Universität Bremen

Automated optimization of complex processing pipelines for pySPACE

Automatische Optimierung komplexer Verarbeitungsketten für pySPACE

Torben Hansing

Master thesis

First reviewer: Prof. Dr. Frank Kirchner
Second reviewer: Dr. Thomas Röfer

September 2, 2016

Abstract

In machine learning the classification of data has been a complex task for years now. Machine learning algorithms typically have many different parameters (so called *hyper-parameters*) that need to be tuned by hand to achieve a good performance. The choice of the right algorithms to build the processing pipeline and the way to combine these is another challenging task. Because many different algorithms exist for different domains and tasks. Thus, the task of classifying data has been done by experts so far.

This thesis uses the knowledge of **algorithm experts** to dynamically find suitable processing pipelines and build the hyperparameter search domains for them. This new approach can then be used to automatically find the best processing pipeline structure and optimize the parameters needed to achieve state of the art performance. The optimization of hyperparameters is done using a third party library called *Hyperopt*.

To be able to use the **algorithm expert's** knowledge, this thesis defines a new way to express experts knowledge and make it usable by non-experts. It also implements a new approach to generate the search domain of the problem dynamically depending on the optimization task that should be executed. While this thesis focuses on the classification of data, this approach can easily be adapted for other data processing problems, like regression or other function approximation problems, as well. The main goal is to make the processing of complex data, using complex processing pipelines, available to non-experts.

Additionally, this thesis shows that the presented approach is able to optimize the structure and hyperparameters of processing pipelines at the same time. Thus, different experiments will cover the main aspects and show that state of the art processing performance can be achieved with it when used on data that require complex processing pipelines.

Contents

1. Introduction **1**

1.1. Structure of this work 3

2. Related Work **5**

2.1. Grid search 6

2.2. pySPACE 7

2.3. Sequential Model-Based Optimization 9

2.4. AutoWEKA 9

2.5. Hyperopt 10

2.6. Hyperopt-Sklearn 12

2.7. The P300-ERP 13

3. Realization **15**

3.1. Modeling 17

3.2. Implementation 18

3.3. Domain expert knowledge 43

4. Evaluation **45**

4.1. The evaluation scheme 47

4.2. Performance evaluation with the baseline structure 49

4.3. Convergence of the model 51

4.4. Complete optimization of a single processing pipeline 54

4.5. Selection of the classifier 57

4.6. Optimization of structure and hyperparameters 61

4.7. Full optimization without knowledge 64

4.8. Summary 68

5. Conclusion **69**

5.1. Future work 71

A. Technical details **75**

B. Logging configuration file example **85**

C. Task descriptions **89**

D. Optimization results **103**

Acronyms **113**

Glossary **114**

Bibliography **119**

List of figures **123**

1. Introduction

Using machine learning algorithms to classify data has always been a complex task as the algorithms have many parameters (so called hyperparameters). Optimizing these has been a task for experts for a long time. In the last few years there have been attempts to automatically optimize these parameters ([17] and [25]). The problem with these approaches is that they assume a single search domain to be given by some expert. This domain has to be defined by hand and contains both all possible algorithm combinations and hyperparameters of these algorithms. Additionally, they use simple reference data sets and simple processing pipelines to prove that their optimization works.
The approach presented in this thesis automatically optimizes the whole process. It generates all possible processing pipelines and optimizes the hyperparameters to find the optimal pipeline configuration. This implies the choice of the algorithms that will be used and how they need to be combined to build a proper processing pipeline. To optimize the hyperparameters of the algorithms, the parameter ranges need to be determined as well. The actual optimization of the hyperparameters is then done using the same library that Komer et al. ([17]) have used. This library is called Hyperopt and is developed by Bergstra et al. ([6]).
It will be based on a software called "*pySPACE*" [19] as it enables the usage of complex processing pipelines with complex data.
Complex data in this case are data like the *electroencephalogram (EEG)*, which are recordings of different brain signals from multiple electrodes. Thus, these data are high dimensional and time continuous.
As mentioned above for the optimization of the hyperparameters a library called Hyperopt [6] will be used as Komer et al. have shown that good results can be achieved with it [17]. Additionally, the description of the task to optimize fits well into the structure of pySPACE.
To automatically estimate the algorithms and their parameters, three main problems have to be addressed in different parts of the approach presented in this thesis:

1. **modeling**: During the creation of the model for the optimization problem the search domain needs to be held small enough to be able to efficiently search through it. Obviously it is not tractable to check every possible combination, as in pySPACE there are about 300 different algorithms [19].
   But the algorithms can be (and already are) grouped depending on their input and output data. For example that no algorithms using time continuous data can be used after an algorithm returning feature vectors.
   Additionally, the length of the whole pipeline or each processing unit can be lim-

ited. But just limiting the search domain is not enough as the ranges of the hyperparameters need to be estimated too. This increases the size of the search domain again. Therefore, good starting points or commonly used ranges for hyperparameters have to be used.

2. **implementation**: Even if the search domain is held small, depending on the data and the number of hyperparameters to optimize, this process can be complex and therefore some kind of parallelization has to be considered to achieve tractable run times. This problem has already been addressed in existing work [25] but was solved trivially by only parallelizing the search for optimal hyperparameters.
   As in this work the choice of the algorithms gets involved, this needs to be considered as well, as there might be dependencies between different configurations.

3. **domain expert knowledge**: The presented approach should be able to produce good results if used by a non-expert. But to achieve even better results or to find the solution faster it has to be possible to bias the search domain and hyperparameters. Making it possible for experts to give some kind of weight or belief to different algorithms or parameter ranges can speed up the process significantly as good estimators will be found faster.

1.1. Structure of this work

Chapter 2 will describe the technical state of the art this paper is based on. It also covers all details needed to understand this thesis. Chapter 3 describes the realization of this work, what problems have occurred, and how they have been solved. Additionally, this chapter covers a technical view on the realized system and describes it's main parts. In Chapter 4, the described realization will be evaluated using different experiments. Therefore, six different experiments have been performed. Each of it covering a different point of view on the solution. Last but not least, Chapter 5 will give a short summary, draw conclusions, and give some ideas of future work.

2. Related Work

In this chapter, the technical concepts and related work this thesis is based on will be described. First of all, this chapter will describe the most common used and most simple method to optimize hyperparameters. This method is called *grid search*. Additionally, this section will show, what problems exist with using it, and why there are other methods needed. Afterwards, the classification framework pySPACE will be described. This thesis is based on pySPACE and thus, a brief description and an overview over the framework are required. This chapter also compares two other optimization frameworks, *AutoWEKA* and *Hyperopt-Sklearn*, that are able to optimize the hyperparameters and structure of a processing pipeline. Both of them are based on an optimization approach called *sequential model-based optimization (Bayesian optimization) (SMBO)*. Because of this, this method is described as well. During the evaluation of this thesis, EEG data will be used. These are recordings of different brain measurements and include *event related potentials (ERPs)*. These potentials can be classified by a processing pipeline. Thus, one ERP, called the *P300* will be described as well.

2.1. Grid search

The grid search is the most commonly used technique for hyperparameter optimization [7]. Using this method, the user defines different possible values for the different hyperparameters. Afterwards, a search grid is build from these values by building the Cartesian product over all hyperparameters set by the user. For example, if the user sets two hyperparameters, $A = \{0, 1, 2\}$ and $B = \{3, 4, 5\}$, then the grid will have nine entries: Each value for A will be combined with each values for B. All these possibilities will then be tested by evaluating the processing pipeline with these hyperparameters and measuring the performance. After every parameter setting has been tested, the best found hyperparameter combination is returned or used for further processing.
This already shows three major problems with this approach. First of all, an expert is required to define the parameter combinations to search through. Thus, this process cannot be automatically done by a software as it does not know which combinations are possible or even useful. The second problem is, that this approach works only for discrete valued hyperparameters, because the user can just sample a few values. Last but not least, the grid search tests only the combinations given by the expert. If the optimal hyperparameters are not within the possible combinations, than they will not be found. Thus, the user needs to define many values for every hyperparameter or be a **domain expert** who knows ranges for the hyperparameters that work well. Because

of this problems, and because Bergstra and Bengio [7] have shown that randomly chosen parameters are more efficient than a grid search, this thesis does not use a grid search. This approach would require to infer the parameter ranges beforehand which is impossible as they depend on the input data set.

2.2. pySPACE

The first framework presented in this chapter is the classification and processing framework called pySPACE. This thesis is based on this framework and thus, it is described in detail here. The description is given first because the other frameworks will be compared to it to point out the problems of the when used with it.
pySPACE is a "**S**ignal **P**rocessing **A**nd **C**lassification **E**nvironment" developed by the *German Research Center for Artificial Intelligence (Robotics Innovation Center) (DFKI RIC)* and the robotics research group at the university of Bremen. It uses the programming language *Python* and allows the user a rapid specification, execution, and analysis of data. Therefore, this framework implements many different algorithms to pre-process, classify, and analyze the data. Another benefit of this framework is, that the different processing pipelines can be executed in parallel [19]. pySPACE currently features approximately 300 different algorithms and is one of the largest libraries for signal processing algorithms available. Many of these algorithms come from other processing libraries as well, like WEKA or *scikit-learn*. pySPACE is structured as a framework and one of it's goals is to be extendable. It is easy for developers to add new algorithms. Because of this, an optimization framework needs to be able to deal with this flexibility. In each installation of the framework other algorithms may exist and thus, the search domain for the processing pipelines is different.
During execution, the user can choose between different parallelization techniques like using multiple processor cores or even using a cluster. The configuration is done via *YAML* files that are easy readable by humans and computers as well [5]. This enables users without programming skills to use pySPACE. Last but not least, pySPACE implements a *grid search* mechanism to do basic hyperparameter optimization. This is required, because many algorithms, like *support vector machines (SVMs)*, define hyperparameters that highly depend on the input data. Thus, these parameters need to be retrained for every training data set [19]. Because of this, pySPACE enables the user to define a node that does a grid search during the training phase of a processing pipeline. The best hyperparameters found in this grid search will then be used during the predic-

tion phase.

Figure 2.1 shows the different algorithm types currently available in pySPACE. It fea-

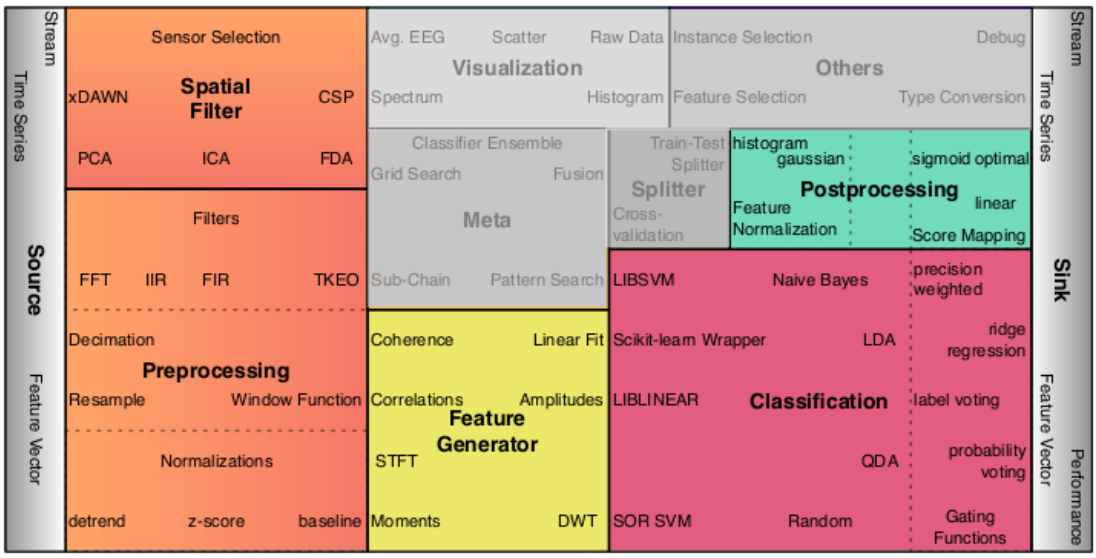


Figure 2.1.: Different algorithm types in pySPACE
Source: Modified version of the figure from the pySPACE paper [19].

tures 11 different categories of algorithms. For example a processing pipeline first needs to load the input data from either a hard disk or by reading them from some kind of sensors. This is done by using a `source` algorithm. After this, these data can be filtered, windowed, down sampled, and normalized using the different `preprocessing` and `spatial filtering` algorithms available. pySPACE is then able to generate features from these data using the different `feature generators`. These can then be post processed and classified using the `postprocessing` and `classification` algorithms. The predictions, the classifier has made, can be stored to disk or benchmarked using the different `sink` algorithms afterwards.

Figure 2.1 shows additional categories which are grayed out because they are either not important for the optimization process (*Visualization*, *Splitter*, *others*) or are used for ensemble learning which is not in the focus of this thesis. Because of this, these algorithm will not be regarded further in this thesis.

Additionally to the algorithms, used to create different processing pipelines, pySPACE implements an infrastructure to execute these pipelines. The execution of a processing pipeline on a dataset is called an *operation*. To enable the user to define more complex processing tasks, these operations can be *chained*.

This thesis focuses on executing and optimizing single processing pipelines and thus the chaining of operations is not used in this approach. Additionally, as described above, the focus of this thesis is the automatic generation and optimization of complex processing pipelines for classifying EEG and other complex data. Because of this, algorithms that are not usable in this scenario are filtered out from the search domain. More details on how and where this is done are described in Chapter 3.

2.3. Sequential Model-Based Optimization

The hyperparameter optimizers described in the following sections are based on a optimization technique called the sequential model-based optimization, also referenced as Bayesian optimization. This is a stochastic optimization technique that can be used to optimize hyperparameters. It is able to work with discrete and continuous hyperparameters and can exploit the benefits of hierarchically structured search domains. A hierarchical search domain is a domain with conditional dimensions. A conditional dimension thereby is a dimension of the search domain, that is being used depending on value of another dimension. For example, if one parameter, called **a**, exists which is selecting one of two different algorithms and each having a single hyperparameter (**b** and **c**) then the search domain has three dimensions. But as the values of **b** and **c** are only important depending on the value selected for **a** these two dimensions are conditional to the dimension of **a**.
SMBO methods build and train a model to infer the expected loss of a given hyperparameter setting. This model is trained by evaluating a given target function using different hyperparameter settings to obtain the corresponding losses [25]. In contrast to other optimization algorithms, like the *grid search* or a *random optimizer*, SMBO methods invest more effort in training the model and finding the best hyperparameter setting that should be tested next to reduce the overall number of evaluations needed to optimize the target function [25].

2.4. AutoWEKA

AutoWEKA is an automated optimization tool for the classification library *WEKA* which is written in Java. It implements the selection of the optimal learning algorithm and optimizing it's hyperparameters to enable non-expert users to use the *WEKA* [14] library and achieve state of the art performances.

The *WEKA* library is a classification framework written in Java and distributed under a open source license. It includes a *graphical user interface (GUI)*, different preprocessing, classification, and regression algorithms [14]. AutoWEKA defines the problem of selecting the best learning algorithm and the best hyperparameters as a single, hierarchical hyperparameter optimization by defining the choice of the learning algorithms itself as a hyperparameter [25]. Because all learning algorithms and their hyperparameters are included in a single search domain, this results in a large space. They are able to manage this search domain by structuring it. Thus, they included two base parameters, one to choose from different classes of classifiers (Base- or Meta- / Ensemble-Classifiers) implemented in *WEKA* and one to enable or disable the feature selection and evaluation algorithms. Depending on the values of these two parameters, new parameters are added to the search domain until one or more classifiers and optionally one of the possible feature selectors or evaluation algorithms are selected [25]. The hyperparameters of the selected model then are added to the search domain for the hyperparameter optimization.
For the 39 different classification, 3 feature search and 8 feature evaluation methods implemented in *WEKA* this results in a tree with four layers and 786 parameters in total [25]. Because AutoWEKA uses a SMBO technique to optimize this search domain, every parameter itself is defined as a distribution over it's possible values or ranges. This whole space is handcrafted and has to be extended by an algorithm expert if new algorithms should be included.
Because complex data, likeEEG-Data, require a lot of pre- and postprocessing the processing pipeline gets complex quite fast. This results in a large search domain that is not manageable by hand as each possible processing pipeline has to be made available inside the search space. This is not tractable by humans. Additionally, AutoWEKA does not optimize the different processing pipelines in parallel. As the search domain is large for a complex dataset, this results in long running processes, that might not use all resources available. Also, as this library is customized for *WEKA* and written in Java, it is not suitable for our goal to optimize processing pipelines using pySPACE.

2.5. Hyperopt

Hyperopt is a library used for optimizing search domains that contain real-valued, discrete, and conditional dimensions [6]. It is implemented using the programming language Python and uses a SMBO algorithm called *tree-structured parzen estimator (TPE)* to

find the optimal parameters for a given target function [6]. TPE uses two estimators to sample new hyperparameters for a given loss. The first estimator is used to sample hyperparameter settings for losses below the best found loss and the other one is used for hyperparameters above this loss. During the optimization process, these two estimators get trained using the evaluations done. Thus, with each evaluation done, the predictions of the estimators for good hyperparameter settings get better [8].
As SMBO algorithms are capable of handling discrete and continuous parameters in hierarchical search domains using different distributions, Hyperopt supports a variety of different parameter distributions the user can choose from.

- The first distribution is a simple choice parameter, which will randomly choose one of the given values
- One variant of this choice parameter is the probability choice. It will choose the given values according to the given probabilities
- Next, a uniform distribution exists. This parameter distribution will uniformly sample a continuous value between two boundaries
- The log-uniform distribution is a variant of this parameter. It samples from the exponential form of the uniform distribution
- The normal distribution will sample a continuous value from a normal distribution
- The log-normal distribution variant samples a continuous value from exponential form of the normal distribution
- Last but not least, the random integer distribution randomly selects an integer from a given range

Additionally, Hyperopt defines quantified variants of the uniform, log-uniform, normal and log-normal distributions as variants for discrete distributions. These distributions sample according to the non-quantified distributions and afterwards round the values to a multiple of the quantifier. These definitions result in a total of 11 different distributions the user can use to define the distribution model of the SMBO algorithm. Because they are a central part in the approach of this thesis, they are further described in Section 3.2.1.
During the optimization process, the domain, given by the user, is used to create an initial distribution model for the SMBO algorithm. From this model the hyperparameter

settings that will be tested get sampled [6].
In contrast to AutoWEKA, Hyperopt supports the parallel execution of multiple evaluations of a single target function. This enables the usage of large clusters for optimizing the hyperparameters of a single processing pipeline. Additionally, as it is implemented using Python, as well as pySPACE, this library seems useful for the problem this thesis tries to solve.

2.6. Hyperopt-Sklearn

Hyperopt-Sklearn is another automated optimization tool for data classification. Like AutoWEKA it treats the problem of finding the optimal processing pipeline structure and optimizing it's hyperparameters as a single optimization problem. It is based on the classification framework scikit-learn and uses Hyperopt to optimize the hyperparameters. Hyperopt-Sklearn is implemented using the programming language Python because the used frameworks are already implemented in this language.
In contrast to AutoWEKA, this framework is able to evaluate multiple hyperparameter settings in parallel as provided by Hyperopt. This enables the usage of the software on a large cluster to speed up the optimization process.
Hyperopt-Sklearn creates processing pipelines which consist of a classifier and zero or one preprocessing steps. But as scikit-learn is a large framework consisting of many learning algorithms, this framework concentrated on a small subset consisting of 5 preprocessing algorithms and 6 classifiers [17]. Like in AutoWEKA, this search domain has been written by the authors and resulted in a domain having 65 hyperparameters in total. Additionally, the authors have defined a blacklist for algorithm combinations that do not work to speed up the optimization process. Again, as the SMBO algorithm provided by Hyperopt support a hierarchical search domain, additional hyperparameters have been defined to describe the structure of the pipeline and activating the different hyperparameters needed for the chosen algorithms. The resulting software has then been tested using three benchmark data sets from the domains of optical character recognition, text classification and object recognition [17].
This framework is written in Python and is using the scikit-learn framework which is included in pySPACE. Additionally, it features parallel evaluation of processing pipelines. But like in AutoWEKA, the search domain is still handwritten. Thus, only a small number of algorithms featuring only small processing pipelines has been included. For complex data like EEG-Data this is not enough and a more general approach is required.

2.7. The P300-ERP

A EEG is a measurement of the brain activity by different electrodes placed at specific spots on the head of a subject. The exact placement depends on the experiment and the desired measurements. Different setups for each type of experiment exist. But because the type of EEG-Data and how they have been collected does not matter for the optimization process, these setups will not be described further.
The electrodes, which are placed on the subjects head, measure electrical potentials. By paring each two of these electrodes and measuring the difference of the potentials a time continuous series of potential differences can be created for each pair [24].
The P300-potential is an ERP that can be measured as a positive deflection in the EEG approximately 300ms after a relevant stimulus has been given [15]. This ERP can be used to build a *Brain-computer-interface (BCI)* by measuring the brain activity using EEG and detecting the P300-ERP using a processing pipeline to classify whether a potential has been deflected or not. This classification data can then be analyzed depending on the task of the BCI and the corresponding actions issued.
This thesis uses EEG-data that contain P300-ERPs and automatically creates and optimizes processing pipelines that are able to classify the occurrence of a P300-ERP and thus, are usable in a BCI.

3. Realization

This Chapter will describe the realization of the software created during this thesis. This thesis implements a new approach to automatically create and optimize processing pipelines to classify complex datasets like EEG data. To solve this problem, this solution uses the knowledge of two different experts to determine all possible processing pipelines and then optimize the hyperparameters of them. The first expert who's knowledge is used is an **algorithm expert**. This expert is a person who is able to implement a specific processing algorithm and has knowledge about the internal behavior of the different parameters. Thus, she or he is able to define which types of input data are required and which output data type will be produced for these. Additionally, this person is able to infer from the internal behavior of the algorithm which hyperparameters should be optimized and which distributions they might follow. In summary the **algorithm expert's** knowledge is used to define the input and output data types and a good starting point for the hyperparameter optimization.
The second expert is a **domain expert**. This person is a specialist in the domain the input data came from (i.e. a domain expert for optical character recognition or EEG). He or she knows about algorithms commonly used in the given domain and might even know a part of the parameters settings that will work well for the given input data. Because of this knowledge, the **domain expert** is able to define algorithms that should or should not be used and domain specific hyperparameter settings. This knowledge can be used by the software to redefine the search domain and the initial distribution model. This will reduce the size of the search domain and speed up the optimization process.
As one goal of this thesis is to create a software solution usable by non-experts, both types of experts knowledge are optional and don't need to be present. But as the knowledge of the **algorithm expert** is static and independent of the different domains, this knowledge can be acquired over time and will stay in the software.
In the following Sections, the different mechanisms created to express the different experts knowledge and how they are used during the optimization will be described. The structure of this Chapter is inspired by the three main problems described in the introduction. At first, the *modeling* of the search domain is described. This section will also cover the generation of the processing pipelines. Next, the implementation of the software will be described and last but not least the abilities of the **domain expert** to express his or her knowledge will be described as well.

3.1. Modeling

In contrast to AutoWEKA [25] and Hyperopt-Sklearn [17], this thesis does not try to define the process of finding the best processing pipeline structure and the optimal hyperparameters as a single optimization problem. Instead, it explicitly treats them separately and tries to keep the search domain as small as possible. By the time of writing this thesis, pySPACE features about 300 different algorithms with up to 39 and an average of 9 hyperparameters this would result in a search domain of about 10^{22} combinations in average. A search domain of that size requires a large exploration phase to visit all parts of the space and therefore, many evaluations have to be done. Because these evaluations take a long time compared to the training of the model and sampling new hyperparameter settings the main goal was to reduce the number of evaluations done. Because of this, a separate search domain is created for every processing pipeline that has been found. This reduces the size of the model that needs to be trained to the hyperparameters of one specific processing pipeline and creates multiple models that get trained.

As most of the processing pipelines, that in theory are able to process the input data, are expected to fail processing the data, a more fine grained control over the optimization process is possible using this approach. If a pipeline fails to process the input dataset multiple times, it can be assumed that it is not able to process them at all and the optimization process can be stopped. This prunes the search domain as a large part (representing the whole processing pipeline with all of its hyperparameter combinations) does not need to be explored.

Additionally, the search domain should be defined as small as possible beforehand. This is done by focusing on the processing pipelines that, at least in theory, are capable to process the given input data. Thus, the first part of the **algorithm expert's** knowledge is used. By defining the *input* and corresponding *output* data types of the different algorithms they can be checked during the generation phase of the processing pipelines. By doing this, only these processing pipelines get generated and tested, in which the data types of the different algorithms match. If, for example, an algorithm `A` requires time continuous input data and produces feature vectors and a second algorithm `B` takes prediction vectors as input and outputs prediction vectors again, these two algorithms cannot be combined, because the input and output data types do not match. But if we add a third algorithm `C` taking feature vectors and producing prediction vectors, than we

might get a valid processing pipeline:

$$A \rightarrow C \rightarrow B$$

In this example algorithm `B` might be appended multiple times to itself because the input data type equals the output data type. Because of this, two additional constraints have been made. First of all only one algorithm of each pySPACE type (Section 2.2) is allowed in a single processing pipeline. This constraint does make sense as most of the algorithms either cannot be used multiple times (i.e. multiple classifiers) or do not have any effect (i.e. multiple normalization). The second constraint is given by the user because a maximal length of pipelines that get generated needs to be defined. If the user defines that the maximal pipeline length is 5 only pipelines up to this length get tested. Pipelines that are shorter than the given maximum always get tested, as simpler (i.e. shorter) processing pipelines with the same performance are always considered better.

3.2. Implementation

The software implemented in this thesis is written in Python. In the following Sections the details of this implementation will be described. This includes describing the structure of the software, how the parts are implemented and how they are connected to each other.
This Chapter is split in two parts. First an overview of the software is given, describing its structure and how the main components interact with each other. The second part describes the components in detail and discusses the problems that have been dealt with during the implementation and how they have been solved.
Figure 3.1 gives an overview of the structure of the whole software and it's main parts. Because the structure of the software with all its details and classes is to complicated to visualize, only the three main packages of the software are shown with their components. The `pySPACEOptimizer core` package defines the user interface and implements main functionality used by the other packages. Thus, the `main` module is used as the entry point of the software and defines the user interface. Depending on the user input this module instantiates an optimization task, an optimizer, and starts the optimization process.
In the `pySPACEOptimizer framework`, base classes for implementing new optimization

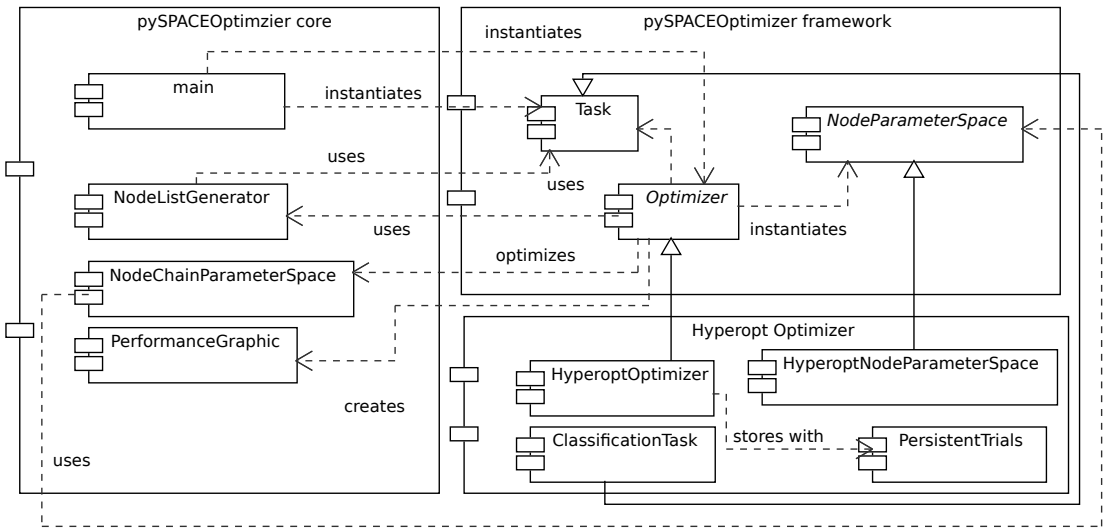


Figure 3.1.: Structure of pySPACEOptimizer

tasks and optimizers are implemented. The base class of the optimizer is used to collect performance data, keep track of the found optimum and visualize the results.
These classes are extended in the `Hyperopt Optimizer` package. It implements a custom `Classification Task` and an optimizer using the Hyperopt library. This optimizer does the execution of the optimization task. As this thesis focuses on the classification of data, the implemented classification task is used for the optimization tasks to define which type of algorithms are useful for this task.
Additionally to the packages shown, an extension to pySPACE has been made to enable **algorithm experts** to express their knowledge. This extension will be described first.

3.2.1. pySPACE extensions

To make the **algorithm expert's** knowledge about the internal behavior of the different hyperparameters usable, a new extension to pySPACE had to be created. As the algorithms are defined inside pySPACE and an **algorithm expert** should be able to implement a new algorithm just using pySPACE, this extension has been included in the framework itself. It creates a new semantic to enable the expert to express his knowledge about the parameters by simply decorating the algorithms.
Therefore, the different hyperparameter distributions already defined in Hyperopt (Section 2.5) have been reused in this extension. For each of this distributions a so-called *decorator* has been created. A *decorator* is a Python function that can be used to dy-

```
@LogUniformParameter("complexity", min_value=1e-6, max_value=1e3)
@PChoiceParameter("kernel_type", choices={"LINEAR": 0.5, "RBF": 0.25,
                                          "POLY": 0.125, "SIGMOID":
↪  0.125})
@QNormalParameter("offset", mu=0, sigma=1, q=1)
@UniformParameter("nu", min_value=0, max_value=1)
@LogNormalParameter("epsilon", shape=0.1 / 2, scale=0.1)
@QUniformParameter("max_time", min_value=0, max_value=3600, q=1)
@NormalParameter("ratio", mu=0.5, sigma=0.5 / 2)
class RegularizedClassifierBase(BaseNode):
```

Figure 3.2.: A decorated algorithm example

namically extend a class or function during the runtime of the program.
An example of a decorated algorithm is shown in Figure 3.2. The lines starting with an "@" are decorators. In this example a class called `RegularizedClassifierBase` is decorated using multiple different distributions for the different hyperparameters. In this section, all different distributions available in the pySPACE extension will be described and visualized.

Choice The first distribution that will be described in this section is a simple choice. It is defined by a set of possible values. Each of these values will be selected with equal probability. Because of this, a value x will be sampled as follows:

$$x = \text{random}(\text{choices})$$

The following formula describes the probability function of this distribution:

$$P(X = x) = \frac{1}{|\text{choices}|}$$

Each value will be sampled with an equal probability. The probability thereby depends on the length of list of input values.

Boolean choice A Boolean choice is a shortcut for a choice with just two values (`True` and `False`). A value x will be sampled as follows:

$$x = \text{random}([\text{True}, \text{False}])$$

As result, the probability function for this type of distribution defined as $P(X = x) = \frac{1}{2}$.

Probabilistic choice A probabilistic choice is like a normal choice with the difference that, instead of all choices being sampled with equal probability, the user is able to define the probability of each choice. A value x will be sampled from the following function:

$$x = \text{random}_p(\text{values})$$

With *values* being the list of possible values for the choice and random_p being a function that randomly returns one of the given values according to the probability p defined for it.

The probability function for this distribution if defined as a look-up:

$$P(X = x) = p(x)$$

With p being a function, that returns the probability assigned to each value.
This distribution is useful in cases where there are multiple values available but in most cases a single one is sufficient. One example for this might be the "kernel" hyperparameter of a SVM. The kernel defines the similarity measurement method for the SVM. Because many different measurement methods are available, many different kernels for a SVM exist. Depending on the domain, different kernels are the best, but in most of the cases, a linear kernel (i.e. measuring the euclidean distance between two points) is sufficient. Thus, the linear kernel will get the highest probability and all other choices share the rest.

Uniform distribution A uniformly distributed variable is sampled between two boundaries (a and b) with equal probability for each value. A value x will be sampled from

this distribution using the following function:

$$x = \text{uniform}(a, b)$$

uniform in this case defines a function that samples a value from the interval $[a, b]$ with a *probability density function (PDF)* defined by the following formula:

$$f(x) = \begin{cases} \frac{1}{b-a} & \text{if } a \leq x \leq b \\ 0 & \text{else} \end{cases}$$

Figure 3.3 shows a plot of a uniform distribution with a minimum value of 2 and a maximum value of 4. An example of a hyperparameter with this distribution is the nu parameter that exist in a special SVM variant, called the *nu-SVM*. This SVM variant has replaced the regularization parameter *C* with a new parameter *nu*. In contrast to the unbound *C*, this new parameter is bound between 0 and 1 and corresponds to the percentage of samples, that will be used as support vectors for the SVM. Because this parameter is bound between 0 and 1, and because it is continuous, a uniform distribution can be used for this parameter to describe it's parameter space.

Quantified uniform distribution A quantified uniformly distributed variable is sampled, uniformly between two boundaries a and b. But afterwards, the sampled value get's rounded to a multiple of the quantifier. Thus, a value x will be sampled using the

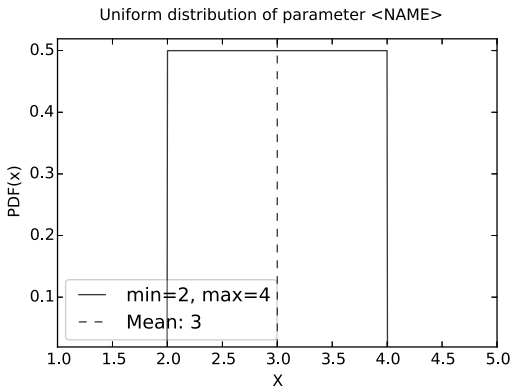


Figure 3.3.: Plot of the PDF of a uniform distribution with min$=2$ and max$=4$

following function:

$$x = \text{round}\left(\frac{\text{uniform}(a,b)}{q}\right) \times q$$

With a being the minimum and b being the maximum limits. The PDF for this distribution is defined by the following formula:

$$f(x) = \begin{cases} \frac{1}{b-a} & \text{if } a \leq x \leq b \text{ and } x \bmod q \equiv 0 \\ 0 & \text{else} \end{cases}$$

A corresponding plot with a minimum value of 2 and a maximum value of 4 is shown in Figure 3.4. The quantifier has been chosen as 0.5. As it can be seen in Figure 3.4 the plot is not a static function but a discrete distribution with points on the multiples of the quantifier. As the quantifier in this example has been chosen to be 0.5, all possible values that can get sampled are 2, 2.5, 3, 3.5, and 4. This distribution is useful for hyperparameters that are discrete and equally distributed between two bounds. One example of a hyperparameter following this distribution is the maximal amount of time a SVM might be given in pySPACE to be constructed. This is a very special hyperparameter that controls the time (in seconds) a classifier is given to be trained. Smaller times are, of course, better, but we cannot assure this as a **algorithm expert**. Thus, a quantified uniform distribution has been chosen for this parameter. On the other hand, a **domain expert** might use this type of distribution, to define that near values do not have a large effects on the performance and do not need to be tested.

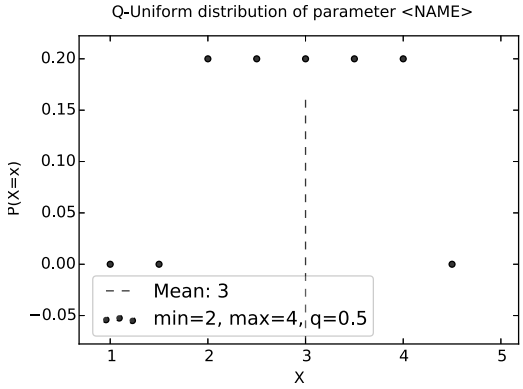


Figure 3.4.: Plot of a quantified uniform distribution with min$=2$, max$=4$, and q$=0.5$

Logarithmic uniform distribution A variable is logarithmic uniformly (*Log-uniformly*) distributed if the logarithm of this variable is uniformly distributed. A value x of this distribution will be sampled from a uniform distribution with logarithmized limits and afterwards exponentiated using the base of the logarithm:

$$x = \exp\left(\text{uniform}(\ln a, \ln b)\right)$$

With a and b being the lower and upper bounds of the distribution.
The following formula describes the PDF of this distribution in a mathematical way:

$$f(x) = \begin{cases} \frac{1}{x(\ln b - \ln a)} & \text{if } a \leq v \leq b \\ 0 & \text{else} \end{cases}$$

This distribution is useful when the possible maximum value is much larger than the minimum value [1]. Because the divisor of the term increases with a higher x, this distribution gives higher densities to lower values and decreases as x increases. This behavior is also shown in Figure 3.5 which shows a plot of a log-uniform distributed parameter with 2 as the minimal and 200 as the maximal borders. As already said, this distribution will sample lower values more often than higher ones. Thus, it can be used for parameters, that have a large range of possible values, but for which, in most cases, lower values are better. One example for such hyperparameter might be the regularization parameter (`C`) of a SVM. This parameter regularizes the influence of a

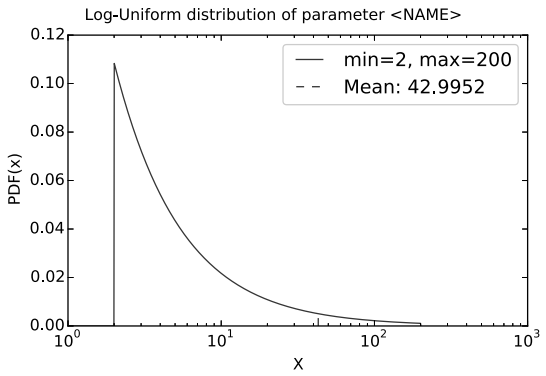


Figure 3.5.: Plot of a log-uniform distribution with min$=2$ and max$=200$

missclassification of a sample during the training of the SVM. Larger values will reduce the missclassification rate of the SVM but also might reduce the ability to use the SVM on other data, as it might be overfitted to the training data. Thus, a value as small as possible and as large as required is needed. As shown in Figure 3.2, a large range of values (10^{-6} to 10^{3}) can be defined for this parameter. Because of this, a log-uniform distribution for this parameter should be chosen.

Quantified logarithmic uniform distribution The quantified logarithmic uniform (*q-log-uniform*) distribution is the quantified version of the *log-uniform* distribution. This distribution can be used if the possible values are a large range and it should be sampled using a fixed step size. Therefore, a value x will be sampled using the following function:

$$x = \text{round}\left(\frac{\exp\left(\text{uniform}(\ln a, \ln b)\right)}{q}\right) \times q$$

The PDF of this distribution is defined by the following formula:

$$f(x) = \begin{cases} \frac{1}{x(\ln b - \ln a)} & \text{if } a \leq x \leq b \text{ and } x \bmod q \equiv 0 \\ 0 & \text{else} \end{cases}$$

Figure 3.6 shows a plot of the PDF of this distribution using 2 as the lower bound and 4 as the upper. 0.5 has been set as the value of the quantifier. This distribution is used best for hyperparameters with discrete values that can be in a large range of possible values. Like the *log-uniform* distribution, this distribution gives higher probabilities to

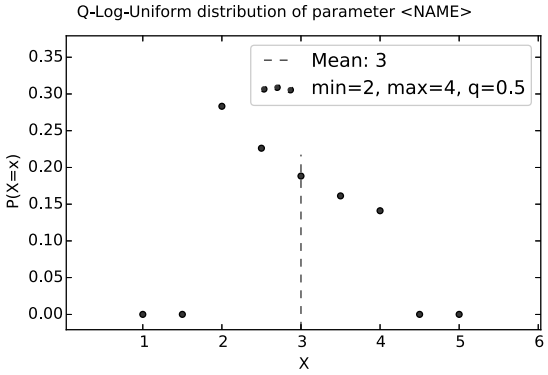


Figure 3.6.: Plot of a quantified log-uniform distribution with min$=2$, max$=4$, and q$=0.5$

lower values. Thus, a discrete parameter, with large bounds that should be as small as possible can be decorated using this distribution. An example of such hyperparameter is the maximal number of iterations a SVM can do. The lower bound for this distribution is 1 as at least one iteration has to be done. Theoretically, the upper bound is infinite. Practically, this would result in an infinite runtime of the SVM and thus, it would not terminate. Because of this, the upper bound of the hyperparameter is bound as well. As only whole iterations are possible, this parameter is also bound to integer values. Thus, a quantified logarithmic uniform distribution can be used to express the parameter space.

Normal distribution If a random variable is normally (or Gaussian) distributed, it can be described using a mean μ and a standard deviation σ. These two parameters define the expected mean value (μ) as the value with the highest probability of being chosen and how likely it is that other values are chosen as well (σ). Using this definition a value x is sampled using the following function:

$$x = \mathsf{normal}(\mu, \sigma)$$

normal is defined as a function that returns an unbound real value according to the following PDF:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}}\mathsf{exp}\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

Figure 3.7 shows the plot of a normal distributed parameter with a mean of 0 and a standard deviation of 1. This distribution is also known as the standardized normal distribution. This type of distribution can use used for hyperparameters that are unbound and for which an **algorithm expert** can define a mean value. The values will then be searched near this value. An example of such hyperparameter is the *ratio* of a classifier as shown in Figure 3.2. The *ratio* is used by the classifier to assign weights to the different classes for the classification. This weighting might be required because one class is much more present than the other. To express, that both classes have the same impact on the performance, the *ratio* can be changed. A *ratio* of 0.5 expresses, that both class labels are equally important. Thus, this parameter is actually bound between 0 and 1, but with values around 0.5 being much more likely than the borders. Because

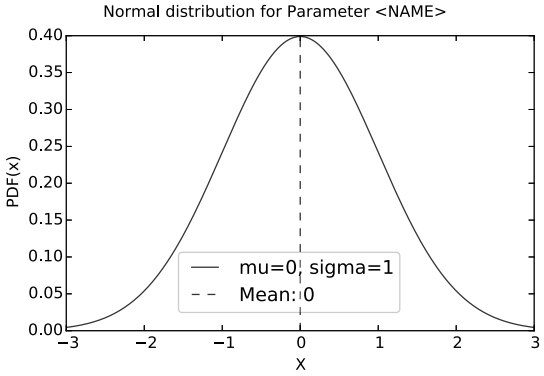


Figure 3.7.: Plot of a normal distribution with $\mu = 0$ and $\sigma = 1$

of this, a normal distribution has been chosen, such that 95% of the samples will be sampled between 0 and 1. The remaining 5% will be handled by the classifier by setting them to either 0 or 1.

Quantified normal distribution The quantified normal distribution is a discrete version of the normal distribution in which a value x will be sampled using a normal distribution defined by μ and σ and afterwards is rounded to a multiple of the quantifier q. Therefore, a value x of a *q-normal* distribution will be sampled using the following function:

$$x = \text{round}\left(\frac{\text{normal}(\mu, \sigma)}{q}\right) \times q$$

The corresponding PDF of this distribution is defined as follows:

$$f(x) = \begin{cases} \frac{1}{\sqrt{2\pi\sigma^2}}\exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) & \text{if } x \bmod q \equiv 0 \\ 0 & \text{else} \end{cases}$$

Figure 3.8 shows a plot of a quantified normal distribution with a mean of 0, a standard deviation of 1, and the quantifier being set to 0.5. Like the *q-uniform* distribution, this distribution can be used for discrete hyperparameters that follow the *normal distribution*. Because this is a very special distribution, no hyperparameter has been decorated using this distribution by now. **Domain experts** might use this type of distribution to define a parameter, which is originally using a *normal* distribution, if they know that values near to each other do not have great impact on the performance. If they define a

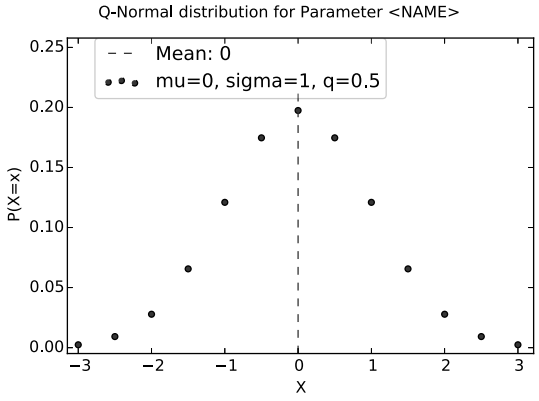


Figure 3.8.: Plot of a quantified normal distribution with $\mu = 0$, $\sigma = 1$, and $q = 0.5$

hyperparameter using this distribution, they reduce the size of the search domain and speed up the optimization process.

Logarithmic normal distribution A random variable is *Log-normally* distributed if the logarithm of the variable is normally distributed [4]. The *Log-normal* distribution has two parameters like the normal distribution, but in contrast to the normal distribution these parameters do not correspond directly to the expectation value and the variance. Instead, they are called *scale* and *shape*. A relationship between these two parameters and the mean and standard deviation of a normal distribution can be defined as $\mu = \ln \text{scale}$ and $\sigma = \text{shape}$ [11]. As the names denote, the *scale* parameter controls the width of the distribution and the *shape* parameter controls the shape. Like the *Log-uniform distribution* this distribution is bound to positive values, because the logarithm of a negative number is undefined.

A value x of a *Log-normally* distributed parameter will be sampled using the following function:

$$x = \exp\left(\text{normal}\left(\ln \text{scale}, \text{shape}\right)\right)$$

The PDF of this distribution for $x \geq 0$ is given by the following formula:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2 x^2}} \exp\left(-\frac{(\ln x - \mu)^2}{2\sigma^2}\right)$$

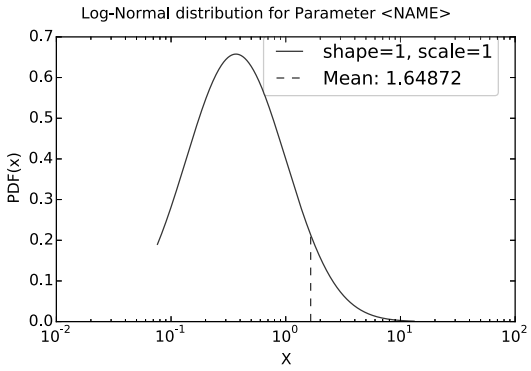


Figure 3.9.: Plot of a log-normal distribution with shape $= 1$ and scale=1

The plot in Figure 3.9 shows a *log-normally* distributed parameter with scale $= 1$ and shape $= 1$ as parameters. This distribution is best for hyperparameters that have a lower bound of 0 and that are unbound in the upper end. An example parameter for this distribution would be the *epsilon* hyperparameters of a SVM. This parameter controls the convergence of the model learned by the SVM. It controls at which measure the SVM considers a model to be "good enough" to explain the input data. A smaller *epsilon*, thus, results in a model more fitted to the data. The lower bound of this parameter is 0. The upper bound depends on the input data and is unbound. Because of this, this parameter can be described using a *log-normal* distribution.

Quantified logarithmic normal distribution The *quantified log-normal distribution* is a discrete version of the *log-normal distribution* defined above. A value x of this distribution is sampled using the *log-normal distribution* and afterwards rounded using a given quantifier. Values of this distribution get sampled using the following function:

$$x = \mathsf{round}\left(\frac{\exp\left(\mathsf{normal}(\ln \mathsf{scale}, \mathsf{shape})\right)}{q}\right) \times q$$

And the PDF of this distribution is defined by the following formula:

$$f(x) = \begin{cases} \frac{1}{\sqrt{2\pi\sigma^2 x^2}} \exp\left(-\frac{(\ln x - \mu)^2}{2\sigma^2}\right) & \text{if } x \bmod q \equiv 0 \\ 0 & \text{else} \end{cases}$$

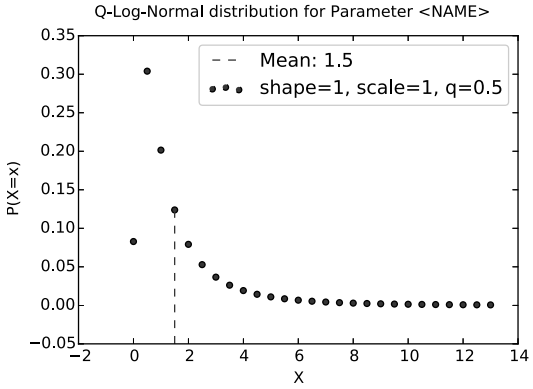


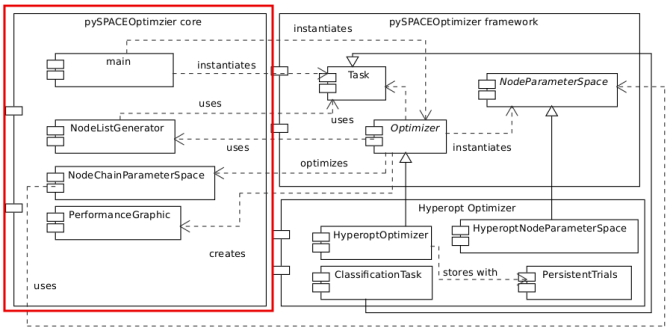
Figure 3.10.: Plot of a quantified log-normal distribution with scale $= 1$, shape $= 1$, and $q = 0.5$

Last but not least Figure 3.10 shows the plot of a parameter using the *quantified log-normal distribution* with scale $= 1$, shape $= 1$, and $q = 0.5$. This like the *q-normal* distribution, this one can be used for hyperparameters that follow a *log-normal* distribution but should be searched using a fixed step size. Like the *q-normal* distribution, this is a very special distribution and it has not yet been used for any hyperparameters. Again, a **domain expert** can use this distribution to define a search step size for hyperparameters that follow the *log-normal* distribution.

These eleven different distributions for hyperparameters enable **algorithm experts** to represent the most important parameter distributions for the different algorithms. The implementation as decorators enable experts to express their knowledge of the algorithms internals in a human and machine readable semantic.

3.2.2. pySPACEOptimizer core

The *pySPACEOptimizer core* package implements all classes and functions that are required to run the optimization software and that are task and optimizer independent. Because of this, these are required only once and do not need to be implemented again when including a new type of task or a new optimization algorithm. This package includes the user interface of the soft-



ware, the generator for the possible pipelines, and a class to generate a performance graphic during the optimization process. These parts will be described in detail in the following paragraphs.

main This module of the software implements the user interface. It is used by the user to start the software, pass the optimization task that should be performed, the path to store the results in, and which pySPACE backend will be used during the optimization process. This software is a console script and does not include a GUI. Thus, the user interface is defined by the accepted command line options. The software can be started using three different modes. First of all, when an optimization task should be executed, the following arguments are accepted:

- `-h, --help` This arguments prints a short help description of the software and which arguments are accepted.
- `-c, --config` This argument defines the name of the pySPACE configuration file that should be used during optimization. Depending on the domain, different pySPACE configuration files may be used to (globally) exclude algorithms or to define different parameters for the pySPACE logging and backends.
- `-t, --task` This argument defines the path to the description of the optimization task that should be performed. This file needs to be written in the YAML format. The exact representation of an optimization task is described in Section 3.2.3.
- `-b, --backend` This argument specifies the backend to use during the optimization process to execute the evaluations of the processing pipelines. Currently pySPACE implements four different backends: "serial" for serial, single core execution, "mcore" for parallel execution using multiple *central processing unit (CPU)* cores, "mpi" for using a cluster with a scheduler called "MPI", and "loadl" for a cluster using a "LoadLeveler" scheduler.
- `-r, --result` This argument defines a path to a directory to store the results of the optimization to. This will store the output of the best processing pipeline that has been found, the log files, the performance graphic, and the output of the pySPACE framework. If using a cluster, this directory needs to be accessible from all nodes. If this argument is not given, the default is to store the results in the pySPACE operation results storage in a sub-directory with the name of the optimization task.

The other two modes can be used during definition of the optimization task. In this cases, the software can either list all optimization algorithms available or all types of optimization tasks it can process. Besides the `--config` argument either `--list-optimizer` or `--list-tasks` needs to be given.

NodeListGenerator One of the main differences of the approach presented in this thesis when compared to existing methods like *AutoWEKA* [25] or *Hyperopt-Sklearn* [17] is the dynamic generation of the search domain. In this thesis, this domain is not handcrafted by human experts but will be generated depending on the input data and the optimization task that should be done. This requires a method to generate all processing pipelines able to process a given input dataset. The `NodeListGenerator` class implements the logic required to generate these pipelines. To search efficiently for all pipelines it requires some additional knowledge from the **algorithm expert** and a few constraints. The following paragraph will describe the logic in detail and present the reasons for the constraints, used to prune the search space of processing pipelines.
First of all, this thesis uses multiple independent search spaces to do an optimization task. On the one hand, a search space for the *processing pipelines* is defined by all algorithms available for a specific optimization task. Normally, these are all algorithms available in the pySPACE installation. But a **domain expert** might exclude algorithms to reduce the size of the search domain. On the other hand, a search domain exists for the hyperparameters of every processing pipeline found. These search domains are highly dynamic because they are defined during the generation of the processing pipelines. The `NodeListGenerator` uses the first domain and searches in it for pipelines that, at least in theory, are able to process a given input dataset. The technical details of the implementation are described in Appendix A.1.
To find a valid processing pipeline, this class uses the input- and output typed of the algorithms to combine them to a valid processing pipeline as described in Section 3.1. This search is done using an exhausting *depth-first* search. It starts by selecting a source node that is capable of reading the given input dataset. Afterwards, the output data type of the source node is used to search for algorithms that take this data type as input. This procedure is continued until either a valid processing pipeline is found, or the maximal pipeline length is reached. After this, *backtracking* is used to search through the whole search space.
If no additional domain knowledge is given by the user, this is the procedure used to generate all possible processing pipelines. Even with these performance optimization this

can take a long time because pySPACE implements ≈ 300 algorithms and the search space might be large. Because of this, the user is capable of defining a list of algorithms that are required in any valid pipeline. If such list is defined, the `NodeListGenerator` class checks every pipeline found, if all required algorithms are contained as well. If not, it is not considered a valid pipeline. Additionally, this list can be used to speed up the generation process as well. If the user defines a set of required algorithms, the number of algorithms that can be chosen by the algorithm can be calculated. Because the maximal number of algorithms in a processing pipeline is known and the number of required algorithms is known as well, the difference of these is the number of algorithms, that can be chosen automatically. During the generation this number can be used to check if a valid processing pipeline is possible or not and thus reduce the search space. If the number of algorithms not in the required nodes is larger than the number of algorithms that can be chosen, no valid pipeline is possible. Thus, the backtracking can start from this place and reduces the size of the search domain.

NodeChainParameterSpace This class represents a single, valid processing pipeline with its parameter space. The parameter space of a processing pipeline is defined as all values of all hyperparameters of all algorithms contained in the processing pipeline. As this might include real valued hyperparameters, a feasible representation of this space is required. Because of this, the distributions defined in Section 3.2.1 are used to represent the parameter space. This class collects the spaces of each algorithm contained and joins them to a single set of hyperparameters. This set can then be used by an optimizer to sample a hyperparameter setting for this pipeline. Because multiple algorithms can have the same hyperparameter names, all parameters are prefixed with the name of the algorithm to ensure uniqueness.
Additionally to representing a parameter space, this class can be used by the optimizer to execute the processing pipeline using pySPACE when given a concrete hyperparameter setting.

PerformanceGraphic The optimization of a complex processing pipeline can be a long running task that requires many evaluations. During this process, the user needs to know the performance of the different processing pipelines already tested. This enables the user to check whether a processing pipeline with a suitable performance has already been found or not. If a processing pipeline with a good performance has been found, the user can either stop the optimization process, or at least start to use the found pipeline

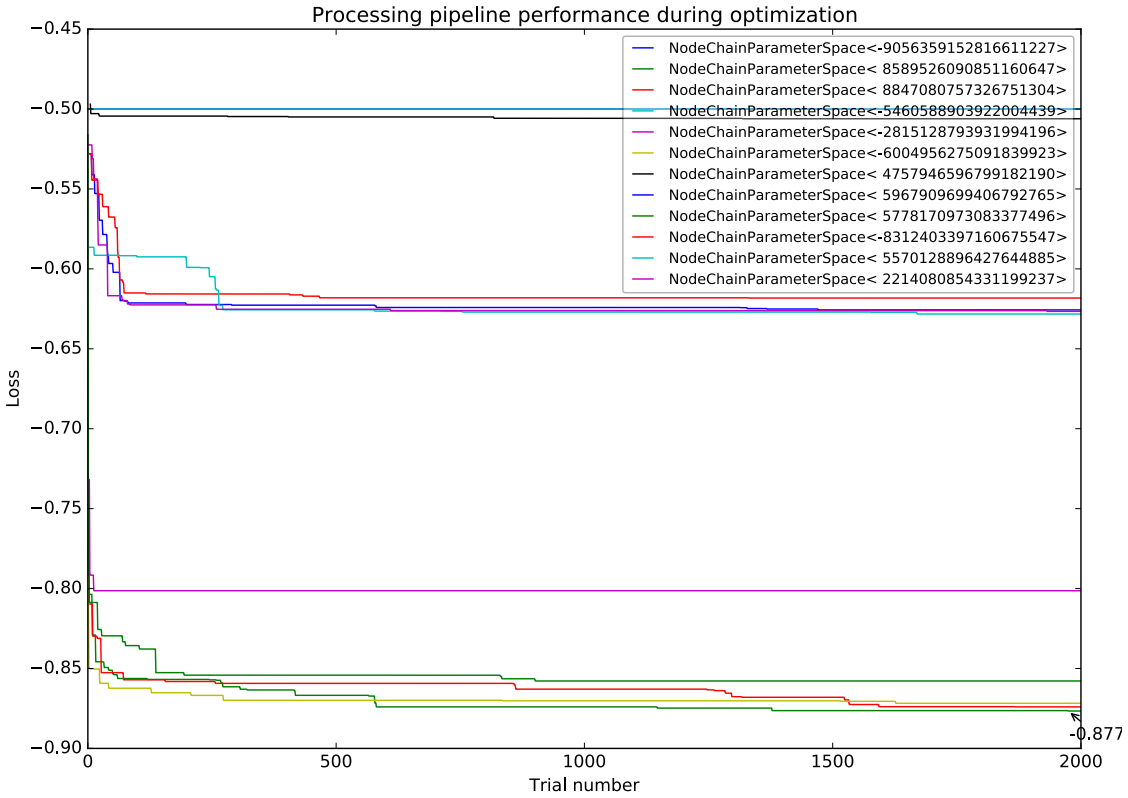


Figure 3.11.: Example plot of a performance graphic

for other purposes and leave the optimization process running. Therefore, this class is used during the optimization to collect the performances of the different pipelines tested and visualizes them as graphs.

Every time an evaluation is done, the result gets added to this class. After a specific amount of time (60 seconds per default), this class will collect all performance values added and generate a file containing plots of the best performance values found for each processing pipeline. Figure 3.11 show an example of such a performance graphic. For every processing pipeline a graph is plotted, which denotes the maximum performance of the pipeline at each trial. Additionally, a global maximum over all processing pipelines is searched and plotted as well. This global maximum is annotated with its corresponding performance. This denotes the performance of the overall best and equals the performance of the processing pipeline currently found as best. Because different metrics might use the performance or a loss as a measurement, this graphic always uses the loss and therefore plots the found minima.

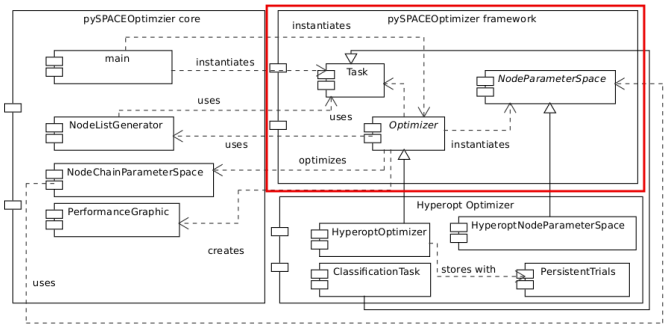
The legend of the figure displays the hash value for every processing pipeline because a

general naming scheme for the pipelines is difficult to find. If more than nine processing pipelines are plotted, no legend will be displayed, because otherwise it might become to large. If the user is interested in the performance of the different processing pipelines, the performance analysis tool (Section 3.2.4) can be used.
This performance graphic gives the user a quick overview which processing pipelines have been processed so far, which performances they have achieved, and which processing pipeline is currently the best one found.
The technical details of the implementation of this class can be found in Appendix A.2.

OptimizerPool To optimize the structure and the hyperparameters of the processing pipelines many evaluations need to be done. To efficiently process these, massive parallelization is needed. The problem that comes with this is that pySPACE itself processes a single processing pipeline evaluation in parallel. This is done because typically a single processing pipeline is executed on multiple datasets and can be paralleled. But if we want to evaluate multiple processing pipelines or even multiple evaluations in parallel, we would spawn a process which itself spawns new processes. This is not allowed in Python by standard. Because of this, the `OptimizerPool` has been created. This class implements a pool of multiple processes which can spawn additional processes. Therefore, this class is used to execute the optimization processes in parallel.

3.2.3. pySPACEOptimizer framework

The *pySPACEOptimizer framework* package implements templates and common functionality that can be used to implement a custom optimizer or a new type of optimization tasks. Therefore, this package defines the required classes and methods that need to be implemented. Additionally, this package implements functions that can be reused by the different optimizers during the optimization process. It includes implementations of an optimizer base class and a base class for different optimization task types. Last but not least this package also defines a base class called `NodeParameterSpace` which defines the parameter space of a single algorithm. The purpose and main functions of these classes will be described in



the following paragraphs.

NodeParameterSpace The `NodeParameterSpace` class implements the determination of the parameter space for a single algorithm. This space can either be defined using the distributions implemented in pySPACE and described in Section 3.2.1 or, if the algorithm is not yet decorated, it is defined by the default values for the parameters. Because of this, every parameter of an algorithm implemented in pySPACE needs to have a reasonable default value. Otherwise, the user needs to set parameters without default values in the task description file like described in Section 3.3.
This class needs to be extended for every optimizer that gets included in the `pySPACEOptimizer` framework, because every optimization algorithm requires it's own representation of the hyperparameter space. This class can be used to gather all hyperparameters with their distributions as defined in pySPACE. A custom implementation will use this definition to map the search domain to a representation the optimization algorithm uses. Multiple implementations of this class can be required for different node types. For example, the `pySPACEOptimizer framework` implements two additional classes for *source* and *sink* nodes. These two node types are treated different, because they should not be optimized at all. A typical *source* node does not contain any parameters to optimize and the *sink* node is used for performance measurement and thus, needs to be configured in the same way for all processing pipelines to make the results comparable. Because of this, the parameter space of these two classes is defined only by the parameters given by the optimization task. Additional information on the details of the implementation are described in Appendix A.3

PySPACEOptimizer The `PySPACEOptimizer` class is an abstract base class of an optimizer. It defines two abstract methods that need to be implemented by a custom optimizer.
Every optimizer requires its own representation of the hyperparameter search space. Because of this, the generation of the processing pipelines needs to be done in two steps. First of all, the `NodeListGenerator` is used to find possible algorithm combinations that form a valid processing pipeline. In a second step, the `PySPACEOptimizer` class calls the first abstract method to create `NodeParameterSpace` objects for these algorithms that represent the search space in a way, the optimizer understand. These nodes will be combined in a `NodeChainParameterSpace` and passed to the second abstract method. This method then optimizes the processing pipeline using the optimization algorithm

it implements. This might include several evaluations of the processing pipelines with different hyperparameter settings.
To keep track of these evaluations and their results, this class implements a queue to which all results will be added. This queue is read by the `PySPACEOptimizer` class and searched for the processing pipeline that achieved the best performance. This pipeline will be stored to disk in a YAML format which pySPACE understands. This file enables the user to use this pipeline to process other datasets from the same domain or to use it in a BCI. Additionally, to give the user an overview of the progress of the optimization, a process bar gets updated every time a new performance result is received.
The technical details of this base class can be found in Appendix A.4. This description includes the methods to implement and additional information on the details of the implementation.

Task To efficiently reduce the number of possible algorithms to create processing pipelines from, the optimization process needs to know which purpose the processing pipeline has. This knowledge can be used to exclude algorithms not suitable for this type of task. To achieve this, the `Task` base class has been created. This class collects all data required from the user to be able to perform the optimization process.
This includes, for example, a *name* of the task. To differentiate between different optimization tasks done with the same dataset and possibly the same processing pipelines, each task has a name. On the one hand, this isolates different optimization tasks but on the other hand this also enables the optimization process to reuse results. If evaluations for a specific processing pipeline are found in a optimization task, these evaluations do not have to be done again and can be reused. This speeds up the optimization process. As already mentioned, knowing the purpose of the processing pipeline can reduce the size of the search domain. Because of this, another aim of the task is to define which algorithms are valid and should be used during the generation. Because of this, a custom implementation of this base class is able to exclude algorithms not suitable for a type of optimization tasks. For example a in *classification* task no *regression* algorithms can be used. Because of this, a *classification* task can exclude them from the search domain of the processing pipelines. Additionally, as described in Section 3.1, depending on the type of task, different algorithm types are required to define a valid processing pipeline. Last but not least, the task is the main object passed between all components. It stores the whole configuration of the task. This includes the optional knowledge of an **domain expert**. How this knowledge can be expressed and how it is used is described in detail

```
name: example_task
input_path: example_data
evaluations_per_pass: 10

type: classification
optimizer: HyperoptOptimizer

class_labels: [Standard, Target]
main_class: Target
```

Figure 3.12.: Example task description

in Section 3.3.

An optimization task is a task that will most likely be executed only once for a specific setup. If it includes domain experts knowledge the definition of such a task can be very large. To enable the user to reuse parts of other task definitions, especially the **domain expert** knowledge, the task is described in a YAML file using a syntax similar to the one used by pySPACE. Figure 3.12 shows a minimal task description with all fields required to set up a valid optimization task. This task description is split into three blocks. The first block defines the parameters needed for every task to describe it. Every task has a `name`, `input_path` of the input dataset, and a number of `evaluations_per_pass`. As in pySPACE, the `input_path` to the input dataset is relative to a pySPACE storage root. By default, passing the value `example_data` results in search for the dataset in `~/pySPACEcenter/storage/example_data`. The path to the pySPACE storage can be adjusted using the pySPACE configuration file as described in Section 3.2.2. The parameter `evaluations_per_pass` defines the number of evaluations that can be done in parallel before new hyperparameter settings need to be sampled. Because the software defaults to 1 pass, this would execute a total of 10 evaluations and can execute all of them in parallel.

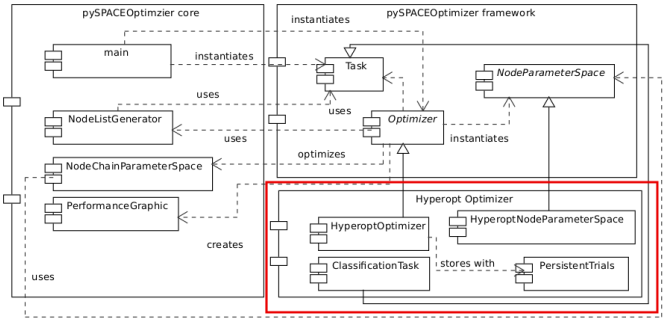
The second block of the task definition defines the type of the task and which optimizer should be used. These arguments are used by the `main` module to instantiate the correct task type and optimizer.

The third block then, defines required arguments for the specific task type. This block may vary depending on the purpose of the processing pipeline. Because this task description executes a classification pipeline, it requires the names of the class labels used

in the dataset and which class the user is interested in.
This is the minimum information required to execute an optimization task. All other fields, like the maximal pipeline length, are set by the software using defaults. A common task description will most likely include additional fields. All possible fields defined by the base class are listed in Appendix A.5. Additionally to these parameters, there are other parameters that express different kinds of **domain expert** knowledge. They will be described in Section 3.3.

3.2.4. Hyperopt Optimizer

The `Hyperopt Optimizer` package is a concrete implementation of the `pySPACEOptimizer framework` for the Hyperopt optimizer library. This package implements a custom task type, a custom node class, and an optimizer. These three things are all that is required to include a new optimizer in the framework.



If no new task type is implemented with the optimizer a new task implementation might not even be needed. Thus, only two classes need to be implemented in most cases.
Additionally to the new task type and the optimizer, this package includes a performance analysis tool. This tool can be used to do offline analysis and inspect the optimization process in more detail.

ClassificationTask A `ClassificationTask` is a new task type and inherits from the `Task` base class. This type of task can be used if the generated processing pipelines should be used to *classify* a given dataset. To do this, a valid processing pipeline requires a *classifier*. Because of this, one thing this class does is redefining the required nodes to add a *classification* algorithm. This ensures that only processing pipelines containing a classifier are considered to be valid. On the other hand, this implies that, for example, an algorithm doing a `regression` is not suitable for the purpose of this pipeline. Thus, the second thing this class does is to redefine the set of possible algorithms to exclude algorithms that are not usable to classify input data.
For a classifier to be able to classify data, it needs to know the class labels used in the

dataset and which label the user is interested in. Because of this, the third and last thing this class does is adding two new required parameters called `class_labels` and `main_class` to the task description.

HyperoptNodeParameterSpace The `HyperoptNodeParameterSpace` class maps the parameter space of a single algorithm to distributions that Hyperopt supports. To do this, it inherits from the `NodeParameterSpace` class and overwrites the parameter space. It first calls the base method, to receive all hyperparameters. Afterwards, it converts the distributions of the hyperparameters to the distributions supported by Hyperopt. By doing this, it redefines the search domain to a format that Hyperopt uses. Because Hyperopt uses different parameters on some of the distributions than the pySPACE extensions do, this class also does some parameter conversions.
Additionally, two subclasses exist implementing the search space for *source* and *sink* nodes. They correspond to the classes defined in the framework. They are both wrappers, that do nothing else than the `HyperoptNodeParameterSpace` base class. They have been included to support future extension of the optimizer.

PersistentTrials During the optimization process Hyperopt stores the evaluations that have been done with the hyperparameters used and the performance result in an object called `Trials`. To make these results persistent and reusable, a class called `PersistentTrials` has been created. This class inherits from the mentioned `Trials` class and implements a mechanism to load and store the evaluations done in a transparent way. Thus, this class overwrites a few methods defined in Hyperopt and adds the functionality to load already done evaluations without executing them again.
This object can be used as a replacement to the `Trials` object used in Hyperopt. Because Hyperopt does not yield the evaluations and their results done during the optimization process, a new method has been added. It is used by the `HyperoptOptimizer` to optimize a single processing pipeline and yield the result of every evaluation to the optimizer. The optimizer can use these results to find the best performance and store them in the performance graphic. This does not force the software to first do all evaluations and afterwards collect the results. Additionally, if the evaluations that should be done, have already been evaluated, Hyperopt would not return them. But the optimizer needs them to correctly keep track of the progress and best performance. Thus, the `PersistentTrials` object will "replay" these evaluations. For every evaluation that has already been evaluated, it does not issue an evaluation but simply returns the result.

This makes it possible to fully replay an optimization process already done and to do optimization experiments in an incremental way.

HyperoptOptimizer The `HyperoptOptimizer` class is the core element of this package. It implements the optimizer used to evaluate the different processing pipeline structures and to optimize the hyperparameters using the Hyperopt library. To do this, this class extends the `PySPACEOptimizer` base class and implements the abstract methods. As described above, one of these methods include the invocation of the optimizer library. In this case, the optimizer is implemented in Hyperopt and thus, this class generates all processing pipelines that need to be tested and afterwards calls Hyperopt to optimize them. Because Hyperopt is a general optimizer, that is capable of optimizing any target function, this class defines a function, that will execute a single processing pipeline with a given hyperparameter setting. Afterwards, this method will collect the performance values of this pipeline and return them as a loss to Hyperopt. Using this loss, Hyperopt optimizes its model of the search domain and samples new hyperparameter values as described in Section 2.3 and Section 2.5. The implementation of the optimizer evaluates the processing pipelines in parallel but not the evaluations of one pass. For the best optimization speed this should also be done. This has been left out in the current software, because a pySPACE backend can only process a single processing pipeline at time and therefore, a separate backend would be required for every evaluation that should be done. Each backend requires resources and opens multiple socket connections. Because of this, this approach does not scale well and a cluster used to do the evaluations soon will run out of resources. Because of this, currently the different processing pipelines are processed in parallel and the evaluations are done in serial. Actually, as shown in Chapter 4 this is not a big issue at all, because in a common setup for an optimization task the number of processing pipelines soon exceeds the resources available. Thus, a parallelization over the evaluations does not speed up the optimization.

Performance Analysis Besides the performance graphic showing the best results found for each processing pipeline, the user might wish to inspect the optimization process in more detail. To do this a performance analysis module has been created. It is located in a `tools` sub-package of the `Hyperopt Optimizer` package. This tool is a GUI based tool to let the user inspect in detail which processing pipelines have been tested, with which hyperparameter settings they have been evaluated, and which performance they achieved. Figure 3.13 shows the GUI of the tool. On the left hand

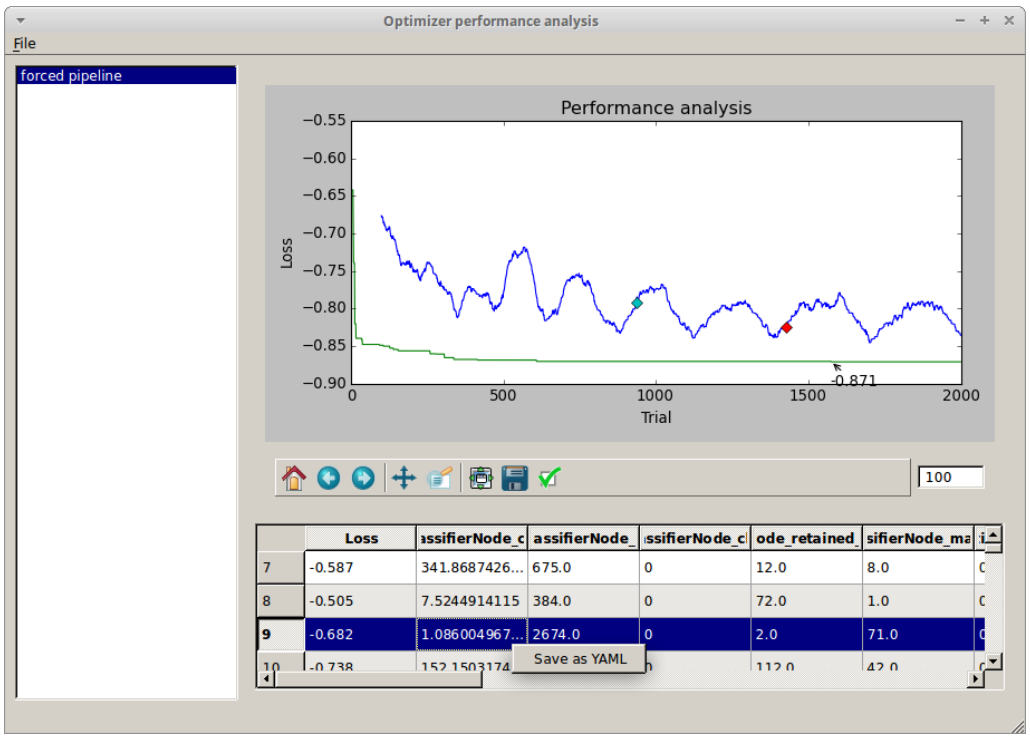


Figure 3.13.: An overview of the GUI of the performance analysis tool

side, a list of all processing pipelines, that have been able to process the input data during the optimization, is shown. For simplicity the name of the processing pipelines is constructed using the names of algorithms not required. In this case, all algorithms have been required and thus, the name "forced pipeline" is given. In this list, the user can choose one or more of the processing pipelines he likes to inspect.

On the upper right side, a plot of all processing pipelines is shown. Additionally, a plot (here in green) of the global best performances of all selected pipelines is shown. Pipelines that have not been selected by the user are made transparent to visualize which processing pipelines have been selected. Next to the controls of the plots, an input field is placed. This field can be used to alter the window size for the *simple moving average (SMA)* of the plots.

If, like in this case, the user has selected only a single processing pipeline, the evaluations are shown with their hyperparameters below the graph. The user can either search in the table and select a single trial or move the mouse over the graph. The current evaluation is shown by the light blue marker in the graph. If the user clicks on a evaluation in the graph, the corresponding evaluation is selected in the table. Additionally, if an evaluation is selected, a marker is placed on the graph. If the user has found an evaluation that

should be used, she or he can do a right mouse click in the table and can save it as a YAML usable in pySPACE as an operation.
This tool allows the user to do detailed inspections on the optimization results and to compare the performance of different processing pipelines during the optimization process. Because this tool relies on the `PersistentTrials` stored by the `HyperoptOptimizer` it does only work for this concrete optimizer (and thus is stored in a sub-package of it).

3.3. Domain expert knowledge

If the software created in this thesis is being used by a **domain expert**, it is most likely the case, that she or he has additional knowledge about the problem. For example, a **domain expert** might know which algorithms might be doing well, which ones cannot process the input dataset or even which algorithms are required to achieve a good performance. In addition, an expert might also know better distributions or even concrete ranges for different hyperparameters.
All this knowledge can be used to reduce the search domain and thus, speed up the optimization process of the software. If, for example, the original search space includes 10 hyperparameters with 5 values each, the search domain would have $5^{10} \approx 9765600$ combinations. If the **domain expert** defines new parameter distributions of 5 hyperparameters, each with 3 values, the search domain is reduced by a factor of ≈ 13 to $5^5 \times 3^5 \approx 759300$ combinations. Because of this, a mechanism to enable an expert to express his knowledge is required. This mechanism has been integrated into the optimization task description and is implemented using the `Task` class. A **domain expert** is able to define a *whitelist* of algorithms that she or he knows to work well on problems in this domain. A *blacklist* is the opposite. If the expert knows algorithms that do not work for his problem they can be excluded using this list.
If the expert has additional knowledge about some hyperparameters, this knowledge can be expressed by changing their parameter ranges. It is important to notice that by setting new ranges or distributions for some hyperparameters, the distributions of the other parameters are not affected. They will be left like the **algorithm expert** defined them and thus, get optimized as well. If, for some reason, a **domain expert** does not want to optimize a hyperparameter at all, this parameter can be marked as not to be optimized as well. In this case, the default value defined in the algorithm will be used for this hyperparameter.
Hyperparameters can be set to single values, a list of values or a distribution. If set to a

```yaml
# Now define new parameter ranges
parameter_ranges:
    - node: SorSvmNode
      parameters:
        # A single values parameter
        max_iterations: 10
        # A list of values
        complexity: [0.001, 0.01, 0.1, 1, 10]
        # A distribution
        epsilon:
            type: Normal
            mu: 0
            sigma: 1
```

Figure 3.14.: Example for *domain expert* knowledge in an optimization task

single value, they will be fixed and not optimized. If set to a list, only values from this list will be sampled. And by redefining a distribution for a hyperparameter, the values get sampled from this new distribution. A **domain expert** is able to use the same distributions as the **algorithm expert** which have been defined in Section 3.2.1. Figure 3.14 shows an example of such a mapping. If a hyperparameter is mapped to a new distribution. A field with the key `type` is required. This field defines the type of distribution to use by simply assigning it the name of the distribution. The parameter name of the distribution gets set automatically in this case. All other fields of the mapping are considered being parameters of the distribution. This enables a **domain expert** to redefine the distributions for every hyperparameters depending on the task.
In some cases, an expert might be interested in processing pipelines with a certain minimal performance (or maximal loss respectively). In this case, the **domain expert** is able to set two additional parameters called `max_loss` and `check_after`. The second parameter defines a number of evaluations after which every processing pipeline is checked. This is required because a processing pipeline might behave bad at the beginning because of bad hyperparameter samples but achieves better results after a few evaluations. The first parameter defines a maximal loss that a processing pipeline must have achieved, to be evaluated further after this check. If a processing pipeline does not meat the required maximal loss, the evaluation will be stopped and an infinite loss will be returned for all remaining evaluations. This can largely speed up the optimization process because those processing pipelines will not be evaluated until the end.

4. Evaluation

This chapter covers the evaluation of the approach presented in Chapter 3. To evaluate that the presented approach works like expected, six different experiments have been done. Each of them uses less **domain expert** knowledge, to proof that the approach works with the given knowledge. Additionally, this tests how much **domain expert's** knowledge is required for this method to be able to achieve good results in an reasonable amount of time.
Thus, the first experiment will evaluate the performance of a processing pipeline that has been developed by **domain experts** and is completely handcrafted. The results of this evaluation are used as baseline to compare the other experiments to. The approach described in Chapter 3 creates the search domain for the hyperparameters dynamically depending on the processing pipeline. Thus, the second experiment will evaluate that this method is able to search this domain for optimal hyperparameter settings at all. This will be done by using the structure of the baseline and using the presented method to optimize the hyperparameters of the classifier. After this, the third experiment will answer the question if **domain expert** knowledge is required to optimize the hyperparameters. Again, this will be done by using the structure of the baseline. But this time no **domain expert** knowledge about the hyperparameters is given and all of them will be optimized automatically. After these experiments it has been evaluated, that the described method is able to optimize the hyperparameters of a given processing pipeline. Thus, the fourth experiment will evaluate whether the structure can be optimized as well. This will be done by leaving out the classifier of the baseline and letting it be chosen automatically. Therefore, in this experiment the classifier and its hyperparameters will be chosen automatically. Afterwards, this can be extended again, by giving less **domain expert** knowledge about the structure of the processing pipeline. Because of this, the fifth experiment leaves out the classifier and the feature generator as well. This experiment will show whether the new approach is able to optimize a processing pipeline in which multiple nodes need to be chosen and whether the number of processing pipelines generated can still be handled. Last but not least, the sixth experiment will evaluate the extreme case. This experiment simulates the usage of the approach by a non-expert who does not have any **domain expert** knowledge at all. Thus, no knowledge about the structure or any hyperparameters is given.

4.1. The evaluation scheme

To understand the experiments described in the following sections, the data and evaluation scheme used, need to be described. To be able to compare the results of the baseline and the experiments they need to be evaluated in the same way on the same data. Thus, an evaluation scheme is required.
But first, the data used for the evaluation need to be described as well. For the evaluation EEG data have been used. They have been recorded by the DFKI RIC and have been used for the experiments of Kirchner et al. [16]. In their experiments they used recordings of P300 brain measurements to show that a special method, called *brain reading*, is able to detect patterns in an EEG that are related to P300-ERPs (Section 2.7). This method uses a complex processing pipeline to classify the EEG data [16]. They recorded this data using an experiment in which the subject is playing a labyrinth game while wearing a head mounted display. The game is used to create some workload on the subject while the display is used to give him or her different visual stimuli. The "standard" stimulus is used most often and does not require a response of the test subject. Additionally, two different "target" stimuli are infrequently shown. If the subject is shown one of these two "target" stimuli, she or he needs to respond to this by pressing a buzzer. This results in three different types of data. The standard stimulus could have been given, denoting unimportant information, or the target stimulus has been given and the subject responded by pressing the buzzer or, as third, the target stimulus is shown but the subject did not respond to it. These events have been marked as "missed target". [16]
This experiment has been done with five different subjects. Each subject has done the experiment in two sessions on two different days. Each session consisted of five runs. Thus, in total $5 \times 2 \times 5 = 50$ different recordings have been done. From these, the recordings of one subject have been split up into the training dataset and an evaluation dataset. The first three recordings of a single session are used to optimize the processing pipelines and train it during the evaluation. For all evaluations the same training and optimization data have been used to be able to compare the results as the optimization result is likely to be dependent on the input data. The last two recordings from the first session are used to do an evaluation of the performance on the same recording session. The second session of the same test subject is used to perform an evaluation of the performance on another recording session from the same subject. Last but not least, the recordings of the other subjects have been used to evaluate the performance on other

subjects.
This results in three different evaluations done for each experiment. Each evaluation tests the processing pipeline on a different level. The first evaluation, on the same session, tests whether the processing pipeline is able to classify new data at all. If this evaluation fails, the processing pipeline would not be usable at all. The second evaluation tests, if the processing pipeline is able to generally classify data from the subject trained for. The last evaluation tests, if the processing pipeline needs to be retrained for every subject or whether it is able to classify data from another subject as well. The performances achieved in these evaluations will then be compared to a processing pipeline that is currently used at the DFKI RIC. To be able to compare the results, a common metric has to be used for all experiments. In this thesis the *balanced accuracy* is used to measure the performance. This metric is used because it tries to exclude the effect of a bias from the performance. A bias of a processing pipeline might arise from an imbalanced dataset, because the processing pipeline has many examples for one class and only few for the others. Because of this, the processing pipeline, and especially the classifier, might tend to classify a new data point with the class from which many samples are available, because in most of the cases during training this was correct. The balanced accuracy eliminates this effect by taking the number of positives and negatives into account [10]. Thus, the balanced accuracy is defined as follows:

$$\frac{1}{2}(\frac{\mathrm{TP}}{\mathrm{P}} + \frac{\mathrm{TN}}{\mathrm{N}})$$

With `TP` being the number of samples that have been correctly classified as positive, `P` being the number of positive samples in total, `TN` being the number of samples correctly classified as negative, and `N` being the number of negative samples in total. This definition weights the true results of the processing pipeline with the total number of samples of the corresponding class. Thus, the correct classification of the both classes get taken into account with same weights.
The baseline uses a two-class classification and thus, like in the paper from Elsa Kirchner et al. [16], the evaluation done in this thesis have left out the third label ("missed target").

4.2. Performance evaluation with the baseline structure

During their experiments Kirchner et al. [16] developed a processing pipeline that is able to classify P300-EEG data. This pipeline is used as reference and baseline for all experiments done to evaluate this thesis.
Because of this, the following experiment will evaluate the performance of the pipeline that has been used at the time of writing this thesis.

Setup Figure 4.1 shows the setup of a simplified version of the processing pipeline developed by Kirchner et al. [16]. This pipeline uses (as every pipeline) a source node

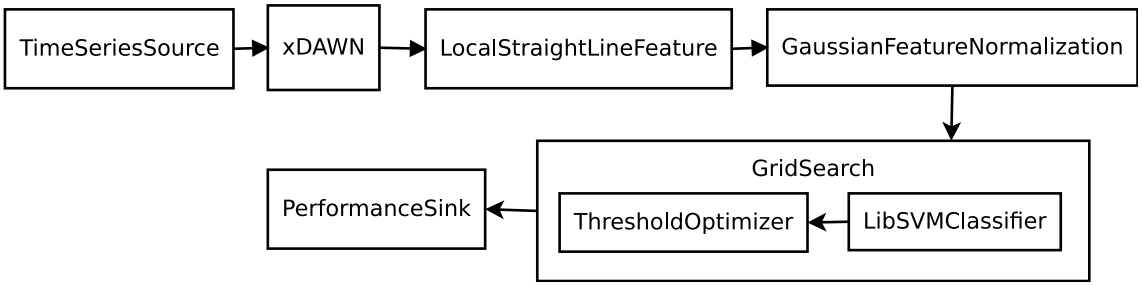


Figure 4.1.: Setup of the P300 processing pipeline

to provide the data. In contrast to the processing pipeline developed by Kirchner et al. the processing pipeline used in this thesis does less preprocessing. The original pipeline reduces the frequency of the samples by down sampling them. Additionally, the time continuous data get windowed. In this thesis this preprocessing steps have already been done on the input data to reduce the size of the processing pipeline and make the processing faster. Of course this steps could be integrated into the optimization process later. After this preprocessing, both processing pipelines are the same again. A spatial filter, called `xDAWN`, is used to reduce the number of channels to be used to eight. From these channels features get generated using a `LocalStraightLineFeature` generator. After this, these features get normalized using the `GaussianFeatureNormalization` and then classified by an SVM. This classifier uses a linear kernel and the regularization parameter is chosen using a `GridSearch`. As last processing step, the threshold gets optimized according to a given metric by the `ThresholdOptimizer` to achieve better performances on the given metric. Last but not least, a sink node is used to store (or evaluate) the data. The full description of the processing pipeline with all parameters that have been set is shown in Appendix C.1.

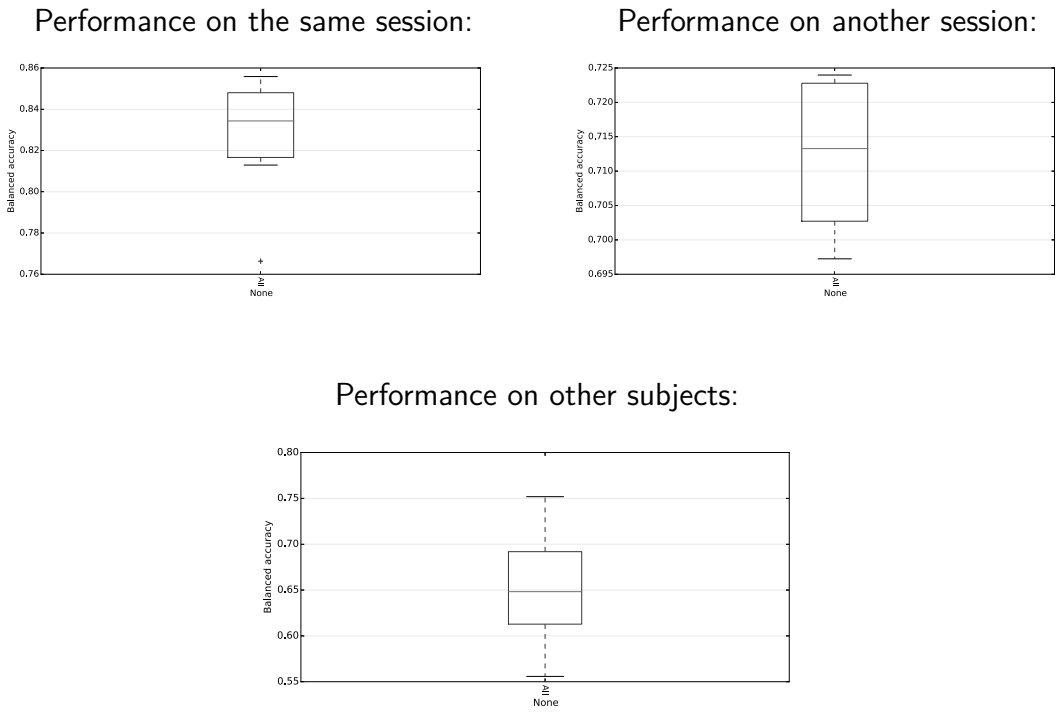


Figure 4.2.: Evaluation performance of Experiment 4.2

Results Figure 4.2 shows in three plots the performance of the baseline when evaluated using the three evaluation data sets. The first plot shows the performance when evaluated using the remaining runs from the same session. In this case, the processing pipeline achieves an average performance of $\approx 83\%$. The best performance achieved is $\approx 86\%$.

The second plot shows the performance on another session from the same subject. In this case the performance is worse than it was when evaluating on the same session. The average performance is $\approx 71.5\%$ with the best performance at $\approx 72.5\%$.

The last plot shows the performance on a session from another subject. In this scenario, the processing pipeline achieves the worst performance of all three evaluations. The average performance is $\approx 65\%$ and the best performance in this case $\approx 75\%$.

Discussion As described in the setup and shown in Figure 4.2, the processing pipeline already achieves good performance results when trained with data from the same session than used for the evaluation. If the evaluation data are from another session or, even worse, from another subject, then the performance decreases. This behavior is expected,

as even a single subject behaves different between two sessions and another subject behaves even more different. Because of this, the EEG data are different between two sessions and subjects. One might overcome this by training the processing pipeline using data from different sessions or even different subjects. This will increase the variance in the input data set and thus, might reduce the overfitting of the processing pipeline to a single subject [13]. This might reduce the performance on the same session but increases the overall performance. But as this is a problem of the right choice of the training data and not an optimization problem, this has been left out in this work. Because of time and resource constraints this has not been evaluated and only evaluation scheme mentioned above has been used.

4.3. Convergence of the model

This experiment evaluates, if the approach described in this thesis is able to optimize the hyperparameters of the given processing pipelines. The hyperparameter optimization is done using the third party library Hyperopt. Internally, it trains a model of the hyperparameters' distributions. If this model converges to a minimum than it has been optimized and the optimal hyperparameters have been found. Because this convergence might require an infinite number of evaluations, this experiment will try to make as many evaluations as required to show the tendency of a convergence.

Setup The convergence is evaluated using three train and test runs of a single subject during a single recording session of a P300-EEG experiment. Appendix C.2 shows the optimization task description used to evaluate this. The task forces only a single processing pipeline to be generated by the software. This is done by adding the algorithms that should be used to the `whitelist` and the required nodes. Thus, only a single combination of these algorithms is left as being valid. This processing pipeline is the same as used in Experiment 4.2, but this time the hyperparameters of the classifier do not get set by the **domain expert** but will be optimized automatically. To reduce the number of evaluations needed until the model starts to converge, the other parameter settings of the baseline have been used as well. This processing pipeline has been used to do 200 evaluation passes, each consisting of 10 evaluations. Therefore, a total of 2000 evaluations have been done. Because balanced accuracy is a performance metric and Hyperopt minimizes a loss, the loss of the pipeline is the negative classification performance.

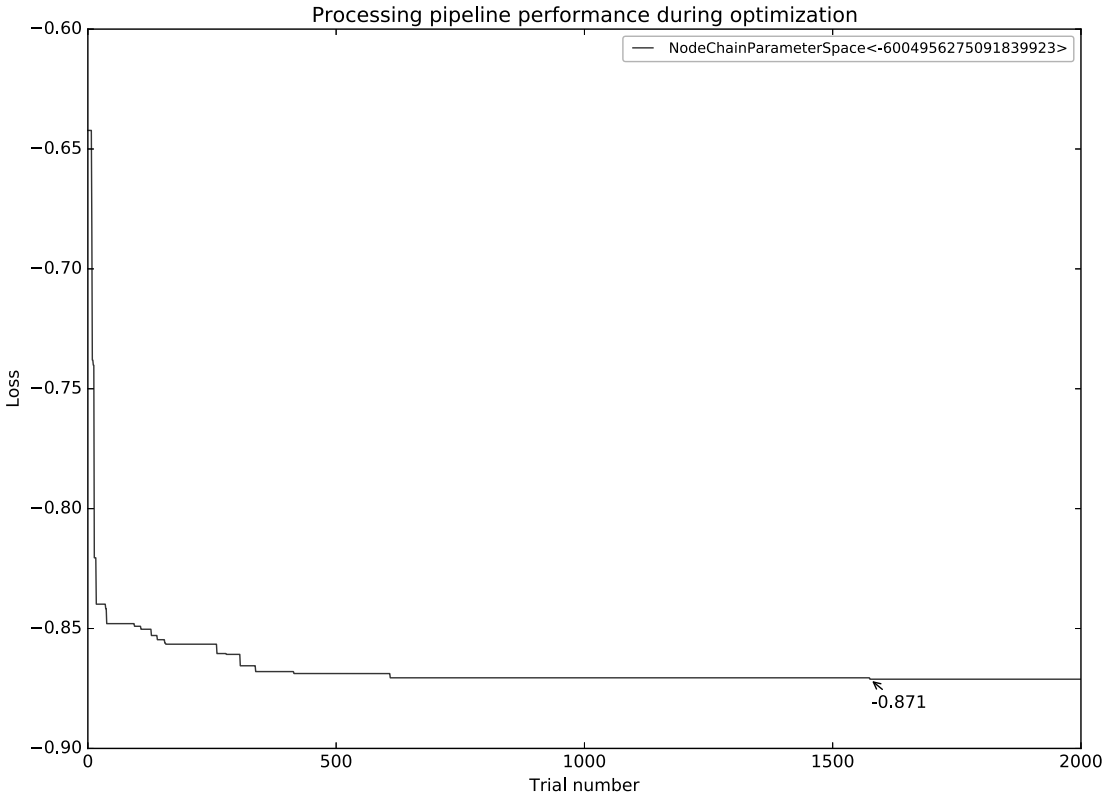


Figure 4.3.: Optimization performance of Experiment 4.3

Results Figure 4.3 shows a plot of the best performance values of the processing pipeline during the optimization process. The graph denotes the performance of the best hyperparameters found after the number of the trials. After approximately 1600 trials, the last minimal loss of -0.871, which expresses a performance of 87.1% balanced accuracy, was found. The graph also denotes that at the beginning of the experiment the performance gets better fast and in large steps from $\approx 65\%$ up to approximately 85%. The periods between two local minima gets larger with the number of the evaluations done. Additionally, the performance gain gets smaller as well. Figure 4.4 shows the performance values on the three different evaluation datasets. To be able to compare the performance values with the ones from the baseline, the right box plot of each graph shows the performance of the baseline and the left plot shows the performance of the optimized processing pipeline. The evaluation on the dataset from the same session results in an average performance of $\approx 85\%$ balanced accuracy. In comparison to the average of $\approx 83\%$ of the baseline the optimized processing pipeline performs $\approx 2\%$ better. This is nearly the same performance that has been achieved during the optimization process.

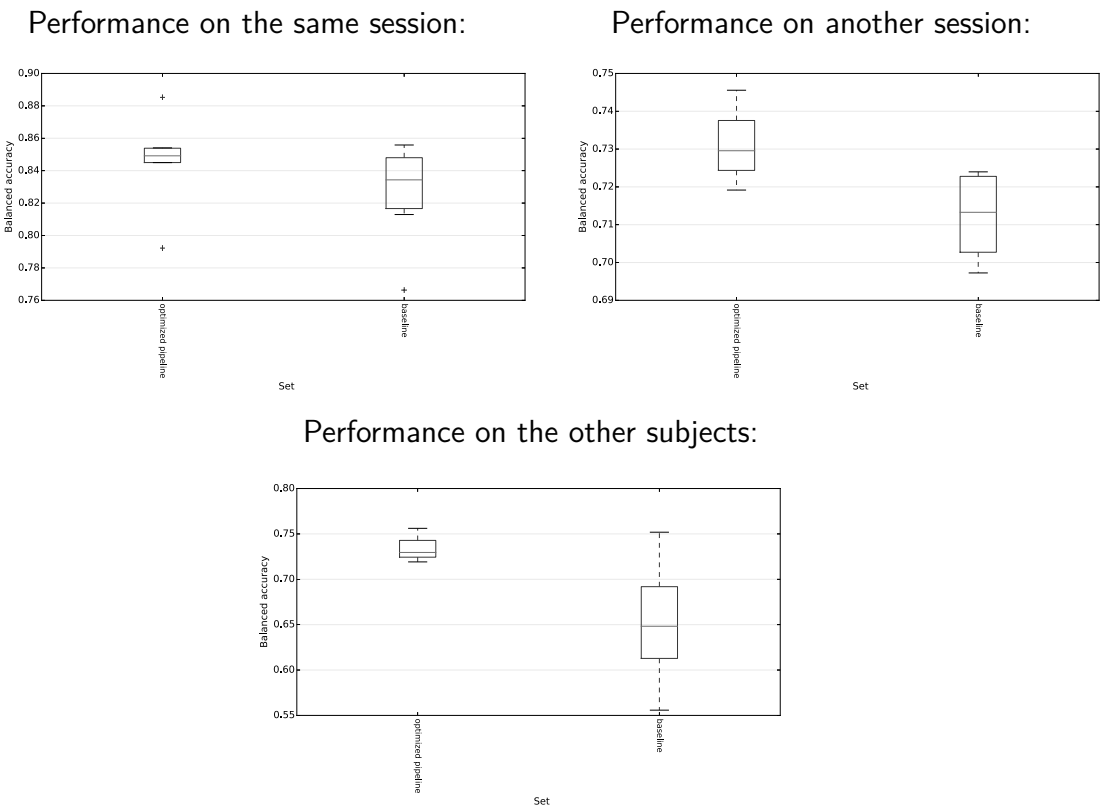


Figure 4.4.: Evaluation performance of Experiment 4.3

The second plot shows the evaluation performance on a data set from another session of the same subject. As in Experiment 4.2, the performance is below the one achieved when evaluating on the same session. The achieved average performance is $\approx 73\%$ balanced accuracy. Compared to the average performance of the baseline ($\approx 71.5\%$) the performance has again increased $\approx 1.5\%$ balanced accuracy. Last but not least, the third plot shows the performances when evaluating the processing pipeline on the recordings from the other subjects. In this case, the achieved average performance is, with $\approx 73\%$ balanced accuracy, the lowest of the three evaluations done. With an average performance of $\approx 65\%$, the baseline performed $\approx 8\%$ worse.

Discussion As shown in Figure 4.3, the number of evaluations between two found local minima gets larger with the number of evaluations done. At first, the number of found minima is high, which is expected. As the pipeline starts with large distributions for the hyperparameters and therefore, many settings that are sampled are better then the already tested ones. As the number of evaluations increases, the number of found

minima decreases, because the parameters are close to the optimal values. Because of this, no, or only a small, performance gain is received by optimizing them. This explains the reduction of the performance gain over the number of evaluations as well. The closer the hyperparameters get to the optimal values, the smaller the performance gain will be. This behavior denotes a convergence of the model that is trained by the optimizer. At the beginning, the model needs to explore a lot and finds good values by chance. After some evaluations, the predictions of the model get better and new minima are found. This adds new confidence to the model and the hyperparameters get closer to the optimal values. Therefore, the predictions of the model only do small changes to these parameters and thus only small performance gains can be achieved.
Last but not least the optimized processing pipeline found, using the method of this thesis, outperforms the baseline by 1.5 to 8% balanced accuracy. Because the performance gain exists on all three evaluation datasets, this has not been achieved by overfitting the processing pipeline to the input data. Because of this, the processing pipeline generalizes well and is usable for other subjects as well. As both processing pipelines use the same values for the hyperparameters, set by the **domain expert**, this performance gain needs to be achieved by optimizing the remaining hyperparameters. The baseline did set the class weights of the SVM to $1:5$, which correlates to a ratio of 0.1. The optimizer did chose a class ratio of 0.4 which equates weights of $1:1.25$. Despite this difference, the other parameters have been chosen similar, with the only difference, that the optimizer did optimize all hyperparameters. Thus, the performance gain is achieved by optimizing the hyperparameters which have been left to default values by the **domain expert**. Appendix D.1 shows the complete processing pipeline that has been considered as best with all hyperparameters.
These results lead to the conclusion that the used approach is able to optimize the hyperparameters of a processing pipeline as used by pySPACE.

4.4. Complete optimization of a single processing pipeline

After it has been evaluated whether the presented approach is able to optimize the hyperparameters of a processing pipeline at all, this experiment will evaluate the effect of the **domain experts** knowledge. Thus, it optimizes the same processing pipeline as used in Experiment 4.3 but without any additional **domain expert** knowledge. Afterwards,

the results of this experiment will be compared to the results of Experiment 4.3 to evaluate whether this approach was able to find a hyperparameter setting comparable to the one defined by the **domain expert** or maybe even a better one.

Setup The complete description of the optimization task used for this experiment is shown in Appendix C.3. The setup of this experiment is very similar to Experiment 4.3 as the same single processing pipeline is used. To be as comparable as possible, this experiment also does 2000 evaluations of the processing pipeline. The only difference between these two experiments is that the **domain expert's** knowledge, defined in Experiment 4.3, has been left out. Thus, the optimization process starts using the hyperparameter distributions defined by the **algorithm expert** for every parameter that should be optimized.

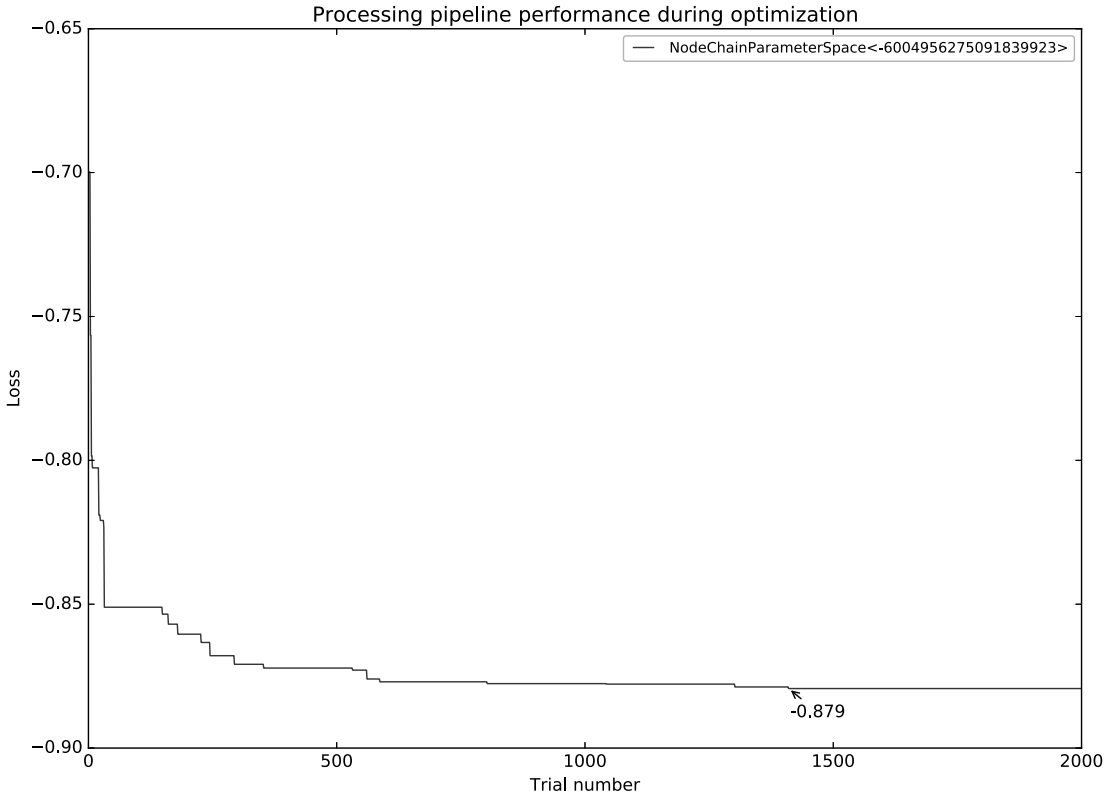


Figure 4.5.: Optimization performance of Experiment 4.4

Results Figure 4.5 shows the performance of the processing pipeline during the optimization process. The plot shows the best found loss of the pipeline at each evaluation.

Like in Experiment 4.3 the optimization starts with many local minima at the beginning and the more evaluations have been done, the less new minima are found. The last minimum has been found after approximately 1450 evaluations with a performance of $87,9\%$ balanced accuracy. Figure 4.6 shows the performance values of the optimized

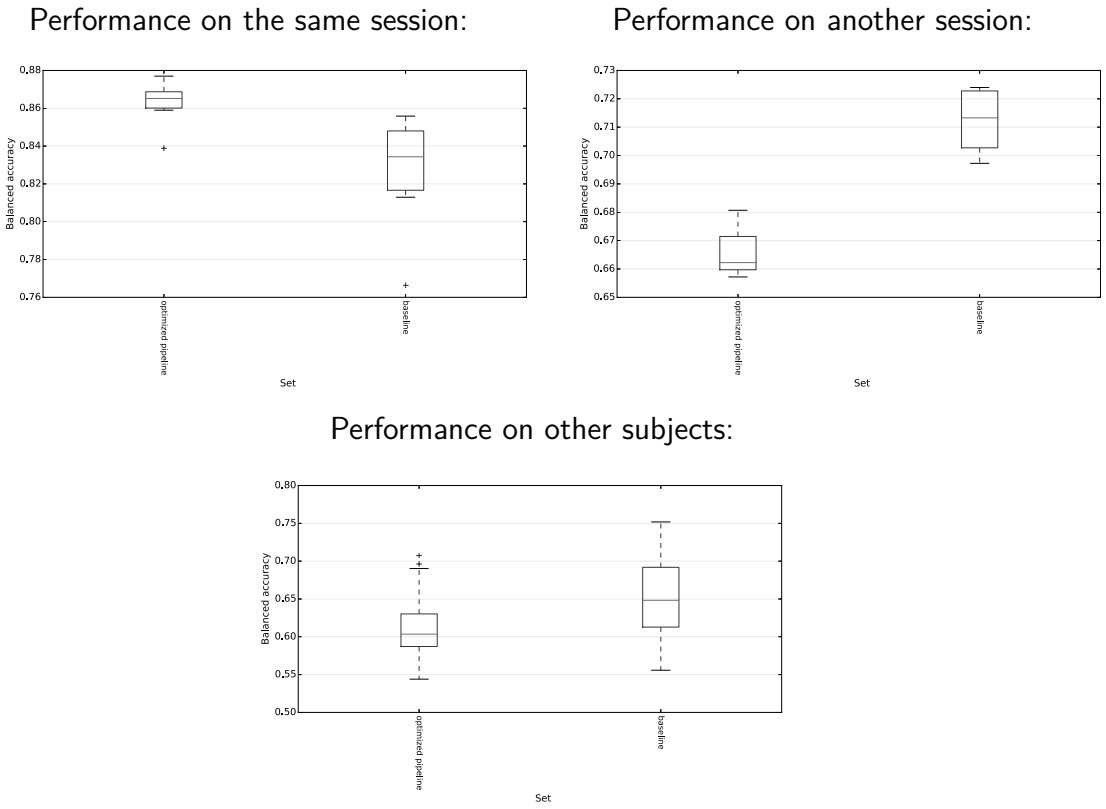


Figure 4.6.: Evaluation performance of Experiment 4.4

processing pipeline on the different evaluation datasets. Like before, the right box plot shows the performance of the baseline and the left plot the performance of the optimized processing pipeline. The average performance of the best processing pipeline found, when evaluating on the same session than used for the optimization, is $\approx 87\%$ balanced accuracy and thus ≈ 4 percentage points better than the baseline with $\approx 83\%$ balanced accuracy. The second plot shows the performance of the evaluation on another recording session from the same test subject. The average performance of the optimized pipeline is $\approx 66\%$ balanced accuracy. In this case, the baseline is, with an average performance of $\approx 71.5\%$, ≈ 5.5 percentage points better as the optimized pipeline. Last but not least, the third plot shows the performance of the evaluation on the other subjects. In this case, the optimized configuration achieves an average performance of

$\approx 60\%$ balanced accuracy and thus is again worse than the baseline with $\approx 65\%$.

Discussion As shown in Figure 4.5 and the first plot of Figure 4.6, the approach presented in this thesis is able to optimize the processing pipeline. The performance achieved when optimized on the same session that has been trained on, the performance is better than the baseline. But, in contrast to Experiment 4.3, the performance values of the evaluations on another session and other subjects are worse than the ones achieved by the baseline. This indicates that the performance gain on the same session is achieved by overfitting to the input data. As the structure of the processing pipelines is the same in all three experiments, this overfitting needs to occur because of the hyperparameters. By comparing the hyperparameters chosen for the processing pipeline in Experiment 4.3 (Appendix D.1) and this experiment (Appendix D.2) it can be observed that most of the parameters have been chosen similar. Because of this, it can be concluded, that even small changes in the hyperparameters can change the performance significantly. Of course, the overfitting reduces the usage of the processing pipeline for other subjects or other data. The effect of overfitting a processing pipeline is currently not addressed by the optimization software directly. But, if the variance in the training data is kept high enough, this problem can be worked around. For example, if the training data set would consist of data from multiple recording sessions of a single subject the effect of overfitting to a single recording session can be reduced. Because each optimization process takes up to several days, this has not been done in this thesis and the current evaluation scheme has been kept.

4.5. Selection of the classifier

The following experiment tries to evaluate, whether the structure of the pipeline can be optimized at all. To evaluate this, the classifier has been left to be chosen by the software.

Setup The setup of this experiment is similar to the one described in Experiment 4.3. The difference is, that the structure of the pipeline was not completely fixed. During this task one node (the classifier) has been left out of the forced nodes and therefore has to be chosen by the software. The schematic setup of these pipelines it the same as shown in Figure 4.1. This results in 20 possible processing pipelines, each with a different classifier.

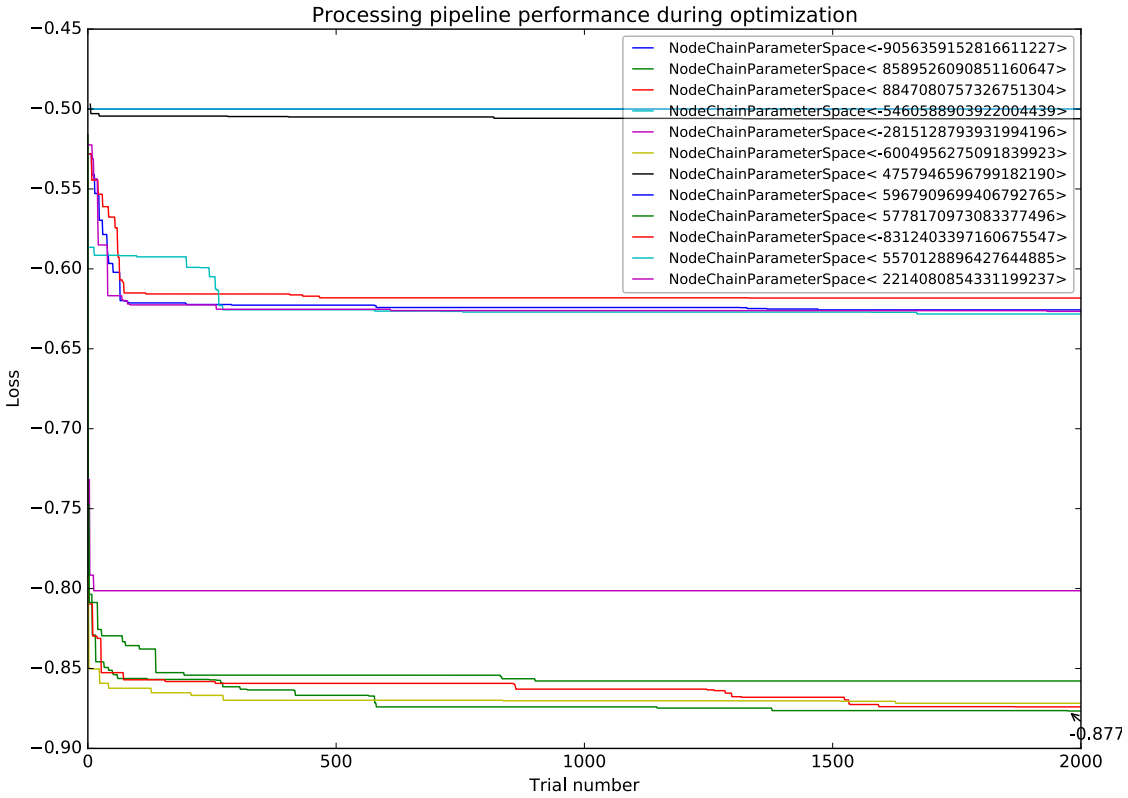


Figure 4.7.: Optimization performance of Experiment 4.5

These pipelines have been optimized by doing 200 evaluation passes consisting of 10 evaluations per pass. Thus, a total of 2000 evaluations have been done for each pipeline. The complete optimization task description of this experiment can be found in Appendix C.4.

Results Figure 4.7 shows the performance values of the different pipelines found by the software during the optimization process. First of all, only 12 processing pipelines of the 20 possible pipelines have been able to process the input data. The remaining eight pipelines have not been able to achieve any results at all. Thus, the performance for these pipelines is $-\infty$ for each trial. Because of this, the performance of these processing pipelines is not plotted. The graphs display the found minimum for each processing pipeline at each evaluation. They can be split into three different groups. The first group achieves approximately 50% balanced accuracy and consists of the `BayesianLinearDiscriminantAnalysisClassifier`, the `AdaptiveThresholdClassifier`, and the `RandomClassifier`. The second group

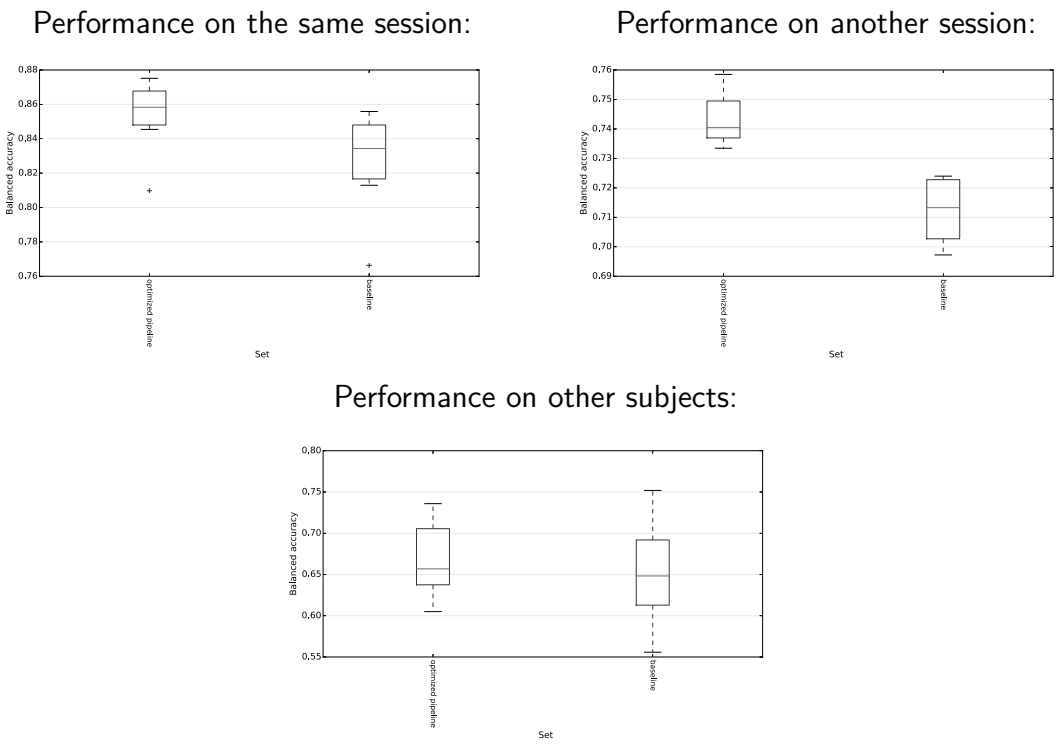


Figure 4.8.: Evaluation performance of Experiment 4.5

stays at approximately 60%. It contains the `UnaryPA0`, `UnaryPA1`, `UnaryPA2`, and the `SvddPassiveAggressive` classifier. The third and last group goes up to 85% balanced accuracy. This group contains the `QuadraticDiscriminantAnalysis`, `LibSVMOneClass`, `LibSVM`, `RMM2`, and the `SorSVM` classifiers. With exception of the `QuadraticDiscriminantAnalysis`, all classifiers in this last group are different variants of SVMs.

The global minimum of all processing pipelines is displayed using an annotation and was found right before the end of the optimization process with a performance of 87.7% balanced accuracy. The pipeline uses the relative margin machine (`RMM2`) classifier. This classifier is a variant of SVMs that uses an additional margin for the classification, that is relative to the input dataset [18]. The complete definition of the best processing pipeline can be found in Appendix D.3.

Figure 4.8 shows the performance of the best found pipeline when evaluating it on the three evaluation datasets. The average performance of this processing pipeline is $\approx 86\%$ balanced accuracy, when evaluated on the same session that has been trained on. Compared to the $\approx 83\%$ of the baseline, this is an improvement of about 3% balanced accuracy. The second plot shows the performance during the evaluation on another

recording session of the same subject. Here the average performance is, with $\approx 74\%$, $\approx 2.5\%$ balanced accuracy higher than the performance achieved by the baseline. Last but not least, the third plot shows the classification performance when evaluated on the remaining subjects. In this case, the average performance is $\approx 66\%$ balanced accuracy. In this evaluation, the baseline achieved $\approx 65\%$ balanced accuracy in average and thus, performs nearly equally well.

Discussion As Figure 4.8 shows, the method presented in this thesis has found 12 processing pipelines that have been able to process the input data. The other eight pipelines have not been able to process the input data. They raised exceptions of different kinds. Most of them have not been able to process the input data because the classifier did not converge and thus could not be trained. Others have raised exceptions because of a bad implementation that did expectations about the input data that have not been true in this case. Both cases might be worked around if pySPACE's type system can be made more fine grained. For example every classifier has the type `FeatureVector` as input independent of the fact that a classifier might only be able to process a specific type of feature vectors.

Additionally, as Figure 4.8 shows, the processing pipelines tend to behave similarly if the structure is similar. As described above, the processing pipelines can be split into three groups. The first group with an average performance of $\approx 50\%$ balanced accuracy is formed by classifiers that use random classification or don't have hyperparameters that can be optimized. Because of this, the performance remains nearly constant (with respect to some randomness) and does not improve over time.

The second group with a performance of $\approx 60\%$ balanced accuracy is formed by classifiers that do have hyperparameters that get optimized. But the classifiers are not able to successfully classify the input data because they are not suitable for the input data. All classifiers in this class are used for single class classification. As described in Section 4.1, the input data use two class labels and thus are for a binary classification. Because of this, these classifiers always have a high rate of miss classification even with optimized hyperparameters.

The third and last group of processing pipelines is defined by algorithms that are able to successfully process the input data and thus, are the pipelines the user is interested in. These processing pipelines use classifiers that are powerful enough to classify the complex input data. The hyperparameters of these algorithms get optimized during the process and the performance of these processing pipelines increases to the optimum.

Like in Experiment 4.4, the performance of the processing pipeline found as best is better than the performance of the pipeline of Experiment 4.3. Additionally, the processing pipeline found in this experiment is not overfitted to the training data and achieves good performances when evaluated on other sessions or subjects as well. Because the processing pipeline optimized in this experiment uses a variant of the SVM as well (Appendix D.3), the hyperparameters can be compared to the one of Experiment 4.3 (Appendix D.1). This shows, that the regularization parameter (for historic reasons named `complexity` in the pipelines) is chosen to be different (≈ 0.005 versus ≈ 0.2). All other hyperparameters are similar. Thus, the performance improvement on the same test subject comes from the relative margin that the `RMM2` classifier uses and the regularization parameter.
This experiment shows that the approach used in this thesis is able to generally optimize the structure of the processing pipeline by choosing the best performing processing pipeline.

4.6. Optimization of structure and hyperparameters

This experiment will evaluate that the approach presented in this thesis is able to optimize the structure and the hyperparameters of a processing pipeline at the same time. As Experiment 4.4 and Experiment 4.5 have already shown, the method developed in this thesis is able to optimize the structure and the hyperparameters of a single algorithms in the pipeline. Thus, this experiment will evaluate, whether the structure and the hyperparameters of multiple algorithms of the processing pipeline can be optimized in parallel.

Setup The setup of this experiment is similar to Experiment 4.4 and Experiment 4.5. As in Experiment 4.4, no **domain expert** knowledge about the hyperparameters of the processing pipeline is given. Additionally, like in Experiment 4.4, the knowledge about the structure of the processing pipeline is also reduced. Unlike before, in this experiment the *classifier* and the *feature generator* will be chosen using the approach presented in this thesis. Thus, only the *source*, *spatial filter*, *normalization*, and *sink* node have been fixed. The complete description of the optimization task can be found in Appendix C.5.

Results The definition of the optimization task resulted in 200 different processing pipelines, that needed to be checked. Each pipeline is a combination of one of 10 possible

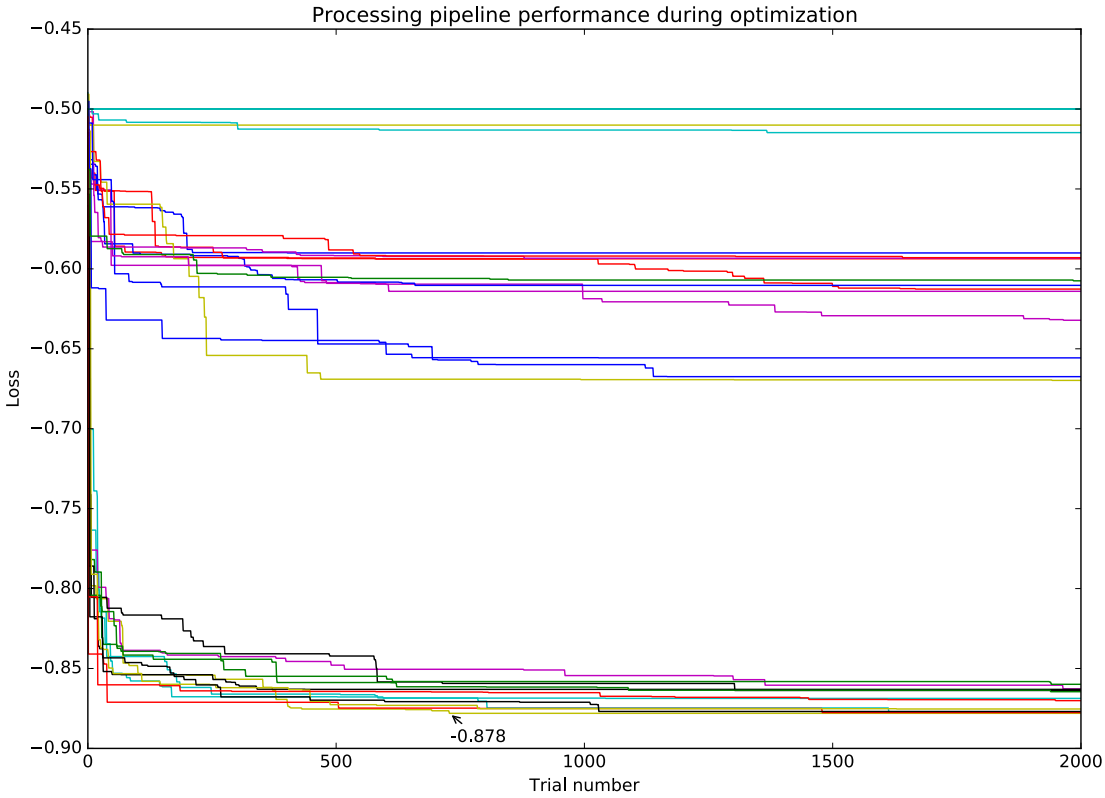


Figure 4.9.: Optimization performance of Experiment 4.6

feature generators and one of 20 possible *classifiers*. Figure 4.9 shows the performance of these processing pipelines during the optimization process. First of all, of the 10 possible `feature generators`, only three (`LocalStraightLine`, `SimpleDifferentiation`, and `TimeDomain`) have been able to generate features from the input dataset. Additionally, from the 20 possible *classifiers*, only eleven have been able to classify the data. This results in 33 processing pipelines, that have been able to process the input dataset. Like in Experiment 4.5, these pipelines can be divided into three groups. The first group, at about 50% balanced accuracy, includes nine processing pipelines. They use the same classifiers as in Experiment 4.5. They are the `AdaptiveThresholdClassifier`, `BayesianLinearDiscriminantAnalysisClassifier`, and the `RandomClassifier`. The second group achieves performances of about 65% balanced accuracy. This group consists of twelve processing pipelines and includes the processing pipelines with the `UnaryPA0`, `UnaryPA1`, `UnaryPA2`, and `SvddPassiveAggressive` classifiers. The third and last group achieves performances of about 85% balanced accuracy. This group also includes twelve processing pipelines. These pipelines use the `LibSVM`, `SorSVM`,

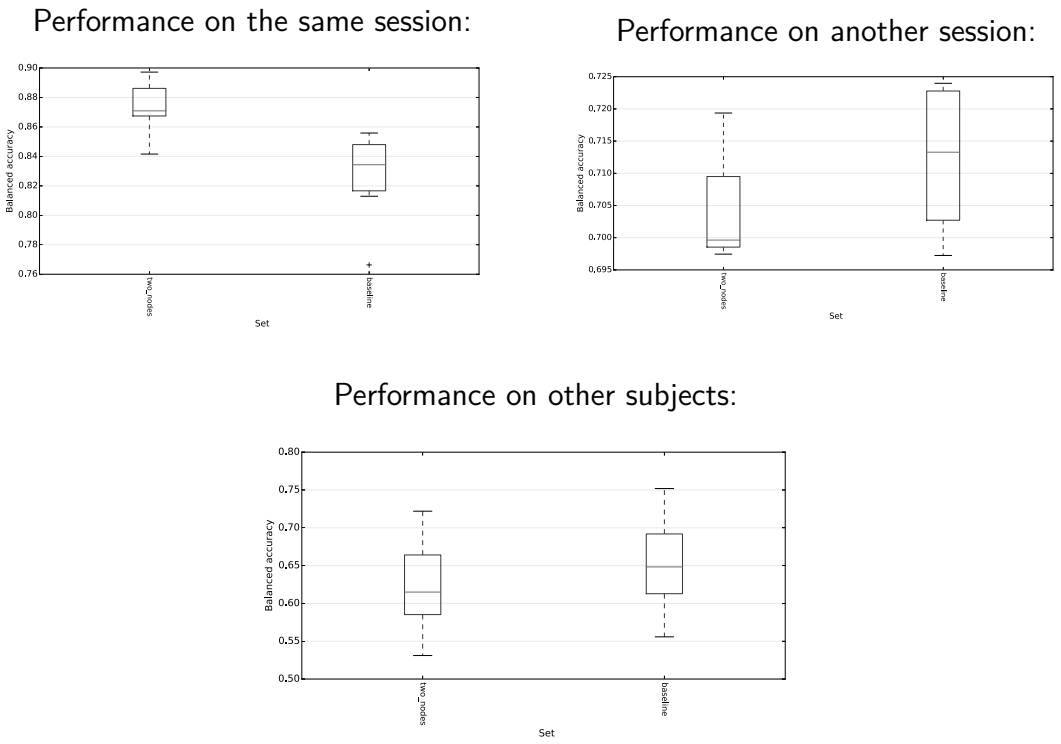


Figure 4.10.: Evaluation performance of Experiment 4.6

`LibSVMOneClass`, and `RMM2` classifiers. This group includes the best processing pipeline, which achieved $\approx 87.8\%$ balanced accuracy. This pipeline uses the `LocalStraightLine` feature generator and the `RMM2` classifier.

Figure 4.10 shows the performance of this processing pipeline when evaluating it the other datasets. As always, the first plot shows the performance of the processing pipeline, when evaluated on the remaining runs from the same recording session than the ones that have been used for optimization. In this case, the found pipeline achieves an average performance of $\approx 87\%$ balanced accuracy. This is a performance gain of $\approx 4\%$ balanced accuracy when compared to the $\approx 83\%$ of the baseline. The second plot shows the performance when evaluating it on another session from the same test subjects that has been used to optimize the pipeline. The performance of the found pipeline drops to $\approx 70\%$ balanced accuracy. This is $\approx 1.5\%$ balanced accuracy below the performance of the baseline. Last but not least, the third plot shows the performance of the evaluation on the remaining subjects. The found processing pipeline achieves $\approx 61\%$ balanced accuracy and thus, performs $\approx 4\%$ balanced accuracy worse than the baseline with 65% balanced accuracy. The complete definition of the best processing pipeline found can

be found in Appendix D.4

Discussion The results of this experiment show, that the approach presented in this thesis is able to optimize the structure of the processing pipeline for more than one algorithm and the hyperparameters in parallel. Figure 4.9 shows that during the optimization process performance results comparable to the other experiments are achieved. But Figure 4.10 shows, that the found processing pipeline is overfitted to the optimization dataset and thus, does not generalize well for data from other sessions and subjects. This is the same behavior as seen in Experiment 4.4. Experiment 4.5 and this experiment yielded the same structure of the processing pipelines. But the pipeline of Experiment 4.5 (Appendix D.3) was not overfitted to the optimization dataset and achieved better performances than the baseline. Thus, a comparison of the two pipelines should give an indicator, why the processing pipeline found in this experiment is worse than the baseline. When comparing these two pipelines, three main differences come to attention. First, the regularization parameter is chosen different. In Experiment 4.5 this parameter is chosen as ≈ 0.22 and in this experiment it is much smaller with a value of ≈ 0.001. Additionally, the second and third differences are in the values of `segment_width` and `stepsize`. In Experiment 4.5, these have been chosen to be the same as in the baseline with a segment width of 400 and a stepsize of 120. In this experiment, a segment width of 200 is chosen and a stepsize of 20. These three changes are the main differences between the two processing pipelines. Thus, these changes are most likely the main reason for the overfitting of the processing pipeline.

4.7. Full optimization without knowledge

As the last experiment, the evaluation of the optimization process without any **domain expert** knowledge will be done. This experiment simulates the usage by a non-expert. The user gets input data of a known type but does not know a good classification pipeline to use with this type of data. Thus, the user creates test data from the input data and afterwards uses the described approach to obtain a good processing pipeline to classify the data.

Setup The setup of this optimization task is as minimal as possible. Because an average user of the software does not have much knowledge about the input data, only the required information to start the optimization has been given. The software

does then search through the whole search domain. This resulted in over four million processing pipelines that would have to be tested. Because only a limited amount of time and resources exist for the evaluation of the approach, a `whitelist` containing 39 algorithms has been used. This `whitelist` has been constructed by including only these algorithms are commonly used. Thus, all highly specialized algorithms have been left out. This was done by looking at the names of the algorithms and including these algorithms that have been heard of by the author of this thesis. Using this `whitelist` reduces the size of the search domain, as from only 39, instead of 295, algorithms processing pipelines can be constructed. This results in approximately 29.000 processing pipelines that need to be tested. Appendix C.6 shows the complete optimization task with all algorithms that can be chosen from. Additionally, to save resources, the `max_loss` of a pipeline that is required has been set to -0.6. Thus, after 100 evaluations all processing pipelines with a performance worse than this will not be evaluated any further, because usually 60% balanced accuracy is not an acceptable performance. Especially for EEG-Data as we know that performances of about 85% are achievable. Because the number of processing pipelines that need to be tested is nevertheless very large only 1000 evaluations will be done. The other experiments have done 2000 evaluations per processing pipeline but as they have shown that no significant improvement has been made after the first 1000 evaluations this number has been chosen for this experiment.

Results The results presented here are intermediate results. The experiment was able to complete $\approx 22\%$ of its work because of a memory leak somewhere in the software. This memory leak has not been found by now. As neither the developers of pySPACE,nor the author of this thesis know the exact cause of the leak, this error would require massive introspection which conflicts with the time constraints of this thesis. Because of this, the error is left in the software and the intermediate results achieved until the software crashed will be presented.

Figure 4.11 shows the performance results of the processing pipelines evaluated up to the time when the experiment was crashing. The best performance found this time is 87.8% balanced accuracy. The pipeline that achieved this performance can be found in Appendix D.5. Like in Experiment 4.5, the processing pipelines can be split into groups. But in this case, only two groups can be defined. The first one with processing pipelines with performances up to $\approx 60\%$ balanced accuracy and the second one with a performance range from $\approx 70\%$ to $\approx 85\%$ balanced accuracy. As the optimization task had defined a minimum performance of 60% all processing pipelines above this

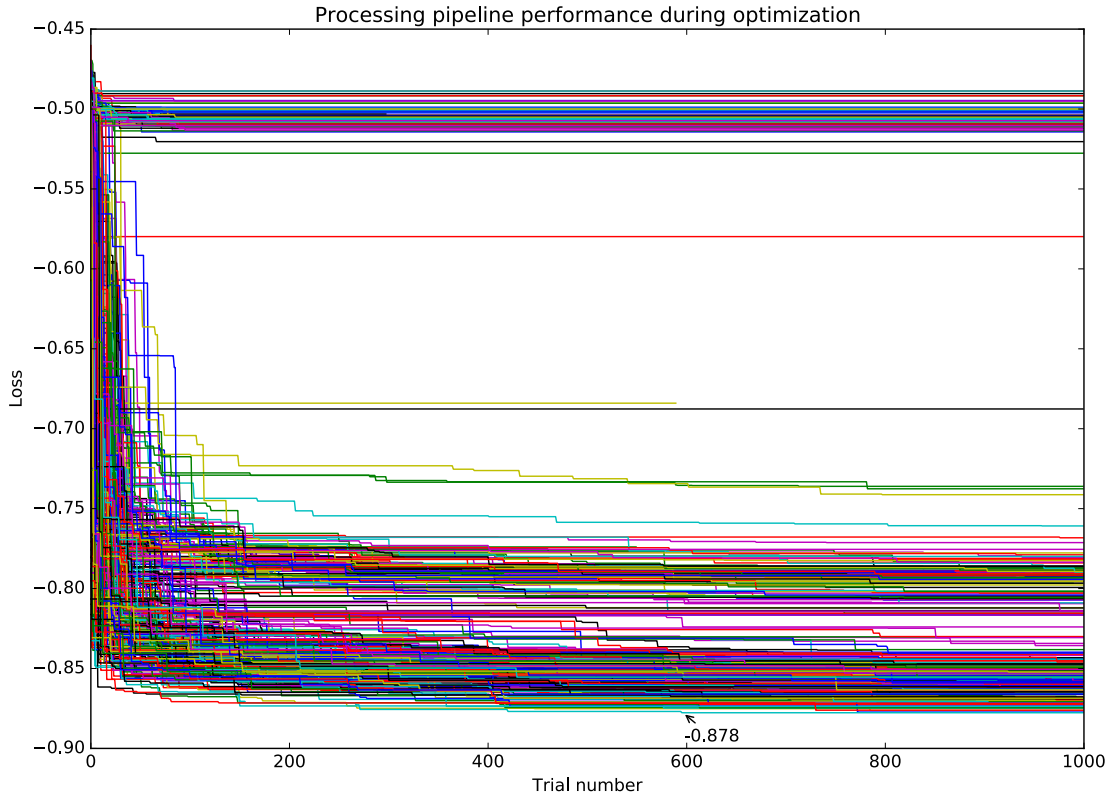


Figure 4.11.: Optimization performance of Experiment 4.7

performance had not been fully tested, but instead they have been tested for only 100 evaluations and afterwards contain the best found performance up to this evaluation. Figure 4.12 shows the performances of the best processing pipeline found so far. Like for every experiment the first plot shows the performance of an evaluation on the same recording session that has been used for training. In this case, the average performance is at about 68% balanced accuracy. This performance is way lower than the performance of the baseline with $\approx 83\%$. Thus, in this scenario, the processing pipeline found by the approach, described in this thesis, is $\approx 15\%$ balanced accuracy worse than the baseline. This repeats for the other evaluations. If evaluated on another session from the same test subject, the average performance decreases to $\approx 51\%$ balanced accuracy. In this case, the pipeline performs even $\approx 20.5\%$ worse compared to the baseline with $\approx 71.5\%$ balanced accuracy. If the found processing pipeline is evaluated on the other subjects, the prediction performance even decreases to $\approx 50\%$. Again, when compared to the baseline, the performance is $\approx 15\%$ worse.

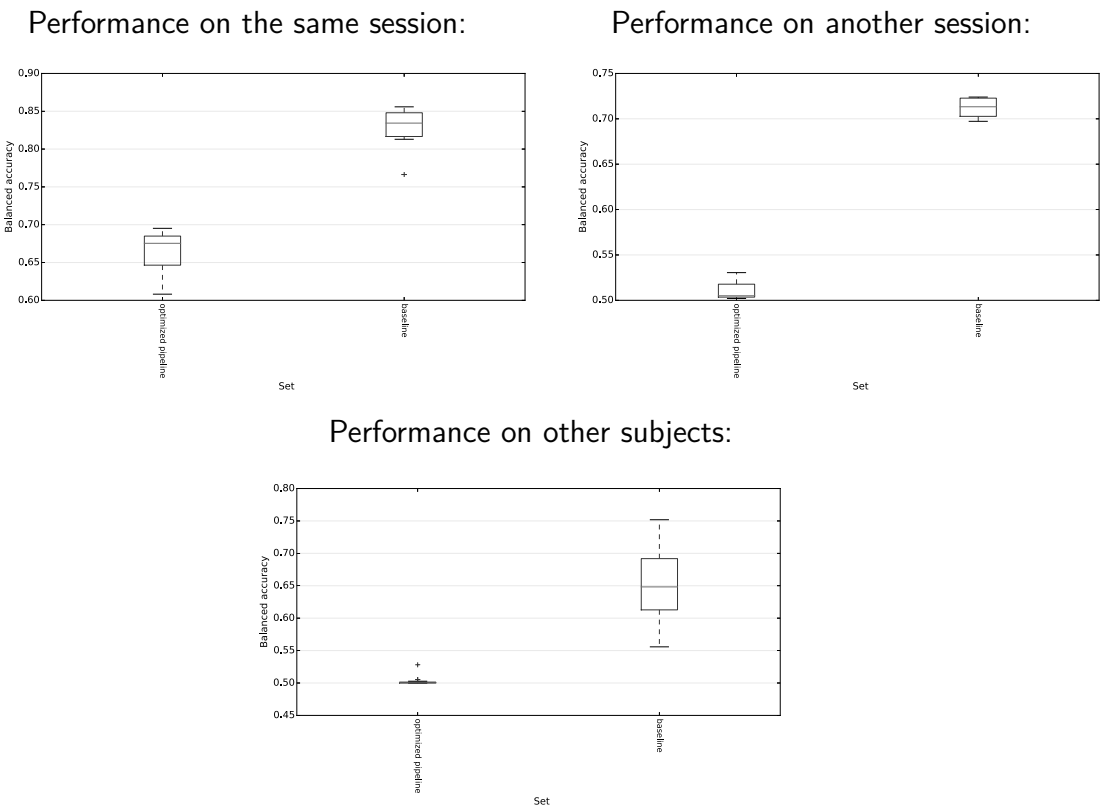


Figure 4.12.: Evaluation performance of Experiment 4.7

Discussion Besides the fact that this experiment could not be completed, the performance values of the best processing pipeline found before crashing show the great importance of a good variance in the training dataset. The found pipeline outperformed the baseline during the optimization process but it is fitted so much on the training data that it does not scale to be used for general classification of these EEG data. During the evaluation on the other subjects, it achieved a performance of $\approx 50\%$ in average. This is the performance of a random guessing. This shows that the processing pipeline did not learn the correct features at all. This theory is also supported, by the fact, that the resulting pipeline uses the `LibSVM` classifier as the baseline, but uses a `TimeDomainFeature` generator. This feature generator uses samples of the input data directly as feature vectors for the classification. By doing this on a EEG dataset which has a low signal-to-noise ratio, it is most likely that the classifier will learn to classify the noise. But as the noise is not directly connected to the target signal, the classification performance is stuck at the level of guessing when the noise changes.

4.8. Summary

The experiments presented in this chapter proof that the software is working as expected and is able to optimize the structure of a processing pipeline as well as the hyperparameters. The knowledge of **domain experts** has a great impact in the optimization speed as the size of the search domain will be reduced. But to achieve state of the art processing performance, it is not required. In contrast, the **algorithm experts** knowledge is very important to be able to optimize the hyperparameters of the different algorithms and achieve good performances.
This chapter also showed that, at least in theory, the approach presented in this thesis can be used without any **domain expert** knowledge and creates the search domain dynamically depending on the optimization task. But as pySPACE is a very large framework, including many different algorithms, this requires a large computation backend. If a single evaluation of one processing pipeline takes about 15 seconds, the complete evaluation of Experiment 4.7 on a cluster with 2000 parallel jobs would require two and a half days. Which is acceptable in most cases. But as the cluster available for these experiments could run only 150 jobs in parallel the evaluation would have required 33.5 days. Thus, a large cluster or at least a minimum amount of knowledge is required to achieve a termination within an acceptable amount of time.
On the other hand, the experiments also showed that good performance results can be achieved really fast, even if the optimization process has not finished yet. Therefore, a user can start the optimization process and get good results fast which she or he can then use while the optimization process is running.
Last but not least, this experiments showed the importance of the right choice of the training data. A large training data set is always desired, but if the samples in the training set are too similar (low variance) than the processing pipeline might get overfitted to these or might even learn to rely on features that are not available in other data sets. The baseline has been optimized using training and evaluation data from different subjects to achieve a good scalability [16]. By doing the same thing during the optimization process, a more robust processing pipeline might be created. But as already mentioned, this would exceed the resources and time available for this thesis.

5. Conclusion

As shown in Chapter 4, the approach presented in this thesis is able to optimize the structure and the hyperparameters of processing pipelines at the same time. Komer et al. [17] and Thornton et al. [25] have presented similar approaches to this problem with equal success. But in contrast to these two approaches the approach presented here does not use a handcrafted search domain for only a few selected algorithms. Instead it creates the search domain dynamically depending on the algorithms available on the installation and usable for a concrete optimization task. This requires a new approach on how the knowledge about parameter distributions can be made explicitly. As described in Section 3.2.1, this has been accomplished by using the **algorithm expert's** knowledge and made available by decorating an algorithm using distributions for the parameters. Additionally, in pySPACE knowledge about the input and output types of each algorithm is available too. This *base* knowledge allows the method, presented here, to dynamically search for possible processing pipelines, gather the hyperparameters required for these, and define the search domain. This approach is independent of the desired complexity of the processing pipeline. This approach is able to combine any number of algorithms creating processing pipelines of any length. Currently the software allows just a single preprocessing algorithm per processing pipeline, but without this explicit restriction a processing pipeline with any number of pre- and postprocessing steps can be generated. This makes the optimized processing pipelines usable for complex data sets like the EEG data used in Chapter 4.

In contrast to the approaches of Komer et al. and Thornton et al., the software described in Chapter 3 does not merge the search domains of each processing pipeline into a single search domain. The merge of the search domains requires the introduction of new hyperparameters to define which processing pipeline is chosen. Additionally, this requires to structure the search domain according to these new hyperparameters as not all hyperparameters are valid for all processing pipelines. Thus, this would increase the complexity of the search domain. Additionally, by separating the search domain for every processing pipeline, this knowledge can be stored to the hard disk and reused for another optimization task on similar (or even the same) input data. Because of this, the search domain of each processing pipeline is kept separate.

As the experiments in Chapter 4 have shown this approach is highly prone to overfitting the processing pipelines to the input data set. This could be circumvented in several ways. The simplest possible approach for this is to assure a good variance in the training data. If the training data contain enough different trials, the processing pipeline is not able to infer features that cannot be relied on. It might not always be possible to assure

the variance of the training data because this requires a large training data set. If only few data are available more elaborated approaches are required. This is left as future work and possible approaches to this problem will be presented in Section 5.1.

5.1. Future work

Because of time and resource constraints some aspects have been left out in this thesis that should be regarded in future work that is based on this thesis. These include both improvements of the optimization process and new features that can be included to improve the usability of the software.

5.1.1. Additional pySPACE extensions

The main problem the approach presented in this thesis tries to solve is the intelligent reduction of the search domain. To do this, additional knowledge in the form of **algorithm experts** and **domain experts** knowledge is required.

Hierarchical type system in pySPACE As described in Section 3.2.1 one extension to pySPACE has already been created. A few more extensions are imaginable to reduce the size. If pySPACE supports a hierarchical type system, algorithm experts would be able to define a domain the algorithm is usable in. For example an algorithm might take time continuous data as input, but this does not define whether EEG, audio signals, *global positioning system (GPS)* tracking, movement data, or any other type of time continuous signal can be used. Therefore, the algorithm needs to be tested for time continuous input data of each domain the optimization software is used for. But if the algorithm expert would be able to define for example that the input data need to be time continuous EEG signal, the optimization software knows that this algorithm is not suitable for processing audio signals. Thus, the search domain would be reduced by excluding algorithms that are not capable of processing input data from other domains. But as many algorithms might be able to process input data from multiple domains, these types need to be structured hierarchically. By defining a hierarchy, this type system is extendable, because new (sub-) domains can be defined in the system. Additionally, a hierarchy allows the already existing algorithms to be reused without the need to alter them. If, for example, an algorithm is defined to be able to process EEG data and a new

domain of the EEG recordings during a movement is defines as *EEG-Movement* data. Then a hierarchy allows the first algorithm to be reused in the domain of movement recordings without explicitly adding this domain to the algorithm.

Dynamic task variant determination This extension can be adapted to other problems like defining for example whether a unary, binary class, or multi class classification should be done. For each of these variants of the classification task, different algorithms are suitable. But they all do the same task: Classifying data. The simplest solution to this problem would be to create multiple different `ClassificationTask` implementations for each of them. Each of these tasks can than exclude algorithms not suitable for the task. But this would require the user to know what kind of classification should be done and requires a new Task implementation every time a new variant is added.
Another approach to solve this problem would be to infer the type of the classification from the input data. Either by the type of the input data or by inspecting them. If more than two class labels are available in the input data, a multi-class classification needs to be done. If only one labels is available a unary classification is required. But if two class labels are available it depends on the distribution of the labels. If the dataset is highly imbalanced, it might be better to do a unary classification instead of a binary one. The same approach can be used for clustering and regression mechanisms. A clustering or regression is a different task than a classification and thus, has its own `Task` implementation, but depending on the input data different variants can be done.

Algorithm constraints The third and last possible extension, described in this section, is the ability to define constraints on algorithms. If the **algorithm expert** is given the opportunity to define which types of algorithms can be used after another algorithm this would reduce the number of combinations of the algorithms and thus, the size of the search domain. Additionally, this would eliminate the restriction of the software to allow only one algorithm of each type per processing pipeline. By including this, the algorithm expert can define that after a frequency filter, no other frequency filter can be used. But a normalization, for example, would be possible. This would enable the software to allow multiple pre- and postprocessing algorithms in a single processing pipeline without enlarging the search domain exponentially.

5.1.2. pySPACEOptimizer extensions

Another, completely different, field of future work based on this software is the extension of the software itself to enable new applications and knowledge sources.

Ensemble learning A current trend in machine learning is the so called "ensemble learning". This approach uses multiple layers of classifiers (or other processing algorithms) and combines them to a new one. For example, a processing pipeline might combine the prediction results of different classifiers and use these as a new input to another classifier. One example for such an approach are *random forests*. A random forest consists of many *decision trees*. Each of them use a random subset of the features available to classify the input data. These class labels are then taken and a new "classifier" uses them to produce the final class label. In a random forest, this last classifier is a simple majority vote [9].
The inclusion of this approach in the optimization process would lead to a complete new problem, as the number classifiers to use in the ensemble as well as which ones needs to be optimized in parallel with the hyperparameters of them. This problem can be solved by implementing a new node in pySPACE. This node uses the k best processing pipelines of a performance optimization already run and ensembles the results of these into a new feature vector. By doing this, two independent runs can be made. The first run will optimize the hyperparameters of each base classifier. Afterwards, the classifier is replaced by the new node and a second optimization task is started optimizing the k of the new node to determine the required number of classifiers.

Optimization of data dependent hyperparameters Many algorithms define hyperparameters that actually need to be determined new for every training data, as they are highly dependent on the input data. An example of such a hyperparameter is the regularization parameter of a SVM. Whereas the kernel should stay the same for all input data, the regularization needs to be retrained on every training dataset. Currently, the optimization process determines these parameters only once for all training data. If the **algorithm expert** is able to define these parameters as being input data dependent, they could be optimized during training of the processing pipeline. Currently pySPACE supports this by using a `grid search`. The problem with this is, that a grid search requires parameter ranges to search in. These need to be defined by the user and thus, the optimization software would not be usable by non-experts. To avoid this, two different approaches can be thought of. First of all, a new `optimization node` can be

implemented in pySPACE. This node takes a single algorithm as input and knows the hyperparameters of this algorithm that should be optimized during the training. As this node is part of the processing pipeline, it is able to issue a new optimization task during the training of the processing pipeline. Depending on the number of hyperparameters, this might require several hundred evaluations to be done during the training and thus, might require a lot of time. To solve this, and re-use the grid search already implemented in pySPACE, another approach could be used. During the optimization process the approach trains a prediction model containing evidence about the parameter distributions. If an algorithm defines input data dependent parameters, then a normal optimization is done. Afterwards, the model can be used to define confidence intervals for these parameters. From these intervals a new parameter range can be sampled which is then used in the grid search during the training.

The internet as expert Last but not least, another extension to the software would involve the world wide web. This can be used to eliminate the requirement of **algorithm experts** and speed up the optimization process by including **domain expert's** knowledge received from the web. This would require a knowledge of the software about the domain the input data are from. Using this domain the software could then search through the web for processing pipelines known to work well in this domain or to search for optimized parameter ranges to use for the algorithms. Thus, some kind of structured web database or a crawler using natural language processing is required to search the web for the required information. Additionally, the software may build up an own database to use for this. Each time the software is used, it saves the results obtained from the optimization process in a database. If input data of the same domain should be processed again, the stored configuration can be tested first before searching for other possible processing pipelines.

All these extensions allow the approach to reduce the size of the search domain, optimize new fields of problems and to use additional knowledge sources. These are the three main extension points currently available to extend the approach presented in this thesis.

A. Technical details

A.1. Implementation details of NodeListGenerator

The `NodeListGenerator` class is one of the core features of this software. It is used by the optimizer to generate all possible processing pipelines that might, in theory, be able to process the given input dataset. To efficiently generate all possible processing pipelines, this class implements the constraints and type checking methods defined in Section 3.1.

This class defines a `__iter__` method. This method gets called by the Python interpreter when iterating over the generator object. This method then uses a depth-first search and backtracking to generate and yield all pipelines one by one. In Python no arrays with a fixed size exist. Instead, a type called `list` exist that is extended as needed. When using backtracking to search for all processing pipelines this results in many unnecessary copying actions as for every algorithm that get's tested, the list needs to be copied, the algorithm checked and appended. This slows down the generation. Therefore, a third-party library called *NumPy* has been used. Numpy is a library for numerical calculations on vectors and matrices and is needed by pySPACE anyway. Additionally, this library brings in fixed size arrays that don't need to be copied when used correctly. Because the user defines the maximal length of the pipelines to generate beforehand, an array of that size can be created together with a counter pointing to the first element of the array that does not store a valid algorithm. Initially this counter will be set to 0 as no valid algorithm has been found. The `__iter__` method then calls a second method called `_make_node_list` which implements the actual depth-first search with backtracking.

During generation this method will call itself recursively and depending on the given counter value it knows which type of algorithm to search for. If the counter is 0, no valid algorithm has been found and an algorithm of type `source` is required. This algorithm can either be defined by the user using the optimization task or chosen automatically. If given by the user, the method will select the given algorithm and proceed. Otherwise it iterates over all algorithm of type `source` and capable of loading the given input dataset and select them one by one. In either case it will insert the type of the algorithm into a second array and call itself again with the counter increased by one.

If the counter is larger then 0 but not the maximal length of the pipeline, the algorithm detects the output type of the last algorithm and iterates over all algorithms capable of taking this output data as its input data. Before selecting one of these algorithms it now checks whether the type of the algorithm is already contained in the pipeline by checking for the type in the second array. If the type is not contained, the algorithm is selected

and the type inserted into the second array. Now the output type of this algorithm is determined and checked whether it matches the input type required by the `sink` node. If it does match, the `sink` node gets appended. Before the pipeline is yielded as a possible pipeline, it is checked that all required types of algorithms are used inside the processing pipeline. This is done because, depending on the optimization task different algorithm types need to be present in a valid processing pipeline. For example a processing pipeline for a classification task without a classifier does not make any sense. If all required node types are used, the found processing pipeline is yielded. If not the method calls itself again with an increased counter.
If the counter equals the maximal length of the pipelines that should be generated, no additional algorithm can be added to the processing pipeline and it breaks the generation and starts backtracking by returning to the caller. This caller then selects the next algorithm of his list and tries to generate pipelines using this one.
This procedure continues until the top-level caller, the one that selected the `source` node has processed all of its algorithms. This guarantees that the whole search space of possible processing pipelines get searched.

A.2. Implementation details of PerformanceGraphic

The `PerformanceGraphic` can be used to give the user a quick overview of the performances achieved by the already evaluated processing pipelines and their corresponding hyperparameters. Every time an evaluation of one pipeline is finished and the performance is measured, it gets added to an instance of this class which then checks if the given performance is a new best performance for the given processing pipeline. If the current performance is better than the last found best performance, the number of the evaluation and the performance get added to a list. For example if in evaluation 1 a performance of 0.55 has been found and in evaluation 10 a performance of 0.8, and the evaluations $2-9$ had a performance worse than 0.55, the list then contains $[(1, 0.55), (10, 0.8)]$. These best performances form a path of the best found performances at each evaluation. The graph is then calculated by interpolating between the entries of the performance list. To keep track of the total number of evaluations done, a second counter is used. Every time a new performance measurement gets added, this counter is checked whether the number of the new evaluation is larger than the counter and, if so, it gets set to this number. By checking this, a parallel processing of the evaluations is possible in which the performance results might not be added in the order

of the evaluation numbers.
The calculation and plotting of the performance graphs thereby is done in a separate thread to keep the load of the main process as low as possible. Therefore, this thread runs every 60 seconds, collects the performance paths and interpolates them to graphs. This is done by starting with the first tuple of the path and iterating over all evaluation numbers. If the number of the evaluation equals the next segment of the path, this segment is used, otherwise the evaluation gets added with the performance of the current segment. This results in a list of all evaluations with the current best found performance. This list is then plotted.

A.3. Implementation details of NodeParameterSpace

The `NodeParameterSpace` class represents the hyperparameter space for a single algorithm. Thus, it implements a method called `parameter_space` which returns a dictionary with the name of each parameter and its distribution of it. The distribution of a hyperparameter can either be defined by the **algorithm expert** using the decorators described in Section 3.2.1, the **domain expert** using the different methods described in Section 3.3 to set them in the optimization task description, or they can be inferred from the default values of the algorithm. If the given algorithm is not decorated, the standard implementation of this method will collect the parameters of the constructors of every base class of the algorithms implementation and create a set of parameters from it. This needs to be done for every class defined in the hierarchy because often parameters of the base classes are just passed as a Python special argument called `**kwargs`. This argument stores a dictionary of all parameters that has been passed to the constructor but is not defined in it. Therefore, not all possible parameters can be determined on the most derived class itself and need to be searched for in the hierarchy. The set of parameters and defaults is then converted into a dictionary of parameter names and distributions. The distributions are created according to the type of the default values. If the default value is a Boolean a `BooleanParameter` is created, if it is a floating point number, a `NormalParameter`, for an integer, a `QNormalParameter`, and if the default value is a list a `ChoiceParameter` is created. Every other type is ignored, because the software can not define a distribution for these parameters automatically and therefore the default values are used.
This method should be overwritten in subclasses for the concrete optimizer. These implementations then call the base method to get a parameter space and then can map the

distributions to the distributions the optimizer supports. For example when using the Hyperopt optimizer this results in a $1:1$ mapping from the pySPACE distributions to the according Hyperopt distribution. This method then returns a dictionary containing the parameter names and the distributions required for the optimizer.
This approach enables the software to optimize algorithms that have not been decorated by an *algorithm expert* at all. So no additional effort is required to use the software with the algorithms already defined in pySPACE. But because only `(Q-)NormalParameter` distributions and choices can be detected, this parameter space will most likely not be optimal. Because of this, new algorithms should always be decorated when writing them. The existing algorithms can easily be decorated by **algorithm experts** one after another as needed. After some time of usage, all algorithms that get used are decorated and can be optimized well.

A.4. Implementation details of PySPACEOptimizer

The `PySPACEOptimizer` class implements an abstract base class that new optimizers can extend to be usable in the *pySPACEOptimizer* framework. To do this, new optimizers need to implement two methods. The first method is called `create_node`. This method gets passed the name of an algorithm and constructs an instance of the parameter space for it. This method is required, because every optimizer brings it's own `NodeParameterSpace` implementations, that will represent the hyperparameter space in the format the optimizer requires it.
The `PySPACEOptimizer` uses this method another method called `_generate_node_chain_parameter_spaces`. As the name already denotes, this method is used to generate all possible processing pipelines. To do this, it uses the `NodeListGenerator` to generate a list of algorithm names. These need to be converted to instances of the `NodeParameterSpace` using the `create_node` method described above. Afterwards, a `NodeChainParameterSpace` object gets instantiated using this list of `NodeParameterSpace` objects. This node chain then gets yielded to the optimizer to be evaluated.
Last but not least, the second abstract method defined is called `optimize`. This method implements the actual optimization process for the custom optimizer. An implementation of this method needs to call the `_generate_node_chain_parameter_spaces` method to receive the possible processing pipelines and afterwards optimizes them by sampling hyperparameter settings and

evaluating them.
To handle the results of the evaluations and pass them to the `PerformanceGraphic` an instance of this class instantiates a queue that is used by the custom optimizer to pass the evaluation results. To read the results from this queue a separate thread is started. This thread waits for results, adds them to the `PerformanceGraphic`, and checks if they are better than the last best values found. If the current result is better, the loss, the pipeline, and the parameters used are stored in a variable. Additionally, this result gets stored to the disk as a YAML file executable by pySPACE. To assure that all results are processed even if the optimization itself is finished, a sentinel value is inserted to the queue when the thread should stop. If this value is received from the queue, it stops its execution.

A.5. Parameter descriptions of the Task

Name	Description
name	The name of the optimization task to identify and store results.
input_path	The (relative) path to the input dataset to use.
evaluations_per_pass	The number of evaluations that can be done in parallel before new hyperparameter settings need to be sampled.
optimizer	The name of the optimizer to use for the task.
max_pipeline_length	The maximal length of the processing pipeline that will be generated for optimization.
passes	The number of optimization passes to do. This defines the number of times hyperparameter settings will get sampled from the model. The total number of evaluations is defined by passes $\times$ evaluations_per_pass.
metric	The name of the metric to use for the performance evaluation.
source_node	The name of the source node to use.
sink_node	The name of the sink node to use. This needs to be a sink node analyzing the performance of the pipeline. Typically leave this the default (PerformanceSinkNode).
max_parallel_pipelines	The maximal number of pipelines that will be optimized in parallel.

Table A.1.: All fields used by the base task

A.6. Implementation details of HyperoptOptimizer

The `HyperoptOptimizer` implements an optimizer for the `pySPACEOptimizer` framework based on the Hyperopt library. Thus, it inherits from the `PySPACEOptimizer` (Appendix A.4) base class and implements the two abstract methods called `optimize` and `create_node`. The second method is used during the generation of the processing pipelines to map the parameter spaces of the different algorithms to parameter space representations for Hyperopt. This method checks whether the type of the algorithm is a sink, a source, or any other node type and creates depending on this type a

ClassificationSourceNodeParameterSpace, a ClassificationSinkNodeParameterSpace or a HyperoptNodeParameterSpace object for it. As described above these objects will then be combined to a NodeChainParameterSpace object in the _generate_node_chain_parameter_spaces method.
This method is then used in the optimize method. This method creates an OptimizerPool and, to make inheritance more easy, then calls a method called _do_optimization. This method does the real optimization. It has been created because during development and testing a second optimizer class called SerialHyperoptOptimizer has been created which creates an OptimizerPool with a single process to force the serial evaluation of all processing pipelines. To let these two classes reuse as much code as possible, the real implementation of the optimization process has been exported to the _do_optimization method. It iterates over the processing pipelines yielded by the _generate_node_chain_parameter_spaces method and adds every processing pipeline to the OptimizerPool. After the addition this method waits for the pool to process the pipelines,checks if any errors occurred, and then returns. When processing one of these pipelines the pool will call a static function called optimize_pipeline this method will create a pySPACE backend instance to execute the processing pipeline evaluations. Afterwards, it creates a PersistentTrials object for this processing pipeline to store the evaluation results and then starts iterating over the *passes* that should be done. For every pass a progress bar is used to let the user keep track on the progress of the evaluation. As many processing pipelines might be evaluated in parallel, this progress bars do not get printed to the screen but instead get logged to a file using the logger of the processing pipeline. For each pass the minimize method of the PersistentTrials object is called with the number of evaluations to do. This method will then sample different hyperparameter settings for these evaluations and execute them. To execute the pipeline and with the given hyperparameters and analyze the results a seconds static method __minimize is used. This method gets the processing pipeline to evaluate, the backend to use for the evaluation, and the hyperparameters to evaluate the processing pipeline with. It then executes the given pipeline using the backend and the hyperparameters and afterwards searches the result directory for a file called results.csv. This file stores the performance results of the processing pipeline. It reads this file, searches for an entry with the metric defined in the optimization task and takes this as the performance. Because Hyperopt minimizes a function that returns a loss instead of maximizing a function with a performance. Depending on the metric chosen the performance value

needs to be multiplied with -1 to transform it into a loss. This is controlled by the `is_performance_metric` parameter of the optimization task. If this parameter is true, the software assumes that the metric is a performance value, that needs to be maximized and multiplies it with -1 to turn it into a loss that needs to be minimized. This loss is then returned to the `PersistentTrials` object and from there on propagated through the software back to the main optimizer where it is stored to the performance graphic and checked for being the best loss found.

B. Logging configuration file example

```
version: 1


formatters:
  console_formatter:
    format: "%(levelname)s: %(message)s"

  file_formatter:
    format: "[%(asctime)s.%(msecs)03d:%(levelname)10s][%(name)s] %(message)s"
    datefmt: "%d.%m.%Y %H:%M:%S"


handlers:
  console:
    class: logging.StreamHandler
    formatter: console_formatter
    level: INFO
    stream: ext://sys.stdout

  file:
    class: logging.handlers.RotatingFileHandler
    formatter: file_formatter
    level: DEBUG
    filename: pySPACEOptimizer.pylog

  pipeline_handler:
    class: logging.handlers.RotatingFileHandler
    formatter: file_formatter
    level: DEBUG
    filename: pipeline_output.pylog

  pipeline_error_handler:
    class: logging.handlers.RotatingFileHandler
    formatter: file_formatter
    level: WARNING
    filename: pipeline_errors.pylog
```

```
  null_handler:
    class: logging.NullHandler

loggers:
  "":
    level: ERROR
    handlers:
      - null_handler

  pySPACEOptimizer:
    level: INFO
    propagate: False
    handlers:
      - console
      - file

  pySPACEOptimizer.optimizer:
    level: INFO
    propagate: False
    handlers:
      - console
      - file

  pySPACEOptimizer.pipeline:
    level: INFO
    propagate: False
    handlers:
      - pipeline_handler

  pySPACEOptimizer.pipeline_errors:
    level: ERROR
    propagate: False
    handlers:
      - pipeline_error_handler
```

C. Task descriptions

C.1. Task description for Experiment 4.2

```
# concatenation of algorithms
type: node_chain
# path relative to storage
input_path: eeg_subject2_session2
runs : 2 # number of repetitions to catch randomness
node_chain: # the algorithms
  - node : TimeSeriesSourceNode
  - node : xDAWN
    parameters :
      erp_class_label : "Target"
      store : False
      visualize_pattern : False
      retained_channels : 8
  - node : Local_Straightline_Features
    parameters :
      segment_width : 400
      stepsize : 120
      coefficients_used : [1]
  - node : Gaussian_Feature_Normalization
  - node : Grid_Search
    parameters :
      parallelization:
        processing_modality : 'serial'
        pool_size : 1
      optimization:
        ranges : {~~log_complexity~~: [0, -1, -2,-3,-4,-5,-6]}
      validation_set :
        split_node :
          node : CV_Splitter
          parameters :
            splits : 5
            stratified : True
      evaluation:
        metric : "Balanced_accuracy"
        std_weight: 0
        performance_sink_node :
```

```
          node : Classification_Performance_Sink
          parameters :
            ir_class : 'Target'
            measure_times : False
            calc_AUC : False

      variables: [~~log_complexity~~]
      nodes :
        - node: LibSVM_Classifier
          parameters :
            complexity : "eval(10**~~log_complexity~~)"
            class_labels : ['Standard', 'Target']
            weight : [1.0, 5.0]
            kernel_type : 'LINEAR'
        - node: Threshold_Optimization
          parameters :
            metric : 'Balanced_accuracy'
            class_labels : ['Standard', 'Target']
  - node : Classification_Performance_Sink
    parameters :
      ir_class : 'Target'
      dataset_pattern: "set_subject_session"
```

C.2. Task description for Experiment 4.3

```
# The name of the optimization task.
# Every optimization task needs to have a name
# to make it distinguishable from each other.
name: single_pipeline
# The type of the optimization task.
# This option tells the software which
# task class should be instantiated.
type: classification
# The (relative) path to the input dataset.
# The path given here defines a relative
# path to the input data set. The base
# of this path is the pySPACE storage
```

```
# directory.
input_path: eeg_data_reduced
# The optimizer to use.
# This parameter defines which optimizer
# should be used. Currently only the
# HyperoptOptimizer is available, but
# later on, new optimizers might be added
# to the software.
optimizer: HyperoptOptimizer
# The name of the source node to use.
# If this parameter is given, the source node
# is fixed and not automatically chosen by the
# software.
source_node: TimeSeriesSourceNode
# The name of the sink node to use.
# This node needs to measure the performance
# of the processing pipeline. Currently only
# the PerformanceSinkNode is implemented in
# pySPACE and able to measure the performance.
# Thus, leave this parameter like this.
sink_node: PerformanceSinkNode
# The metric to use for the performance
# measurement. This can be one of the
# different metrics defined in pySPACE.
# For more details on the available
# metrics please see the pySPACE documentation.
metric: Balanced_accuracy
# Defines whether the given metric is a
# performance metric or not. Depending on this,
# the software decides, whether the target
# function should be maximized or minimized.
is_performance_metric: true
# The maximal length of the pipeline that
# will be generated.
# Only processing pipelines with less or
# equal to this number of nodes will be
# generated and optimized.
```

```
max_pipeline_length: 10
# The list of algorithms allowed in this
# optimization task.
# All processing pipelines will be
# generated only using the algorithms
# defined in this list.
whitelist:
  - TimeSeriesSourceNode
  - XDAWNNode
  - LocalStraightLineFeatureNode
  - GaussianFeatureNormalizationNode
  - LibSVMClassifierNode
  - ThresholdOptimizationNode
  - PerformanceSinkNode
# The list of algorithms required in
# each pipeline.
# Each processing pipeline, that will
# be generated must contain the
# algorithms defined in this list.
forced_nodes:
  - TimeSeriesSourceNode
  - XDAWNNode
  - LocalStraightLineFeatureNode
  - GaussianFeatureNormalizationNode
  - LibSVMClassifierNode
  - ThresholdOptimizationNode
  - PerformanceSinkNode


##############################################
# The following parameters depend on the
# type of the task selected. In this case
# a classification should be done. Thus,
# the required parameters for a classification
# are defined here.
##############################################
# The class labels available in the input data.
```

```
# This parameter is automatically given to all
# nodes that have a class_labels
# hyperparameter.
class_labels: [Standard, Target]
# The main class the user is interested in.
# This parameter might be required for the
# performance measurement, as it defines
# the class of "positive" samples.
main_class: Target


################################################
# The following parameters depend on the
# optimizer chosen. They can be used to
# customize the behavior of the optimizer
# and the optimization process.
################################################
# The number of hyperparameter settings that
# will be sampled in evaluated in a single
# pass. After each pass, a new bunch of
# hyperparameters are sampled and evaluated.
evaluations_per_pass: 10
# The number of passes to do.
# The combination of this parameter and
# the evaluations_per_pass parameter
# defines the total number of evaluations
# done.
passes: 200


################################################
# The following section defines domain expert
# knowledge about the hyperparameter
# distributions of the different algorithms.
# This section is optional and not required to
# optimize the hyperparameters of the
# pipelines.
```

```
##############################################
parameter_ranges:
    # Set the parameter ranges for the
    # xDAWN spatial filter node
    xDAWNNode:
        # The xDAWN Node does not use
        # main_class as the parameter for the
        # "positive" class. Thus, give the
        # "positive" class label here.
        erp_class_label : ["Target"]
        # Do not store and visualize
        # the data
        store : False
        visualize_pattern : False
        # Retain only 8 channels of all
        # available channels
        retained_channels : [8]
    # Set the parameter ranges for the
    # feature generator
    LocalStraightLineFeatureNode:
        # Fit the straight lines to
        # segments of 400ms length.
        segment_width: 400
        # And shift input window by
        # 120ms. This results in
        # overlapping windows to which the
        # straight lines get fitted.
        stepsize: 120
        # Define the coefficient of the
        # straight line that should be used.
        # 1 means that the slope is used as
        # feature. 0 would mean that the
        # offset would be used and
        # [0, 1] that both should be used.
        coefficients_used: [[1]]

    # Set the hyperparameters for the
```

```
    # threshold optimizer
    ThresholdOptimizationNode:
        # The threshold optimizer requires
        # the metric, that is used to measure
        # the performance, to optimize the
        # threshold according to this metric.
        metric : 'Balanced_accuracy'
```

C.3. Task description for Experiment 4.4

```
name: single_complete_optimization
type: classification
input_path: eeg_data_reduced
optimizer: HyperoptOptimizer

class_labels: [Standard, Target]
main_class: Target

source_node: TimeSeriesSourceNode
sink_node: PerformanceSinkNode
metric: Balanced_accuracy
is_performance_metric: true

evaluations_per_pass: 10
passes: 200

max_pipeline_length: 7
restart_evaluation: false

whitelist:
  - TimeSeriesSourceNode
  - XDAWNNode
  - LocalStraightLineFeatureNode
  - GaussianFeatureNormalizationNode
  - LibSVMClassifierNode
  - ThresholdOptimizationNode
  - PerformanceSinkNode
```

```
forced_nodes:
  - TimeSeriesSourceNode
  - XDAWNNode
  - LocalStraightLineFeatureNode
  - GaussianFeatureNormalizationNode
  - LibSVMClassifierNode
  - ThresholdOptimizationNode
  - PerformanceSinkNode
```

C.4. Task description for Experiment 4.5

```
name: multi_classifiers
type: classification
input_path: eeg_data_reduced
optimizer: HyperoptOptimizer

class_labels: [Standard, Target]
main_class: Target

source_node: TimeSeriesSourceNode
sink_node: PerformanceSinkNode
metric: Balanced_accuracy
is_performance_metric: true

evaluations_per_pass: 10
passes: 200
check_after: 100
max_parallel_pipelines: 20

max_pipeline_length: 7
restart_evaluation: false

forced_nodes:
  - TimeSeriesSourceNode
  - XDAWNNode
  - LocalStraightLineFeatureNode
```

```
  - GaussianFeatureNormalizationNode
  - ThresholdOptimizationNode
  - PerformanceSinkNode

blacklist:
    - LiblinearClassifierNode

parameter_ranges:
  xDAWNNode:
    erp_class_label : [Target]

  LocalStraightLineFeatureNode:
    segment_width: 400
    stepsize: 120
    coefficients_used: [[1]]

  ThresholdOptimizationNode:
    metric: Balanced_accuracy
```

C.5. Task description for Experiment 4.6

```
name: two_nodes
type: classification
input_path: eeg_data_reduced
optimizer: HyperoptOptimizer

class_labels: [Standard, Target]
main_class: Target

source_node: TimeSeriesSourceNode
sink_node: PerformanceSinkNode
metric: Balanced_accuracy
is_performance_metric: true

evaluations_per_pass: 10
passes: 200
check_after: 100
```

```
max_parallel_pipelines: 40

max_pipeline_length: 7
restart_evaluation: false

forced_nodes:
  - TimeSeriesSourceNode
  - XDAWNNode
  - GaussianFeatureNormalizationNode
  - ThresholdOptimizationNode
  - PerformanceSinkNode

blacklist:
    - LiblinearClassifierNode
```

C.6. Task description for Experiment 4.7

```
name: multi_nodes2
type: classification
input_path: eeg_data_reduced
optimizer: HyperoptOptimizer

class_labels: [Standard, Target]
main_class: Target

source_node: TimeSeriesSourceNode
sink_node: PerformanceSinkNode
metric: Balanced_accuracy
is_performance_metric: true

evaluations_per_pass: 10
passes: 100

max_pipeline_length: 7
restart_evaluation: false
max_parallel_pipelines: 50
check_after: 100
```

```
max_loss: -0.6

forced_nodes:
  - TimeSeriesSourceNode
  - ThresholdOptimizationNode
  - PerformanceSinkNode

whitelist:
    - LiblinearClassifierNode
    - InfinityNormFeatureNormalizationNode
    - MaximumStandardizationNode
    - PearsonCorrelationFeatureNode
    - XDAWNNode
    - QuadraticDiscriminantAnalysisClassifierNode
    - LinearDiscriminantAnalysisClassifierNode
    - TimeDomainFeaturesNode
    - GaussianFeatureNormalizationNode
    - SimpleDifferentiationFeatureNode
    - TimeDomainDifferenceFeatureNode
    - Simple5DifferentiationNode
    - RandomFeatureCompressionNode
    - OutlierFeatureNormalizationNode
    - FDAClassifierNode
    - CoherenceFeatureNode
    - LocalStandardizationNode
    - BayesianLinearDiscriminantAnalysisClassifierNode
    - CSPNode
    - FeatureNormalizationNode
    - AXDAWNNode
    - HemisphereDifferenceNode
    - StatisticalFeatureNode
    - PCAWrapperNode
    - EuclideanNormalizationNode
    - RmmPerceptronNode
    - FDAFilterNode
    - HistogramFeatureNormalizationNode
    - ClassAverageCorrelationFeatureNode
```

```
    - RMM2Node
    - PlattsSigmoidFitNode
    - ConstantChannelCleanupNode
    - PywtWaveletNode
    - LibSVMClassifierNode
    - ICAWrapperNode
    - STFTFeaturesNode
    - EuclideanFeatureNormalizationNode
    - LinearTransformationNode
    - LocalStraightLineFeatureNode

# Give a higher weight to the nodes known to work good.
# The weight of a node is a degree of belief of the user
# in a node. This weight will have impact on the order
# the processing pipelines will be generated (and thus,
# optimized). A higher weight means that processing
# pipelines containing these nodes will get generated
# first. As default, each nodes receives a weight of
# 1 / len(nodes).
node_weights:
    XDAWNNode: 100
    LocalStraightLineFeatureNode: 100
    GaussianFeatureNormalizationNode: 100
    LibSVMClassifierNode: 100
```

D. Optimization results

D.1. Optimized pipeline of Experiment 4.3

```
input_path: eeg_data_reduced
node_chain:
    -   node: TimeSeriesSourceNode
    -   node: XDAWNNode
        parameters:
            erp_class_label: ${XDAWNNode_erp_class_label}
            retained_channels: ${XDAWNNode_retained_channels}
    -   node: LocalStraightLineFeatureNode
        parameters:
            coefficients_used: ${LocalStraightLineFeatureNode_coefficients_used}
            segment_width: ${LocalStraightLineFeatureNode_segment_width}
            stepsize: ${LocalStraightLineFeatureNode_stepsize}
    -   node: GaussianFeatureNormalizationNode
    -   node: LibSVMClassifierNode
        parameters:
            class_labels: ${LibSVMClassifierNode_class_labels}
            complexity: ${LibSVMClassifierNode_complexity}
            epsilon: ${LibSVMClassifierNode_epsilon}
            kernel_type: ${LibSVMClassifierNode_kernel_type}
            max_iterations: ${LibSVMClassifierNode_max_iterations}
            max_time: ${LibSVMClassifierNode_max_time}
            nu: ${LibSVMClassifierNode_nu}
            offset: ${LibSVMClassifierNode_offset}
            ratio: ${LibSVMClassifierNode_ratio}
            tolerance: ${LibSVMClassifierNode_tolerance}
    -   node: ThresholdOptimizationNode
        parameters:
            class_labels: ${ThresholdOptimizationNode_class_labels}
            metric: ${ThresholdOptimizationNode_metric}
            preserve_score: ${ThresholdOptimizationNode_preserve_score}
            recalibrate: ${ThresholdOptimizationNode_recalibrate}
    -   node: PerformanceSinkNode
        parameters:
            classes_names: &id001
            - Standard
            - Target
```

```
            ir_class: Target
parameter_settings:
    -   LibSVMClassifierNode_class_labels: *id001
        LibSVMClassifierNode_complexity: 0.00496861907944004
        LibSVMClassifierNode_epsilon: 0.10991484505052136
        LibSVMClassifierNode_kernel_type: RBF
        LibSVMClassifierNode_max_iterations: 531.0
        LibSVMClassifierNode_max_time: 3142.0
        LibSVMClassifierNode_nu: 0.437917298984716
        LibSVMClassifierNode_offset: -2.0
        LibSVMClassifierNode_ratio: 0.4030089945452672
        LibSVMClassifierNode_tolerance: 0.0010017072993929088
        LocalStraightLineFeatureNode_coefficients_used:
        - 1
        LocalStraightLineFeatureNode_segment_width: 400
        LocalStraightLineFeatureNode_stepsize: 120
        ThresholdOptimizationNode_class_labels: *id001
        ThresholdOptimizationNode_metric: Balanced_accuracy
        ThresholdOptimizationNode_preserve_score: true
        ThresholdOptimizationNode_recalibrate: true
        XDAWNNode_erp_class_label: Target
        XDAWNNode_retained_channels: 80.0
type: node_chain
```

D.2. Optimized pipeline of Experiment 4.4

```
input_path: eeg_data_reduced
node_chain:
    -   node: TimeSeriesSourceNode
    -   node: XDAWNNode
        parameters:
            erp_class_label: ${XDAWNNode_erp_class_label}
            retained_channels: ${XDAWNNode_retained_channels}
    -   node: LocalStraightLineFeatureNode
        parameters:
            coefficients_used: ${LocalStraightLineFeatureNode_coefficients_used}
            segment_width: ${LocalStraightLineFeatureNode_segment_width}
```

```
            stepsize: ${LocalStraightLineFeatureNode_stepsize}
    -   node: GaussianFeatureNormalizationNode
    -   node: LibSVMClassifierNode
        parameters:
            class_labels: ${LibSVMClassifierNode_class_labels}
            complexity: ${LibSVMClassifierNode_complexity}
            epsilon: ${LibSVMClassifierNode_epsilon}
            kernel_type: ${LibSVMClassifierNode_kernel_type}
            max_iterations: ${LibSVMClassifierNode_max_iterations}
            max_time: ${LibSVMClassifierNode_max_time}
            nu: ${LibSVMClassifierNode_nu}
            offset: ${LibSVMClassifierNode_offset}
            ratio: ${LibSVMClassifierNode_ratio}
            tolerance: ${LibSVMClassifierNode_tolerance}
    -   node: ThresholdOptimizationNode
        parameters:
            class_labels: ${ThresholdOptimizationNode_class_labels}
            metric: ${ThresholdOptimizationNode_metric}
            preserve_score: ${ThresholdOptimizationNode_preserve_score}
            recalibrate: ${ThresholdOptimizationNode_recalibrate}
    -   node: PerformanceSinkNode
        parameters:
            classes_names: &id001
            - Standard
            - Target
            ir_class: Target
parameter_settings:
    -   GaussianFeatureNormalizationNode_retrain: 0
        GaussianFeatureNormalizationNode_store: 0
        LibSVMClassifierNode_class_labels: *id001
        LibSVMClassifierNode_complexity: 0.020014715543786588
        LibSVMClassifierNode_epsilon: 0.0948931974742637
        LibSVMClassifierNode_kernel_type: RBF
        LibSVMClassifierNode_max_iterations: 453.0
        LibSVMClassifierNode_max_time: 3260.0
        LibSVMClassifierNode_nu: -1.8112260558696622
        LibSVMClassifierNode_offset: 6.0
```

```
        LibSVMClassifierNode_ratio: 0.3386794886455848
        LibSVMClassifierNode_tolerance: 0.001000331230905007
        LocalStraightLineFeatureNode_coefficients_used:
        - 0
        - 1
        LocalStraightLineFeatureNode_segment_width: 200.0
        LocalStraightLineFeatureNode_stepsize: 40.0
        ThresholdOptimizationNode_class_labels: *id001
        ThresholdOptimizationNode_metric: Balanced_accuracy
        ThresholdOptimizationNode_preserve_score: false
        ThresholdOptimizationNode_recalibrate: false
        XDAWNNode_erp_class_label: Target
        XDAWNNode_retained_channels: 78.0
type: node_chain
```

D.3. Optimized pipeline of Experiment 4.5

```
input_path: eeg_data_reduced
node_chain:
    -   node: TimeSeriesSourceNode
    -   node: XDAWNNode
        parameters:
            erp_class_label: ${XDAWNNode_erp_class_label}
            retained_channels: ${XDAWNNode_retained_channels}
    -   node: LocalStraightLineFeatureNode
        parameters:
            coefficients_used: ${LocalStraightLineFeatureNode_coefficients_used}
            segment_width: ${LocalStraightLineFeatureNode_segment_width}
            stepsize: ${LocalStraightLineFeatureNode_stepsize}
    -   node: GaussianFeatureNormalizationNode
    -   node: RMM2Node
        parameters:
            class_labels: ${RMM2Node_class_labels}
            complexity: ${RMM2Node_complexity}
            epsilon: ${RMM2Node_epsilon}
            kernel_type: ${RMM2Node_kernel_type}
            max_iterations: ${RMM2Node_max_iterations}
```

```
            max_time: ${RMM2Node_max_time}
            nu: ${RMM2Node_nu}
            offset: ${RMM2Node_offset}
            offset_factor: ${RMM2Node_offset_factor}
            range_: ${RMM2Node_range_}
            ratio: ${RMM2Node_ratio}
            regression: ${RMM2Node_regression}
            squared_loss: ${RMM2Node_squared_loss}
            tolerance: ${RMM2Node_tolerance}
    -   node: ThresholdOptimizationNode
        parameters:
            class_labels: ${ThresholdOptimizationNode_class_labels}
            metric: ${ThresholdOptimizationNode_metric}
            preserve_score: ${ThresholdOptimizationNode_preserve_score}
            recalibrate: ${ThresholdOptimizationNode_recalibrate}
    -   node: PerformanceSinkNode
        parameters:
            classes_names: &id001
            - Standard
            - Target
            ir_class: Target
parameter_settings:
    -   LocalStraightLineFeatureNode_coefficients_used:
        - 1
        LocalStraightLineFeatureNode_segment_width: 400
        LocalStraightLineFeatureNode_stepsize: 120
        RMM2Node_class_labels: *id001
        RMM2Node_complexity: 0.222420698071498
        RMM2Node_epsilon: 0.09424792879901636
        RMM2Node_kernel_type: RBF
        RMM2Node_max_iterations: 167.0
        RMM2Node_max_time: 205.0
        RMM2Node_nu: 0.8494804836089036
        RMM2Node_offset: 1.0
        RMM2Node_offset_factor: 10
        RMM2Node_range_: 65.41895485199798
        RMM2Node_ratio: 0.3865957105839527
```

```
        RMM2Node_regression: false
        RMM2Node_squared_loss: false
        RMM2Node_tolerance: 0.0010009045589999825
        ThresholdOptimizationNode_class_labels: *id001
        ThresholdOptimizationNode_metric: Balanced_accuracy
        ThresholdOptimizationNode_preserve_score: false
        ThresholdOptimizationNode_recalibrate: true
        XDAWNNode_erp_class_label: Target
        XDAWNNode_retained_channels: 110.0
type: node_chain
```

D.4. Optimized pipeline of Experiment 4.6

```
input_path: eeg_data_reduced
node_chain:
    -   node: TimeSeriesSourceNode
    -   node: XDAWNNode
        parameters:
            erp_class_label: ${__XDAWNNode_erp_class_label__}
            retained_channels: ${__XDAWNNode_retained_channels__}
    -   node: LocalStraightLineFeatureNode
        parameters:
            coefficients_used: ${__LocalStraightLineFeatureNode_coefficients_used__}
            segment_width: ${__LocalStraightLineFeatureNode_segment_width__}
            stepsize: ${__LocalStraightLineFeatureNode_stepsize__}
    -   node: GaussianFeatureNormalizationNode
    -   node: RMM2Node
        parameters:
            class_labels: ${__RMM2Node_class_labels__}
            complexity: ${__RMM2Node_complexity__}
            epsilon: ${__RMM2Node_epsilon__}
            kernel_type: ${__RMM2Node_kernel_type__}
            max_iterations: ${__RMM2Node_max_iterations__}
            max_time: ${__RMM2Node_max_time__}
            nu: ${__RMM2Node_nu__}
            offset: ${__RMM2Node_offset__}
            offset_factor: ${__RMM2Node_offset_factor__}
```

```
            range_: ${__RMM2Node_range___}
            ratio: ${__RMM2Node_ratio__}
            regression: ${__RMM2Node_regression__}
            squared_loss: ${__RMM2Node_squared_loss__}
            tolerance: ${__RMM2Node_tolerance__}
    -   node: ThresholdOptimizationNode
        parameters:
            class_labels: ${__ThresholdOptimizationNode_class_labels__}
            metric: ${__ThresholdOptimizationNode_metric__}
            preserve_score: ${__ThresholdOptimizationNode_preserve_score__}
            recalibrate: ${__ThresholdOptimizationNode_recalibrate__}
    -   node: PerformanceSinkNode
        parameters:
            classes_names: &id001
            - Standard
            - Target
            ir_class: Target
parameter_settings:
    -   __LocalStraightLineFeatureNode_coefficients_used__:
        - 0
        - 1
        __LocalStraightLineFeatureNode_segment_width__: 200.0
        __LocalStraightLineFeatureNode_stepsize__: 20.0
        __RMM2Node_class_labels__: *id001
        __RMM2Node_complexity__: 0.001129869866941247
        __RMM2Node_epsilon__: 0.09194077699198887
        __RMM2Node_kernel_type__: POLY
        __RMM2Node_max_iterations__: 528.0
        __RMM2Node_max_time__: 260.0
        __RMM2Node_nu__: 0.9292528088100147
        __RMM2Node_offset__: 2.0
        __RMM2Node_offset_factor__: 10
        __RMM2Node_range___: 27.250164324900883
        __RMM2Node_ratio__: 1.1972711802788927
        __RMM2Node_regression__: false
        __RMM2Node_squared_loss__: true
        __RMM2Node_tolerance__: 0.0009974896299573233
```

```
        __ThresholdOptimizationNode_class_labels__: *id001
        __ThresholdOptimizationNode_metric__: Balanced_accuracy
        __ThresholdOptimizationNode_preserve_score__: false
        __ThresholdOptimizationNode_recalibrate__: false
        __XDAWNNode_erp_class_label__: Target
        __XDAWNNode_retained_channels__: 128.0
type: node_chain
```

D.5. Optimized pipeline of Experiment 4.7

```
input_path: eeg_data_reduced
node_chain:
    -   node: TimeSeriesSourceNode
    -   node: XDAWNNode
        parameters:
            erp_class_label: ${XDAWNNode_erp_class_label}
            retained_channels: ${XDAWNNode_retained_channels}
    -   node: TimeDomainFeaturesNode
        parameters:
            absolute: ${TimeDomainFeaturesNode_absolute}
    -   node: OutlierFeatureNormalizationNode
        parameters:
            outlier_percentage: ${OutlierFeatureNormalizationNode_outlier_percentage}
    -   node: LibSVMClassifierNode
        parameters:
            class_labels: ${LibSVMClassifierNode_class_labels}
            complexity: ${LibSVMClassifierNode_complexity}
            epsilon: ${LibSVMClassifierNode_epsilon}
            kernel_type: ${LibSVMClassifierNode_kernel_type}
            max_iterations: ${LibSVMClassifierNode_max_iterations}
            max_time: ${LibSVMClassifierNode_max_time}
            nu: ${LibSVMClassifierNode_nu}
            offset: ${LibSVMClassifierNode_offset}
            ratio: ${LibSVMClassifierNode_ratio}
            tolerance: ${LibSVMClassifierNode_tolerance}
    -   node: ThresholdOptimizationNode
        parameters:
```

```
            class_labels: ${ThresholdOptimizationNode_class_labels}
            metric: ${ThresholdOptimizationNode_metric}
            preserve_score: ${ThresholdOptimizationNode_preserve_score}
            recalibrate: ${ThresholdOptimizationNode_recalibrate}
    -   node: PerformanceSinkNode
        parameters:
            classes_names: &id001
            - Standard
            - Target
            ir_class: Target
parameter_settings:
    -   LibSVMClassifierNode_class_labels: *id001
        LibSVMClassifierNode_complexity: 0.5527526527489306
        LibSVMClassifierNode_epsilon: 0.10777453458317456
        LibSVMClassifierNode_kernel_type: RBF
        LibSVMClassifierNode_max_iterations: 62.0
        LibSVMClassifierNode_max_time: 3334.0
        LibSVMClassifierNode_nu: 0.27545315291906236
        LibSVMClassifierNode_offset: -0.0
        LibSVMClassifierNode_ratio: 0.36556378418523167
        LibSVMClassifierNode_tolerance: 0.0010001154809138072
        OutlierFeatureNormalizationNode_outlier_percentage: 2.0
        ThresholdOptimizationNode_class_labels: *id001
        ThresholdOptimizationNode_metric: Balanced_accuracy
        ThresholdOptimizationNode_preserve_score: true
        ThresholdOptimizationNode_recalibrate: false
        TimeDomainFeaturesNode_absolute: false
        XDAWNNode_erp_class_label: Target
        XDAWNNode_retained_channels: 128.0
type: node_chain
```

Acronyms

BCI

Brain-computer-interface. 13, 37

CPU

central processing unit. 31

DFKI RIC

German Research Center for Artificial Intelligence (Robotics Innovation Center). 7, 47, 48

EEG

electroencephalogram. 2, 6, 9, 10, 12, 13, 16, 47, 49, 51, 65, 67, 70–72, *Glossary:* electroencephalogram

ERP

event related potential. 6, 13, 47, *Glossary:* event related potential

GPS

global positioning system. 71

GUI

graphical user interface. 10, 31, 41, 42

PDF

probability density function. 22–29, *Glossary:* probability density function

SMA

simple moving average. 42, *Glossary:* simple moving average

SMBO

sequential model-based optimization (Bayesian optimization). 6, 9–12, *Glossary:* sequential model-based optimization (Bayesian optimization)

SVM

support vector machine. 7, 21–26, 29, 49, 54, 59, 61, 73

TPE

tree-structured parzen estimator. 10, 11, *Glossary:* tree-structured parzen estimator

YAML

YAML Ain't Markup Language. 7, 31, 37, 38, 43, 80, *Glossary:* YAML Ain't Markup Language

Glossary

AutoWEKA

AutoWEKA is an automated optimization tool for the classification library *WEKA*. It is written in Java and implements the optimization of the structure and hyper-parameters of a processing pipeline [25][14]. For more details see Section 2.4. 6, 7, 9, 10, 12, 17

balanced accuracy

The balanced accuracy is a metric, which measures the performance of a classification. This metric can be used for binary classification and defines two classes. The class the user is interested in is called the "positive (P)" class. Thus, the number of positive examples, which the classifier classified as correctly as *positive* is called the "true positives (TP)". The same holds for the other class. It is called "negative (N)" and the number of correct classified examples of this class is called the "true negatives (TN)". This metric can be used for imbalanced datasets, because it excludes the effect of a bias the classifier can learn. This is achieved by taking both classes (TP and TN) into account. The performance is calculated by using the following formula:

$$\frac{1}{2}(\frac{TP}{P} + \frac{TN}{N})$$

[10]. 48, 51–54, 56–60, 62, 63, 65, 66

electroencephalogram

An electroencephalogram (EEG) is a record of electrical brain activity. It is recorded using an electrical monitoring device and recorded during a neurological test [2]. 2

event related potential

An event related potential is an epoch of the EEG that is time-locked to a stimulus. This stimulus might result in voltage changes in the EEG and because of this, is assumed to be a response of the brain to the stimulus measured by the EEG [24]. 6, *see also* EEG

Hyperopt

A Python library for optimizing search-spaces over real-valued, discrete, and conditional dimensions [6]. For more details see Section 2.5. III, 2, 10–12, 19, 39–41, 51, 79, 81, 82

Hyperopt-Sklearn

Hyperopt-Sklearn is an automated optimization tool for the classification library called scikit-learn [23]. It implements the automatic optimization of the structure and the hyperparameters of processing pipelines. The library Hyperopt [6] is used to search the domain for the optimal processing pipeline configuration [17]. For more details see Section 2.6. see. 6, 12, 17

hyperparameter

A Parameter of an algorithm, which is itself learning parameters for optimization. III, 2, 3, 6, 7, 9–12, 16, 17, 19–30, 32, 33, 35–38, 40–44, 46, 51–55, 57, 60, 61, 64, 68, 70, 73, 74, 77–79, 81, 82

NumPy

'NumPy is the fundamental package for scientific computing with Python' [12]. It is a library written in C and implementing many different algorithms and data types required for scientific computing like a vector data type, which is (unlike Pythons lists) fixed size. Additionally, it implements matrices, different distribution types, and many other utility functions required for numeric calculations. Because it is written in C and wrapped to Python, the library is compiled and can use processor extensions available to speed up the calculations [26]. 76, *see also* Python

P300

The P300 is an ERP. For more details see Section 2.7. 6, 13, 47, 49, 51, *see also* ERP

probability density function

'In probability theory, a probability density function (PDF), or density of a continuous random variable, is a function that describes the relative likelihood for this random variable to take on a given value.' [3]. 22

pySPACE

pySPACE stands for **S**ignal **P**rocessing **A**nd **C**lassification **E**nvironment in Python. As the name denotes, it is a machine learning framework written in Python and includes many different algorithms that can be used for classification and regression of arbitrary complex data with arbitrary complex processing pipelines [19]. For more information see Section 2.2. 2, 6–8, 10, 12, 17–20, 23, 31–33, 35–38, 40, 41, 43, 54, 60, 65, 68, 70, 71, 73, 74, 76, 79, 80, 82

Python

Python is a programming language that has many extensions and is feature rich. It is often used in scientific works because it features rapid prototyping and fast numeric analysis. It is established as the most often used programming language for machine learning [22],[20]. Because Python is an interpreted programming language, no compilation is required and thus rapid development is supported. Additionally, it has many powerful extensions available like scikit or NumPy for example. 7, 10, 12, 18, 19, 35, 76, 78

scikit-learn

scikit-learn is a machine learning framework for Python providing different classification, model selection, and preprocessing algorithms [23]. 7, 12, *see also* Python

sequential model-based optimization (Bayesian optimization)

The sequential model-based optimization (SMBO), or Bayesian optimization, is an technique that is used by many algorithms to optimize computational problems in which the evaluation is costly and therefore cannot be performed many times. For details see Section 2.3. 6

simple moving average

A simple moving average (SMA) is an average calculated on a series of data points by adding each data point to the average until a defined window size is reached

and then dividing with the number of data points in the window.
The SMA of width N for a data point $i >= N$ of the series T is defined by

$$SMA_i^T(N) = \frac{sum(T_{i-N}, T_{i-N+1}, \ldots, T_i)}{N}$$

[21]. Because a SMA of width N requires as least N data points to build an average, it is undefined for every data point $i < N$. 42

tree-structured parzen estimator

The tree-structured parzen estimator (TPE) is one implementation of the SMBO optimization technique. For a given hyperparameter space X, this approach models the probability of a hyperparameter setting x, given a loss y. This is done by splitting the estimation into two separate estimators. One estimator l is used to estimate hyperparameters for losses below the currently found best loss and another estimator g is used to estimate hyperparameters for losses above the best found loss. For each new sample, one of these estimators will receive new evidence, depending on the loss for that sample. Because l estimates the hyperparameter settings for losses better than all observed, no evidence to train this estimator would exist, as all samples would be used to train g. Because of this, TPE does not split the two estimators exactly at the best found loss, but uses a quantile γ above the best loss to split. Thus, l has confidence as well. During an optimization, TPE tries to minimize the loss by using the two estimators to sample new hyperparameters to test. After the result has been obtained, the estimators are updated and new hyperparameters will be sampled [8]. 10, *see also* SMBO

Bibliography

[1] Log-uniform distribution. http://ecolego.facilia.se/ecolego/show/Log-Uniform%20Distribution. Accessed: May 02, 2016.

[2] electroencephalogram. http://medical-dictionary.thefreedictionary.com/electroencephalogram. Accessed: November 2, 2015.

[3] Probability density function. https://en.wikipedia.org/wiki/Probability_density_function. Accessed: March 21, 2016.

[4] J. Aichison and J. A. C. Brown. *The Lognormal Distribution*. Cambridge University Press, 1957.

[5] O. Ben-Kiki, C. Evans, and I. döt Net. Yaml ain't markup language (yaml™) vesion 1.2. http://yaml.org/spec/1.2/spec.html, 2009. Accessed: July 21, 2016.

[6] J. Bergstra, D. Yamins, and D. D. Cox. Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms. In *Processding of the 12th Python in science conference*, 2013.

[7] J. S. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13:281–305, 2012. URL http://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf.

[8] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for hyper-parameter optimization. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 2546–2554. Curran Associates, Inc., 2011. URL http://papers.nips.cc/paper/4443-algorithms-for-hyper-parameter-optimization.pdf.

[9] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001. ISSN 1573-0565. doi: 10.1023/A:1010933404324. URL http://dx.doi.org/10.1023/A:1010933404324.

[10] K. H. Brodersen, C. S. Ong, K. E. Stephan, and J. M. Buhmann. The balanced accuracy and its posterior deistribution. In I. C. Society, I. A. for pattern recognition, and I. conference on pattern recognition, editors, *20th Internation Conference on Pattern Recognition*, pages 3121–3124. IEEE, Apr. 2010. ISBN 978-1-4244-7542-1. doi: 10.1109/ICPR.2010.764.

[11] T. S. community. scipy.stats.lognorm. `http://docs.scipy.org/doc/scipy-0.16.1/reference/generated/scipy.stats.lognorm.html`. Accessed: July 25, 2016.

[12] N. developers. Numpy. `http://www.numpy.org/`. Accessed: August 1, 2016.

[13] S. Fortmann-Roe. Understanding the bias-variance tradeoff. `http://scott.fortmann-roe.com/docs/BiasVariance.html`, 2016. Accessed: August 12, 2016.

[14] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: An update. *SIGKDD Explorations*, 11(1), 2009. URL `http://www.kdd.org/exploration_files/p2V11n1.pdf`.

[15] U. Hoffmann, J.-M. Vesin, T. Ebrahimi, and K. Diserens. An efficient p300-based brain–computer interface for disabled subjects. *Journal of Neuroscience Methods*, 167(1):115 – 125, 2008. ISSN 0165-0270. doi: 10.1016/j.jneumeth.2007.03.005. URL `http://dx.doi.org/10.1016/j.jneumeth.2007.03.005`.

[16] E. A. Kirchner, S. K. Kim, S. Straube, A. Seeland, H. Wöhrle, M. M. Krell, M. Tabie, and M. Fahle. On the applicability of brain reading for predictive human-machine interfaces in robotics. *PLoS ONE*, 8(12):1–19, 12 2013. doi: 10.1371/journal.pone.0081732. URL `http://dx.doi.org/10.1371%2Fjournal.pone.0081732`.

[17] B. Komer, J. Bergstra, and C. Eliasmith. Hyperopt-sklearn: Automatic hyperparameter configuration of scikit-learn. *ICML 2014 AutoML Workshop*, 2014.

[18] M. Krell, D. Feess, and S. Straube. Balanced relative margin machine — the missing piece between fda and svm classification. *Pattern Recognition Letters*, 41: 43 – 52, 2014. ISSN 0167-8655. doi: 10.1016/j.patrec.2013.09.018. URL `http://www.sciencedirect.com/science/article/pii/S0167865513003541`.

[19] M. M. Krell, S. Straube, A. Seeland, H. Wöhrle, J. Teiwes, J. H. Metzen, E. A. Kirchner, and F. Kirchner. pyspace - a signal processing and classification environment in python. *frontiers in Neuroinformatics*, 7, December 2013.

[20] K. J. Millman and M. Aivazis. Python for scientists and engineers. *Computing in Science Engineering*, 13(2):9–12, March 2011. ISSN 1521-9615. doi: 10.1109/MCSE.2011.36.

[21] P. Nicolas. *Scala for Machine Learning*. Community Experience Distilled. Packt Publishing, 2015. ISBN 9781783558759. URL `https://books.google.de/books?id=d5EIBgAAQBAJ`.

[22] T. E. Oliphant. Python for scientific computing. *Computing in Science Engineering*, 9(3):10–20, May 2007. ISSN 1521-9615. doi: 10.1109/MCSE.2007.58.

[23] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[24] M. D. Rugg and M. G. H. Coles. *Electrophysiology of Mind: Event-related Brain Potentials and Cognition*. Oxfort psychology series. Oxford University Press, 1996. ISBN 9780198524168.

[25] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Auto-weka: Combined selection and hyperparameter optimization of classification algorithms. Department of Computer Science, University of British Columbia, 201-2366 Main Mall, Vancouver BC, V6T 1Z4, Canada.

[26] S. van der Walt, S. C. Colbert, and G. Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science Engineering*, 13(2):22–30, March 2011. ISSN 1521-9615. doi: 10.1109/MCSE.2011.37.

List of Figures

2.1. Different algorithm types in pySPACE 8

3.1. Structure of pySPACEOptimizer 19
3.2. A decorated algorithm example 20
3.3. Plot of the PDF of a uniform distribution with min=2 and max=4 . . . 22
3.4. Plot of a quantified uniform distribution with min=2, max=4, and q=0.5 23
3.5. Plot of a log-uniform distribution with min=2 and max=200 24
3.6. Plot of a quantified log-uniform distribution with min=2, max=4, and q=0.5 25
3.7. Plot of a normal distribution with $\mu = 0$ and $\sigma = 1$ 27
3.8. Plot of a quantified normal distribution with $\mu = 0$, $\sigma = 1$, and $q = 0.5$. 28
3.9. Plot of a log-normal distribution with shape $= 1$ and scale=1 29
3.10. Plot of a quantified log-normal distribution with scale $= 1$, shape $= 1$, and $q = 0.5$ 30
3.11. Example plot of a performance graphic 34
3.12. Example task description 38
3.13. An overview of the GUI of the performance analysis tool 42
3.14. Example for *domain expert* knowledge in an optimization task 44

4.1. Setup of the P300 processing pipeline 49
4.2. Evaluation performance of Experiment 4.2 50
4.3. Optimization performance of Experiment 4.3 52
4.4. Evaluation performance of Experiment 4.3 53
4.5. Optimization performance of Experiment 4.4 55
4.6. Evaluation performance of Experiment 4.4 56
4.7. Optimization performance of Experiment 4.5 58
4.8. Evaluation performance of Experiment 4.5 59
4.9. Optimization performance of Experiment 4.6 62
4.10. Evaluation performance of Experiment 4.6 63

4.11. Optimization performance of Experiment 4.7 66
4.12. Evaluation performance of Experiment 4.7 67