad & x_m \quad \leq uB_m \end{aligned} \tag{2.31}$$

In the following sections we describe approaches for solving such optimization problems. It has to be remarked, that our objective is to solve the optimization problems *globally*, that is we want to find a *best* solution, i.e., the vector producing the minimal value of f.

2.4.2 Local search

Local search comprises methods for stochastic optimization by refining a candidate solution. A well-known example is the classical steepest descent, which moves along the gradient of the function in a sequential process. Other methods for local search have been proposed, e.g., the Newton method, which uses the inverse Hessian matrix for a better estimate of the best search direction. However, as we are performing black-box optimization where no analytic solution of the objective function exists, it is not possible to calculate derivatives analytically.

Instead, direct search methods can be used which perform a random stochastic search, and thereby try to approximate the gradient or the Hessian matrix. An example of such a heuristic is the algorithm of Broyden, Fletcher, Goldfarb and Shanno (BFGS) [32, 81, 99, 219], which approximates Newton's method, but without requiring second-order derivatives to guide the search.

The BFGS algorithm is based on the method by Davidon [54] and Fletcher and Powell (DFP) [81], where new points are determined by deriving information from the previous search steps:

$$\vec{x}^{(k+1)} = \vec{x}^{(k)} + \vec{s}^{(k)}\vec{v}^{(k)} \tag{2.32}$$

where $\vec{s}^{(k)}$ is the step-size and $\vec{v}^{(k)}$ is an approximation of the inverse Hessian matrix.

Algorithms like BFGS are local search methods, i.e., they are performing a search using a starting point and guiding the search to a optimum. In the best case the global optimum is reached, but it can happen that the algorithm converges to a near local optimum. A simple strategy for finding the global optimum with local search methods is to perform *restarts*, that is initiating multiple local searches with different starting points. However, it might be the case that this restarting strategy is not successful either.

2.4.3 Evolution Strategies

Evolution Strategies (ES) are search heuristics inspired by biological evolution. They perform as well biologically-inspired variation as selection to guide the search to the optimal solution. One of the main advantages of ES is, that they don't require any additional information like gradient information. ES can be used for global optimization, at least when a set of solutions (a population) is initiated for approximating the global optimum. ES were firstly developed by Rechenberg [203] and Schwefel [217] in the 1960s.

The essential ingredient of today's state-of-the-art ES is a completely derandomized adaptation of the mutation step-sizes. This step-size adaptation was first-time established in the well-known Covariance-Matrix-Adaptation ES [107, 108]. Many extensions have been proposed for CMA-ES, including strategies for uncertainty-handling [105, 106], or mirrored sampling [30] which can improve the original algorithm. In our experiments with ES we use the CMA-ES by Hansen [104], but are aware of the fact that other variants can profitably support the CMA-ES. E.g., although the CMA-ES was originally proposed as a local optimization strategy in [107], we also use it as a global optimizer with larger population size and a uniformly but random initialization strategy based on Latin hypercube sampling (cf. Sec. 2.5.2). Instead of that, Auger and Hansen [4] have established the increasing population sizes (IPOP) strategy for global optimization on multimodal landscapes.

2.4.4 Genetic Programming

Genetic Programming (GP) can be seen as a substantial part of Evolutionary Algorithms (EAs). It was originally proposed for the automatic generation of computer programs [7, 158, 193]. Today, it has especially received interest in applications of *symbolic regression*. The goal of symbolic regression is to find a functional relationship between given input and measured output signals. Thereby it would be equivalent to regression, but in symbolic regression a symbolic representation of the functional relationship (e.g., by a mathematical expression) is returned. State-of-the-art is Pareto GP [227] which enables to determine functional relationships of varying complexities. Starting with a high-level problem definition, GP creates a population of random symbolic expressions, termed *individuals*, that are progressively refined through an evolutionary process of variation and selection until a satisfactory solution is found.

Although GP requires no prior knowledge about the solution structure it can be difficult to apply it to functions, where certain structures are required. The goal of GP is to minimize the error of a given task, which is usually defined by a *fitness function* like in ES. An inherent advantage of GP is the representation of solutions as symbolic expressions, i.e., as terms of a formal language, which makes them accessible to human reasoning and symbolic computation. The main drawback of GP is its high computational complexity due to the potentially infinitely large search space of symbolic expressions.

For applying GP, several problem specific and algorithm specific parameters have to be specified:

Fitness function

A fitness function associates a numerical fitness value to a candidate solution represented as a symbolic expression. This function encodes the task to be solved. GP is an optimization algorithm in the sense that it searches for solutions that (by convention) minimize this fitness function.

Symbolic expressions

Any GP function consists of function symbols, constant symbols, and variable symbols, used for constructing symbolic expressions. Together with the variation operators, these building blocks define the structure of the GP solution search space.

Initialization strategy

The *initialization strategy* defines how the initial GP population is generated. Often complete randomized strategies are considered for this, but we want to bias our search to

more simple individuals. Therefore a strategy that grows individuals to a random tree depth less then or equal to a maximum tree depth given as a parameter is employed [158].

Variation operators

Like in ES, variation operators are methods for mutating and recombining existing solutions. Because the implementation of these operators is highly dependent on the solution representation (tree, graph, etc.), a variety of different operators have been developed. Still, the classical mutation and crossover operators originally proposed by Koza often work well in practice and are used in a type-safe manner [158, 193].

General EA parameters

The remaining parameters are common to most Evolutionary Algorithms and include among others population size, selection strategy, and termination criteria. Most generic extensions to Evolutionary Algorithms, such as niching and automatic restarts, can be directly applied to GP.

2.4.5 Other search spaces

Besides the investigation of continuous parameter spaces (e.g., in Evolution Strategies or local search methods), other search space types are possible. Examples of such parameters are integer or discrete values. Schwefel [217] invented an ES for integer search spaces with a binomially distributed mutation operator. Compound representations with real-valued, integer and discrete attributes can be solved with a special formulation of ES as proposed by Bäck and Schwefel [6]. Various applications using this ES formulation can be found in [5, 70, 165]. It has to be noted, that discrete and integer parameters can not simply be considered as continuous values, but the variation operators can be adopted, and special mutation distributions can be used inside this mixed-integer ES formulation.

2.5 Model-assisted optimization

It is often desired to find solutions of nonlinear optimization problems under very limited budgets. While optimization heuristics like ES often require many function evaluations until they converge in an optimum, an alternative can be to perform the main part of the optimization on a surrogate model. This is what we call *model-assisted optimization*. The idea of model-assisted optimization is, that the evaluations of the surrogate model are very cheap, while the real function might be expensive. For this reason the main part of the optimization can be performed on the surrogate function, while the real objective function must be only considered for re-fitting of the surrogate model and for validation purposes.

2.5.1 Related work

In the Evolutionary Computation (EC) field global optimization problems are precisely solved by techniques as presented in Sec. 2.4 like the CMA-ES by Hansen and Ostermaier [107] or Differential Evolution (DE) by Storn and Price [231]. As both strategies often require a lot of function evaluations, we will describe strategies which are especially suited to solve optimization problems with very restrictive budgets. Nevertheless the strategies of EC are well understood and still remain important, because global optimization on the easier surrogate function can finally be performed using these strategies.

Jones *et al.* [133] presented the efficient global optimization (EGO) algorithm. This method is especially designed to solve very expensive functions which frequently occur in industry. EGO makes use of Kriging [160, 201] as a surrogate model, and the expected improvement (EI) criterion, which are both important concepts and are described in detail later in this thesis. Other frameworks for parameter optimization include the Relevance Estimation and Value Calibration (REVAC) by Nannen and Eiben [184]. REVAC was developed to determine robust parameter settings for evolutionary algorithms. While REVAC was mainly proposed to find robust parameters, Smit and Eiben [222, 223] extended the method with other heuristics to reduce the computation times. Birattari *et al.* [17] invented the F-Race algorithm based on the racing algorithm by Maron and Moore [173]. The algorithm ranks solutions by statistical tests for steering the search more directly. Birattari *et al.* [17] tested their algorithm especially for combinatorial optimization problems with good performances.

Bartz-Beielstein *et al.* [11] invented the sequential parameter optimization (SPO), which combines methods from classical Design of Experiments (DoE) [76] and Design and Analysis of Computer Experiments (DACE) [209]. EGO by Jones *et al.* [133] and SPO share that they are both model-assisted iterated optimizers, which means that they train and refine a model for saving evaluations on the real objective function f. EGO and SPO have several parallels, but also differ in the flexible number of repeated evaluations and the choice of the surrogate model in SPO. In Section 2.5.3 we give a brief overview of the SPO algorithm.

Based on the work of SPO and Racing [173], Hutter *et al.* [123] started their research with a variant called iterated local search (ILS), and later extended their work in [122] where they compare a variant of SPO which is evaluated on several deterministic test functions. The main change is to use log-transformations of the response values to give better estimates for the surrogate model. This idea has also been discussed by Wagner and Wessing [250] where they compare various response transformations for EGO. In a recent work Hutter *et al.* [121] propose the sequential model-assisted algorithm configuration (SMAC) which also incorporates Kriging and RF surrogate models.

The main difference between the existing model-assisted approaches and other algorithms for stochastic optimization is, that model-assisted optimization aims at keeping the number

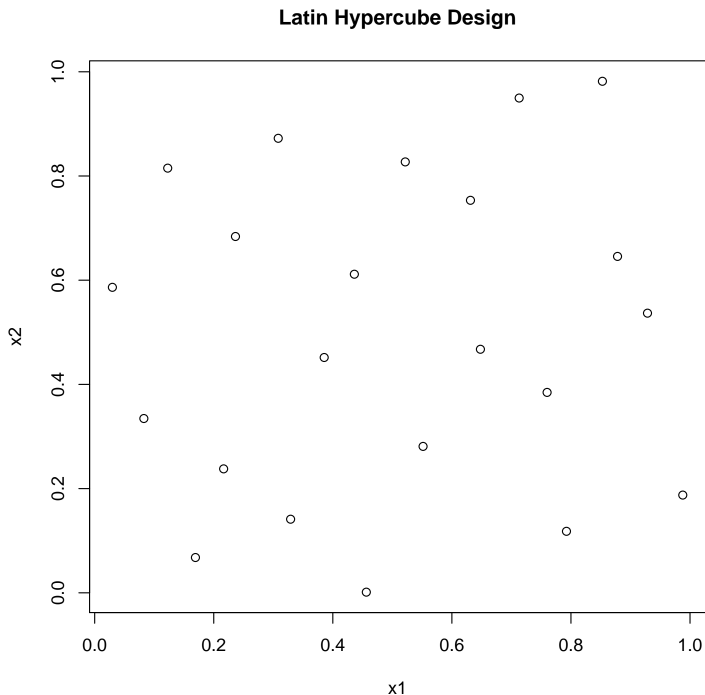

Figure 2.5: The Figure shows an optimal latin hypercube design of size 22 in a two-dimensional feature space with bounds $[0, 1]$ for both features $x1$ and $x2$.

of function evaluations relatively low. Besides that, sometimes also more about the sensitivity or importance of the parameters can be learned. This can support later analyses like the landscape analysis described in Chapter 6.

2.5.2 Design of Experiments

Design of Experiments (DoE) and here especially Latin hypercube sampling [175] can be seen as a starting point for model-assisted optimization. For exploration of the search space, a random uniform distribution of the design points is created. The number of points must be given, which dictates the coverage of the search space. However, in contrast to random sampling, the distribution of the design points most likely will lead to a better exploration of the search space, at least for small dimensions.

In classical Design of Experiments (DoE) [76] often systematic sampling is applied, to give good exploration of the search space. E.g., in Latin hypercube sampling (LHS) [175] a restricted search space $S \subseteq \mathbb{R}^m$ is assumed, having box-constraints as described in Eq. 2.31. When very few parameters have to be set, a simple grid search or LHS can be very effective, assuming that the optimum does not lie in narrow and steep valleys. But when the dimension

increases, these methods usually have difficulties due to the *curse of dimensionality* [172].

2.5.3 Sequential parameter optimization

In this section we describe the sequential parameter optimization (SPO) approach, where an arbitrary surrogate function is trained to enable cheaper and faster optimization runs. SPO was originally designed as a framework for improving the performance of optimization algorithms by experimentation. But SPO can also be used as an optimizer, which is also our main domain in this thesis. The SPO research started with the analysis of stochastic search algorithms [13], including *Evolution Strategies* (ES) and *Simulated Annealing* (SA) [143] to give a better understanding about the behaviour of these algorithms.

SPO combines methods from classical *Design of Experiments* (DoE) [76] and modern *Design and Analysis of Computer Experiments* (DACE) [209]. The optimization loop is visualized in figure 2.6. DoE dominates the first phase of the SPO development. This approach includes several established and well-understood procedures for the analysis of deterministic and stochastic data, especially regression and analysis of variance techniques.

In the first phase of SPO, an exploration of the parameter search space is performed. Therefore, a set of initial design points $\vec{x}$ is generated by a sampling strategy of the user's choice. Usually a space-filling design $DES := (\vec{x}^{(1)}, \vec{x}^{(2)}, ..., \vec{x}^{(l)})$ like a Latin hypercube sample of size $\ell = |DES| \in \mathbb{N}^+$ is created. All ℓ initial design points are then evaluated at least once on the real objective function f resulting in a vector of responses $(y_1, y_2, ..., y_\ell)$. SPO now uses the initial design points DES and their responses y for fitting a surrogate function. In general, any regression function or surrogate model can be selected therefore, but often Kriging is chosen, since it is almost parameter-free and can handle most landscapes. We will introduce this technique in more detail in Sec. 2.5.4.

The idea is now to sequentially refine this model by performing a pre-defined number of function evaluations. The prior information can be used for better exploration of the search space and to improve the surrogate models. The surrogate model serves as a cheap alternative for evaluations on the true objective function and a much larger number of evaluations can be performed to produce new design points, which in turn are evaluated by the algorithm and used to update the model. As surrogate model often Kriging is used, because it can nicely model non-linear functions with only few available design points. However, SPO does not require a specific model and provides an interface for setting other surrogate models like RF or linear models.

SPO distinguishes itself from other surrogate modeling approaches by the following components:

- refinement of the surrogate model by establishing additional design points,

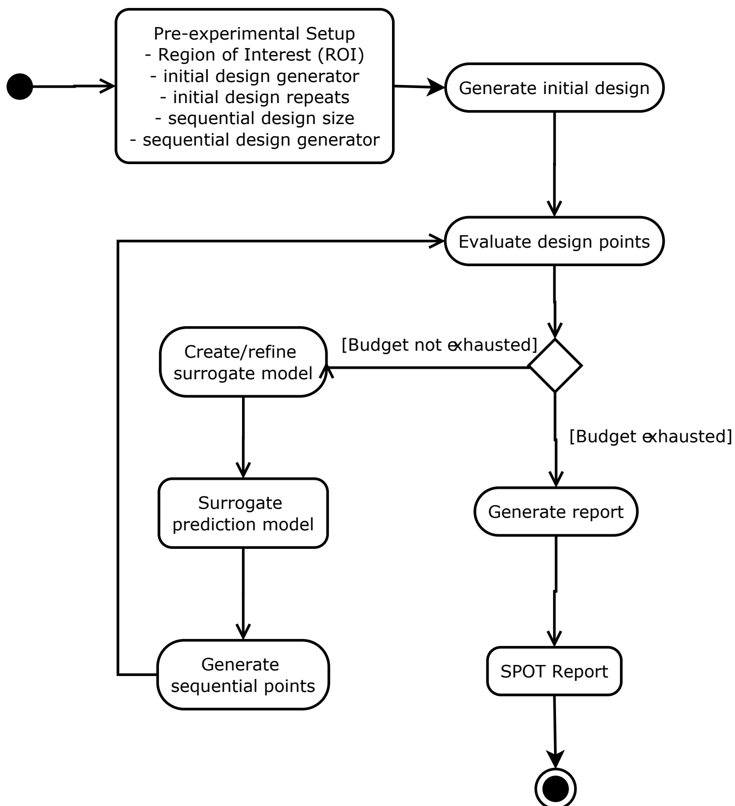

Figure 2.6: The sequential parameter optimization process.

Algorithm 1: Sequential parameter optimization (SPO)

```
// phase 1, building the model:
```
let A be the algorithm we want to tune;
generate an initial design $DES = \left(\vec{x}^{(1)}, \ldots, \vec{x}^{(n)}\right)$ of n parameter vectors;
let $k = k_0$ be the initial number of replications for determining estimated responses;
foreach $\vec{x} \in DES$ **do**
  run A with $\vec{x}$ k times to determine the estimated response y of $\vec{x}$;
end
```
// phase 2, using and improving the model:
```
while *budget not exhausted* **do**
  let $\bar{\vec{x}}$ denote the parameter vector from DES with best estimated response $\bar{y}$;
  let k be the number of repeats already computed for $\bar{\vec{x}}$;
  build surrogate model $Y(\vec{x})$ based on DES and $\left(y^{(1)}, \ldots, y^{(|DES|)}\right)$;
  optimize the model with respect to some utility function;
  thus produce a set DES' of d new parameter vectors;
```
  // improve confidence
```
  run A with $\bar{\vec{x}}$ once and recalculate its estimated mean response using all $k + 1$ test results; let $k = k + 1$;
  run A k times with each $\vec{x} \in DES'$ to determine the estimated mean response;
  extend the design by $DES = DES \cup DES'$;
end

- repeated evaluations of design points using a dynamic number of performed evaluations (OCBA) and an aggregation function (e.g., mean) to receive a comparable quality measurement.

In Algorithm 1 the pseudo code of SPO is presented. Note, that in this section we will use the notation $\vec{x}^{(i)}, y^{(i)}$ for the data passed to the surrogate model instead of the data used for the learning algorithms.

In the sequential improvement loop SPO optimizes the current model $Y(\vec{x})$ over the considered space of input variables by means of a utility function. In the simplest case this utility function is the estimated output itself as this should reflect the performance of the algorithm A. But more sophisticated criteria are possible to select the next sampling points, and in case of Kriging surrogate models they are often termed infill criteria. When the new sampling points have been selected, the number of replications is increased, the required evaluations at the design points are performed and the surrogate model is updated.

The whole procedure serves two primary goals. One is to determine good parameter settings for A, thus SPO may be used as a tuner. Secondly, variable interactions can be revealed in order to understand how the tested algorithm works when confronted with a specific problem or how changes in the settings influence the algorithm's performance. The SPO approach tries to tackle both goals of (i) tuning and (ii) understanding complex procedures, e.g., optimization algorithms or machine learning models.

2.5.4 Kriging

In the 1950s Krige [160] presented an regression technique, which was mathematically formalized by Matheron [174] and later became popular as Kriging. Sacks *et al.* [209] used the Kriging approach in their Design and Analysis of Computer Experiments (DACE). Jones *et al.* [133] successfully integrated the DACE approach for performing a global search on various test functions yielding in the efficient global optimization (EGO) algorithm. Inspired by the findings in DACE and DoE, Bartz-Beielstein *et al.* [11] integrated Kriging as surrogate model into the sequential parameter optimization framework, which offers the user an easy to handle interface with a fully flexible design of inner components like the initial design choice, and infill criteria.

2.5.4.1 Kriging in SPO.

First of all we make the usual assumption for a general regression setting that the output data are subject to model errors, i.e., we have noisy measurements $y^{(i)}$ at the i-th data point $\vec{x}^{(i)}$, where $\epsilon^{(i)}$ is the measurement noise.

The SPO approach consists of two steps:

(1) model construction and

(2) optimizing the model

We will only describe the model construction for Kriging here, where both steps are based on the maximum likelihood estimation (MLE) approach presented in [132, 83].

The Kriging model is constructed as follows: Assume we have a number of evaluated points (e.g., an initial design by LHS) denoted by $\left((\vec{x}^{(i)}, y^{(i)})\right)$, $(i = 1, \ldots, n)$. The observed responses $\vec{y} = \left(y^{(1)}, y^{(1)}, \ldots, y^{(n)}\right)$ are considered as if they were from a Gaussian process, i.e., we will use a set of random vectors $\mathbf{Y} = \left(Y(\vec{x}^{(1)}), \ldots, Y(\vec{x}^{(n)})\right)^T$ with associated $n \times n$ correlation matrix

$$\mathbf{\Psi} = \left(\mathrm{cor}(Y(\vec{x}^{(i)}), Y(\vec{x}^{(j)}))\right)_{i=1,j=1}^{n} \tag{2.33}$$

and correlation function

$$\mathrm{cor}(Y(\vec{x}^{(i)}, Y(\vec{x}^{(l)}) = \exp\left(-\sum_{j=1}^{n} \theta_j (x_j^{(i)} - x_j^{(l)})^2\right), \tag{2.34}$$

where θ is the correlation parameter of the Kriging model. Under standard assumptions, cf. [83], the likelihood can be measured by

$$\begin{aligned} L(\mathbf{Y}^{(1)}, \ldots, \mathbf{Y}^{(n)} | \mu, \sigma) = \\ \frac{1}{(2\pi\sigma^2)^{n/2}} \exp\left(-\frac{\sum(\mathbf{Y}^{(i)} - \mu)^2}{2\sigma^2}\right). \end{aligned} \tag{2.35}$$

In order to filter noise, a regression constant λ can be added to the leading diagonal of $\mathbf{\Psi}$, which is known as nugget effect in the literature [248]. Expressing Eq. 2.35 in terms of the sample data, taking derivatives, and setting to zero, we obtain the estimates

$$\hat{\mu} = \frac{\vec{1}^T(\mathbf{\Psi} + \lambda \mathbf{I})^{-1}\vec{y}}{\vec{1}^T(\mathbf{\Psi} + \lambda \mathbf{I})^{-1}\vec{1}}, \tag{2.36}$$

$$\hat{\sigma}^2 = \frac{(\vec{y} - \vec{1}\hat{\mu})^T(\mathbf{\Psi} + \lambda \mathbf{I})^{-1}(\vec{y} - \vec{1}\hat{\mu})}{n}, \tag{2.37}$$

and the *concentrated ln-likelihood function*

$$\ln(L) \approx -\frac{n}{2}\ln(\hat{\sigma}^2) - \frac{1}{2}\ln\det(\mathbf{\Psi} + \lambda \mathbf{I}). \tag{2.38}$$

The unknown Kriging parameters can be determined, that is the vector $\vec{\theta}$ introduced in Eq. 2.34 and the regression constant λ, by maximizing the concentrated ln-likelihood function.

In a next step an optimization on this model can be performed using any optimization heuristic (e.g., CMA-ES or any iterated local search strategy).

A prediction at $\vec{x}$ can be given by

$$\hat{y}(\vec{x}) = \hat{\mu} + \vec{\psi}(\vec{x})^T(\mathbf{\Psi} + \lambda \mathbf{I})^{-1}(\vec{y} - \vec{1}\hat{\mu}), \tag{2.39}$$

with $\hat{\mu}$ as defined in Eq. 2.36 and $\vec{\psi}(\vec{x})$ the vector of correlations between the observed data and a new prediction, i.e.,

$$\vec{\psi}(\vec{x}) = \Big(\mathrm{cor}(Y(\vec{x}^{(1)}), Y(\vec{x})), \ldots, \mathrm{cor}(Y(\vec{x}^{(n)}), Y(\vec{x}))\Big)^T.$$

The variance (model uncertainty) at this prediction can be estimated by

$$\begin{aligned} \hat{s}^2(\vec{x}) = & \hat{\sigma}^2\Bigg(1 - \vec{\psi}(\vec{x})^T(\mathbf{\Psi} + \lambda \mathbf{I})^{-1}\vec{\psi}(\vec{x}) + \\ & + \frac{1 - \vec{\psi}(\vec{x})^T(\mathbf{\Psi} + \lambda \mathbf{I})^{-1}\vec{\psi}(\vec{x})}{\vec{1}^T(\mathbf{\Psi} + \lambda \mathbf{I})^{-1}\vec{1}}\Bigg), \end{aligned} \tag{2.40}$$

with $\hat{\sigma}^2$ as defined in Eq. 2.37. Because of the inclusion of the nugget effect λ Eq. 2.40 does not completely reduce to zero when we calculate it at an already evaluated sample point, which can be advantageous for noisy optimization.

2.5.4.2 Infill criteria

After having built the Kriging model for the current design, we can perform an optimization on the surrogate function $\hat{f} = \hat{y}(\vec{x})$. This can be done by generating new design points for which a search on the surrogate function is performed. New promising points (infill points) are then evaluated on the real function, and thereafter the surrogate model is updated with the new information. It is wise not simply to choose the best response from

the surrogate model $\hat{f}$ as next candidate solution, but to select the next point using the *expected improvement* (EI) criterion proposed by Jones *et al.* [133] which is defined by the following function:

$$EI(\vec{x}) = E\left[max(f_{min} - Y(\vec{x}), 0)\right] \tag{2.41}$$

where f_{min} is the minimum of all previously obtained real evaluations, and $Y(\vec{x})$ is the random variable of the Kriging surrogate model, which is usually assumed to be normal distributed with mean $\hat{y}(\vec{x})$ and standard deviation $\hat{s}(\vec{x})$.

Jones *et al.* [133] have shown in their article that under this assumption the expected improvement can be calculated in closed form

$$EI(\vec{x}) = (f_{min} - \hat{y}(\vec{x}))\Phi\left(\frac{f_{min} - \hat{y}(\vec{x})}{\hat{s}(\vec{x})}\right) + \hat{s}(\vec{x})\rho\left(\frac{f_{min} - \hat{y}(\vec{x})}{\hat{s}(\vec{x})}\right) \tag{2.42}$$

where $\Phi(\cdot)$ and $\rho(\cdot)$ are the cumulative distribution function and probability density function of the normal distribution, respectively. Note, that the expected improvement in its presented form is only valid for deterministic problems, in other words with a regression constant of $\lambda = 0$. For the expected improvement the gradient can be calculated analytically, but as it is a multimodal function it is usually maximized with an Evolutionary Algorithm or by employing restart strategies.

The EI deals as infill criterion, that is by maximizing the EI new promising candidate points can be determined. While interpolating surrogate-models like Kriging assume uncertainties in undiscovered regions of the search space, an uncertainty of zero is inserted at evaluated points. As described in Sec. 2.5.4.1, in case of noisy observations a regression constant λ can be added to the leading diagonal of $\mathbf{\Psi}$. The EI can support the exploration, because high variances in unknown regions can lead to points with a better $\hat{y}$ promising as the next infill candidate. A more detailed study on infill criteria has been given by Picheny *et al.* [192].

2.5.5 Optimization in noisy environments

In noisy optimization, the target function is not deterministic. A computer experiment would produce an output $y_k^{(1)}$ for a point $\vec{x}^{(k)}$, but when the same point is evaluated again, a different output $y_k^{(2)}$ is produced, which can differ from $y_k^{(1)}$. In the following, we present two possible solutions to this.

2.5.5.1 Replications

Although the nugget effect tries to filter noise within the model predictions, it still can happen that very high noise levels occur. The usual procedure to cope with high noise levels in DoE is to use replicated measurements and to calculate an aggregated value, e.g., the mean of all replicates. This reduces the noise level and may avoid wrong tuning decisions.

But the price for replicated measurements under a limited budget is that fewer infill points can be generated: e.g., if each design point is evaluated three times, we have only one third in the number of infills.

Standard aggregation procedures like calculating the mean of a series of evaluations can be taken as value for the Kriging model. Since Kriging always assumes a certain confidence, it is valuable to pass the replicates to the Kriging model to refine the uncertainty estimation of the model. In this case SPO can make use of replications, that are a pre-defined number of evaluations, that are aggregated via a function $Aggregate$. Often the mean is chosen for this aggregation.

$$Aggregate(\vec{y}) := \frac{1}{m}\left(y^{(1)} + y^{(2)} + ... + y^{(m)}\right) \tag{2.43}$$

Also, different strategies for choosing the number of replications are possible, e.g., spending more repeats in the final exploitation phase, while spending only a small number of repeats in the beginning of the optimization. In order to keep things simple, we present a strategy where in each iteration the newly proposed points are evaluated $n_{repeats}$ times.

The aggregated value is finally fed to the Kriging model. This procedure is the usual way to cope with noise in DoE.

2.5.5.2 Re-interpolation

An elegant method for noisy Kriging optimization (NKO) is the re-interpolation (RI) method by Forrester *et al.* [82]. In SPO either the strategy using repeated evaluations described in Sec. 2.5.5.1 can be used or the *re-interpolation* is incorporated for this purpose: in re-interpolation the Kriging model is fitted twice. A first model is fitted with a non-interpolating Kriging variant due to the noisy observations, then the response values $y^{(i)}$ of the data are substituted by the predictions of the regression model. The data is interpolated in a second step by an interpolating Kriging model, for which the expected improvement can be calculated without modification. The corresponding pseudocode of the RI procedure is shown in algorithm 2.

Algorithm 2: Re-interpolation procedure for noisy Kriging optimization

Set initial design $DES := \left(\vec{x}^{(1)}, \vec{x}^{(2)}, ..., \vec{x}^{(k)}\right)$
Fit non-interpolating model mod_1 using nugget estimation for design DES
Evaluate DES using mod_1
Fit interpolating model mod_2 for responses from previous step
Maximize expected improvement for mod_2

Also, it is provable that for the second model the parameters of the covariance kernel from the first model are already optimal, leading to a much faster fitting algorithm.

2.5.5.3 Comparing Kriging and SVM

Note, that there is a close connection between Kriging (Gaussian processes) and SVM and this holds whether they are used to model regression or classification problems. This connection becomes clearer if one compares the resulting optimization problems for both models and especially if we consider an SVM with least squares loss function instead of the hinge loss. The most important difference is that no full stochastic model or interpretation for the SVM is currently known. Some work has been done by Sollich in that regard [230], but his interpretation does not seem to be universally accepted by the community (Rasmussen and Williams call his construction "rather contrived" in [201]). A full and proper discussion of this connection would exceed the scope of this work and the reader is referred to [74, 188, 201].

2.6 Conclusions

In this chapter we introduced different types of learning, including classification, regression, and time series problems. Learning algorithms that are capable to solve these learning tasks were described, as well as methods to evaluate them. We focus on two different learning algorithms, that are Support Vector Machines (SVMs) and random forests (RFs). Although there exist a large number of other learning algorithms, these two algorithms usually result in good performances on many instances or datasets. The generalization performance was introduced as desired performance measure, but its calculation is computationally intractable. For this reason we gave a number of other evaluation measures, including cross-validation, the holdout set error, and the root mean squared error, which all aim at approximating the real generalization performance. In the experiments of this thesis these evaluation measures are used for benchmarking purposes, and in most cases they give good informations about the real performances of the learning algorithms. As another important part of machine learning, we gave a thorough introduction to feature processing methods, including feature selection and feature construction.

In Section 2.4 we formally defined optimization problems, which are of great interest, since in the remainder of this thesis we apply optimization algorithms to the supervised learning algorithms. Finally, in Section 2.5 we gave a thorough introduction to model-assisted optimization. We described the sequential parameter optimization (SPO), which is a method for tuning algorithms using arbitrary surrogate models. Although the user is free in choosing any surrogate model within SPO, we mainly focus on Kriging surrogate models, since these models generally give good solutions also for complex tuning tasks.