FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR’S THESIS

Title:	Meta-learning for automatic model selection and hyperparameter optimization
Student:	Ivan Rychtera
Supervisor:	Ing. Markéta Jůzlová
Study Programme:	Informatics
Study Branch:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	Until the end of summer semester 2020/21

Instructions

Selecting a machine learning model and tuning its hyperparameters is a crucial task for applying machine learning. Recently, hyperparameter optimization methods, such as Bayesian optimization or Hyperband algorithm, were established for automatic optimization. However, these methods use only knowledge from the task in hand. In contrast, an experienced expert uses knowledge from already solved tasks. Meta-learning is a technique that aims to use information from different tasks to warm-start the optimization.

1. Review state of the art hyperparameter optimization and model selection methods.
2. Review and theoretically describe the state of the art approaches for initialization hyperparameter optimization methods with knowledge from previous tasks.
3. Use or implement meta-learning methods with at least three hyperparameter optimization methods.
4. Compare the performance of hyperparameter optimization with random initialization and initialization with meta-learning.

References

Will be provided by the supervisor.

Ing. Karel Klouda, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague December 2, 2019

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Meta-learning for automatic model selection and hyperparameter optimisation

Ivan Rychtera

Department of applied mathematics
Supervisor: Markéta Jůzlová

June 2, 2020

Acknowledgements

I would like to thank all the teachers at FIT CTU who helped me attain the knowledge I needed for writing this thesis. Special thanks goes to my supervisor Markéta Jůzlová for her help and patience in helping me with the thesis.

Declaration

In Prague on June 2, 2020 ....

Citation of this thesis

Rychtera, Ivan. *Meta-learning for automatic model selection and hyperparameter optimisation*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstract

Model selection and hyperparameter optimisation is an important step in creating well-performing machine learning models. Optimizing hyperparameters for a given task can be enhanced with information from previous tasks – meta-learning. This thesis looks into combining meta-learning with algorithms for hyperparameter optimisation and model selection. We look at several state-of-the-art hyperparameter optimisation and meta-learning methods. We assess the change in performance for 3 of the reviewed hyperparameter optimisation algorithms when incorporating meta-learning. The results does not prove any significant improvement of meta-learning compared to vanilla model selection methods. The subsequent analysis implies that this is due to the capability of hyperparameter optimisation and model selection methods to find well-performing configurations on their own in the given time frame.

Keywords machine learning, hyperparameter optimization, meta-learning, model selection

Abstrakt

Optimalizace hyperparametrů je důležitý krok pro vytváření přesných modelů strojového učení. Optimalizace hyperparametrů pro danou úlohu může být rozšířena informacemi z předchozích úloh – s meta-učením. Tato práce zkoumá kombinaci meta-učení s algorithmy pro optimalizaci hyperparametrů a výběr modelu. Posuzujeme přesnost predikce pro 3 algoritmy při zahrnutí meta-učení. Výsledky nedokazují žádné výrazné zlepšení optimalizačních algoritmů při zahrnutí meta-učení. Následná analýza naznačuje, že toto je výsledkem schopnosti metod pro optimalizaci hyperparametrů a výběr modelu najít v daném čase dobře fungující konfigurace bez pomoci.

Klíčová slova strojové učení, optimalizace hyperparamterů, meta-learning, výběr predikčního modelu

Contents

List of Figures

List of Tables

Introduction

In the past few years, machine learning has come back to the forefront after it was underperforming due to the technological limitations of our computers in the previous century. But thanks to increasing amount of the available computational power and a large amount of data, we are able to put machine learning into practical use. However, this has presented several challenges connected with this domain. While we understand many aspects of machine learning, some challenges are still waiting to be solved.

In machine learning we are training a model. The model can be thought of as a formula that is used to calculate a variable, similar to how we can get average speed by dividing distance by time. However, machine learning models are applied on less explored relations for which the exact formula is very complicated or unknown. These can be for instance calculating the price of a house based on its features, or deciding if a person has a given illness using medicinal data.

The way machine learning deals with this is by training a model using a set of observations of the real world. The training is done via adjusting the values of the model's parameters in order to approximate relationships in the measured data with as little error as possible. However, one of the problems is that the adjusting can take a lot of time, in some cases even hours.

Model perfomance is also affected by the so-called hyperparameters, which describe the model's structure or the training process. They need to be set before training. Finding the values of hyperparameters is a non-trivial task and we are going to describe some of the methods which address this task.

One type of method which aims to decrease the total number of model training iterations, called hyperparmater optimisation, is to try different hyperparameter values in every problem more efficiently than by random guessing. There are numerous approaches based on statistics or various domains of machine learning, but generally, they have one thing in common: they only utilise information from one particular problem at hand.

The other approach is to look at past solved problems for which we know the performance of a hyperparameter configuration and try to guess well-performing hyperparameters from them. This is referred to as meta-learning. This includes a variety of algorithms, but in general, once these methods provide the information for the learning task, they do not influence the learning process itself. Hyperparameter optimisation and meta-learning methods are not exclusive and can be combined. In this work, we evaluate how both of these methods, meta-learning and hyperparameter optimisation, perform and how they can be combined.

In the theoretical part, we review hyperparameter optimisation and meta-learning. In the experimental part, we survey the effects of the combination.

CHAPTER 1

Thesis goals

The thesis is split into a theoretical and an experimental part. The goal of the theoretical part is to survey meta-learning techniques and state-of-the-art methods of automatic model selection and hyperparameter optimisation, and look at the ways in which the two can be combined.

The main goal of the experimental part is to incorporate selected meta-learning methods into selected methods for automated model selection and hyperparameter optimisation. The first step is to implement meta-learning and then combine meta-learning with model selection and hyperparameter optimisation methods. After that, we compare different methods for hyperparameter optimisation with and without meta-learning.

Chapter 2

Thearetical part

2.1 Machine learning problem

A supervised machine learning problem is one where we need to estimate a very complex function, which is beyond either our understanding of the problem or our computational power, with a simpler and more manageable function. This function is represented by a model. There are various types of models, but they generally share some characteristics.

At first, we select a general model with free parameters. This model's parameters are then modified in order to approximate the specific function at hand. This is done via the training process. The model uses measured values of the real function to guess the function shape. This means that we have a set of examples, each of which is represented by a vector from which one element is the one that we wish to predict. This element is called the target variable and the rest are considered to be the function arguments. Depending on the selected model, the training process can greatly vary in time and computing power requirements, from a simple matrix multiplication for a linear regression to a multi-level gradient descent when using a convolutional neural network.

Despite the differences in the estimating process itself, the training pipeline is done in the same pattern across all the models. First, the measured examples referred to as datapoints are divided into two groups, a training dataset, and a test dataset. After that, a configuration describing the model's structure and a training algorithm is chosen and the training set is further divided into the actual training dataset and the validation dataset. The model is then trained on the training set, which means that it is led to try to simulate the function that would give the right target values for every example in the training set. By the very nature of this process, it is not guaranteed that if the model's approximation works well on the training set, it will also give good results in a real situation. Because the training set is not a complete representation of the input space, the model can get too fixed on the examples in the training set and then perform poorly when faced with a new case. Should this be the instance, it is called *overfitting*. That is when the validation dataset comes in. The model's performance is evaluated on examples it has not yet seen to get an idea about how well it is going to perform in a real situation (i.e. how it generalizes to unseen data). If a model has a good performance on the training dataset and a poor performance on the validation set, it probably means it is overfitted.

After evaluating the model on the validation set, another configuration is chosen and the process is repeated until a certain condition is met (we do not have any more time or we have a good enough performance etc.). Then the model with the best performance on the validation set is chosen to be evaluated in the test set to get the final performance we have achieved. [1]

Solving all the problems connected to the training process can be very time-consuming if done manually by a human practitioner. Therefore, several ways have been proposed to automate the process, which will be discussed further in the paper. They are known as hyperparameter optimisation algorithms and can usually find well-performing configurations faster than a human. However, despite having some general rules about the model configurations or being able to tell which configurations are worth exploring when we have already computed some of the possible ones, as we will see, there is no general way of telling which configurations are promising right at the beginning of the training process.

In this thesis we attempt to solve this issue with the use of meta-learning, sometimes referred to as transfer learning. Meta-learning is a set of methods which work on the principle of looking into the past and finding similar problems which have already been solved and using their solutions to help solve the problem at hand. With this approach, we should be able to warm start the hyperparameter optimisation algorithms so that they can start exploring the hyperparameter space with some suggested configurations which should work well. Thus, the assumption is that it may reduce the time they need as well as find better performing results.

2.1.1 Related terms

In this part, we are going to look at the problems we will be facing and explain the fundamental terms.

First, we need to differentiate between two types of values, parameters and hyperparameters. Parameters are variables in the model whose values we obtain via the learning process itself. They are adjusted during learning in order to minimise the loss function. Examples of parameters are the weights in a neural network or a linear regression model or the margins in a support vector machine.

Hyperparameters, on the other hand, are variables describing the overall architecture of the model or the learning process itself. Examples of hyperparameters are the number of hidden layers in a neural network, the number of trees in a random forest or even the initialization process of the parameters. But there are also other, much less apparent hyperparameters, including ways of dealing with missing values in the data, pre-processing methods or various sampling methods, which all play a part in how well the model will perform. Hyperparameter values are usually specified manually by a practitioner for each run of the training algorithm in order to find the configuration which yields the best performance.

However, the problem of selecting hyperparameters assumes that we already know which model is going to be the best in terms of performance on a given dataset. Since all the parameters and hyperparameters are dependent on the model, we first need to select the model before any hyperparameter configurations can be tested. This process is called model selection and can be tackled in a number of ways. With manual probing, the practitioner would simply test one model and in case it would perform poorly, another model is tested. Should we look at this more formally, the model can be thought of as a special case of a hyperparameter, which needs to be chosen first. After that, other hyperparameters can be assigned. Another view is that finding the appropriate model is a task that needs to be solved prior to finding the hyperparameters' values. For simplicity, we are going to use the first approach and include the model types in hyperparameter configurations.

Thus, we find ourselves standing before a problem where we need to choose hyperparameter configuration M, so that

$$M = \mathsf{arg\,min}_x \ f(x, D) \tag{2.1}$$

where D is the validation dataset at hand, f is the loss function (i.e. prediction error of model) and x is an element of the hyperparameter configuration space. While parameter values are acquired through the learning process, hyperparameters are set at the beginning of the learning process and cannot be changed during it. Therefore, naturally, we want to find the best hyperparameters in the least amount of attempts possible.

The problem with such space is that it is far from ideal for any optimisation. Not only does this space consist of discrete as well as continuous dimensions, but also, even the presence of some dimensions is dependent on the value of others. For instance, the existence of a hyperparemeter "number of neurons in a n-th hidden layer" of a neural network depends on whether the number of hidden layers is at least n. [1]

2.2 Hyperparameter optimization methods

In this section, we discuss some methods used for searching for a hyperparameter configuration.

2.2.1 Grid search

The first method which can be fully automatized is Grid search. As the name suggests, this approach works by drawing an even grid across the hyperparameter space and then evaluating every point on that grid. The grid is the cartesian product of selected finite subsets of values of each hyperparameter. The configuration with the best validation performance is then selected. This sounds good at first glance, since it tries configurations evenly across the whole space. This method is suitable for low dimensional spaces, as they are not too big and Grid search offers an exhaustive way to explore them. However, when used in a high dimensional space, it tends to perform poorly, due to the fact that it spends too much time exploring non-perspective areas of the hyperparameter space instead of trying to focus on the more promising areas. Another problem with high dimension is that the number of combinations of hyperparameter values increases exponentially with each dimension, making Grid search take a large amount of time.

2.2.2 Random search

The simpler, yet often more effective method is Random search. Again, as the name suggests, this method consists of trying random configurations and then picking the configuration with the best validation performance. Random configurations are sampled by approaching hyperparameters as distributions, mainly even or uniform. Considering that this method is straightforward, the results are comparable to more sophisticated methods. [2]

Unlike Grid search Random search does not need to test all possible combinations. Therfore, it is able to avoid some problems with high dimensionality. It also does not need to go through some explicitly given number of configurations, so it may run as long or as short as needed if we have limited training resources.

2.2.3 Sequential Model-based Optimisation

The first more complex method that we will describe is Sequential Model-based Optimisation, or SMBO, as it is commonly referred to. SMBO aims to improve on Random search, capitalising on its advantages while trying to add something more complex than just random actions. The main idea behind SMBO is that around well-performing hyperparameter configurations in the hyperparameter space, there should be more well-performing configurations, while near poorly performing configurations there are more likely to be other poorly performing ones. Balance between exporation and exploitation is needed. Therfore, it alternates between evaluating configurations and gathering additional information from evaluated configurations.

The way this is implemented is that a response surface model is cast over the whole hyperparameter space. In the first step of the optimization process, a number of configuration points are chosen and their validation loss is acquired. These configuration points serve as the first key datapoints which are used to train the response surface model. The best-performing configuration is selected as the incumbent. The response surface model can now predict validation loss for the whole input space and uncertainty of the prediction, the latter decreasing the closer we get to known points in the space.

After predicting the validation loss for multiple points, the optimisation selects the candidate with best utility for the next iteration of the configuration training. The utility is accessed by calculating the so-called acquisition (utility) function. The most common acquisition function is the *Expected Improvement* across the space. *Expected Improvement* combines predicted mean and uncertainty of a configuration loss estimation by response surface. The point with the highest *Expected Improvement* is the point with the best balance of a low mean, so that we exploit areas with well-performing configurations of the space, and high uncertainty, so that we explore unvisited areas. The candidate then competes against the incumbent to assess whether the candidate is performing better. This is done by training the configurations with a certain amount of seeds to avoid too much variance. If the candidate proves to be better, it becomes the new incumbent. The new configuration and its computed loss is then fed back to update the surface model and another candidate configuration is selected for evaluation. This process can be repeated for as long as needed.

2.2.3.0.1 Response surface According to Hutter et al. [3] the most prominent model architectures for the response surface are GAUSSIAN STOCHASTIC PROCESS (GP) models and RANDOM FORESTS. GP is a probabilistic linear basis function model, using a multivariate normal distribution. RANDOM FORESTS is a collection of regression decision trees which have real values in the leaves.

2.2.3.0.2 Challenger selection As stated, acquisition function should return high values for points with low mean and/or high variance in the response surface. *Expected Improvement* gives good results, as shown by Hutter. There is still a problem of how to look for the points where high EI is expected since random sampling is often insufficient in high-dimensional spaces. Therefore, in addition to that, we can compute EI on already evaluated configurations and then start evaluating their neighbours.

Since the previously defined approach relies on training each configuration multiple times to determine if it outperforms the incumbent, it can be quite time consuming to get a reasonable idea about the whole hyperparameter space. [3]

2.2.3.1 Random online aggressive racing

Random online aggressive racing fits the framework of SMBO. We consider the model to be a constant which results in the configuration selector returning randomly sampled configurations every time. Its core lies in the way it is evaluating models and considering how many evaluations must be done in order to say whether the configuration we are evaluating is better than the best-known configuration so far (the incumbent).

ROAR is a procedure that is given an incumbent configuration evaluated on s randomly selected seeds with the seeds and a challenger. It starts by evaluating the challenger on a randomly selected seed from the pool on which the incumbent was already evaluated. If the challenger performs worse than the incumbent, it is aggressively rejected before having enough empirical evidence to support it. If the challenger is at least as good as the incumbent, the number of seeds on which the challenger is evaluated is doubled. The process is repeated either until the challenger performs worse on the current subset of seeds or the challenger is evaluated on at least s seeds and still performs better. In that case, challenger becomes the new incumbent.

2.2.3.2 Sequential Model-based Algorithm configuration

SMAC is a more sophisticated instance of the SMBO framework. It extends the ROAR procedure by adding a more complex response surface and an *expected improvement* function described in the SMBO section, instead of random uniform sampling.

2.2.4 Genetic programming

Another more sophisticated procedure is selecting the hyperparameter values via tree-based pipeline optimisation. This approach works with the hyperparameter configurations as if they were trees which have different hyperparameters in their nodes. Nodes can be split into three categories.

The first category is pre-processing methods, which transform the data before they are used for training. Common pre-processing methods produce new features or conversely reduce dimensionality with methods such as Principal component analysis. The second category includes only one node, which decides what model should be used. The last subset of nodes contains hyperparameters describing the model (depth of a tree, learning rate). When we adopt this tree-based look at the hyperparameters, the optimisiation can then be done via Genetic programming.

Genetic programming is a method of optimising programs, which in our case are represented by a tree. The name of this method comes from natural selection. The similarities between the two is that both are aiming to find the best possible genome (in Genetic programming it is the tree structure) operating on a large number of individuals, which is called a population.

To be more precise, Genetic programming in this case first randomly generates a population of pipelines and evaluates their performance. The unfit, poor performing pipelines are then eliminated from the population. To make up for the lost individuals, for each pipeline eliminated a new one is created via the *crossover* operation, which takes two well-performing pipelines and combines their features into a new individual, in this case by randomly substituting subtrees from one another. Additionally, to introduce more exploration into the system to prevent premature convergence into a local minimum, *mutation* operation is then applied. Mutation takes a small number of individuals and makes a random change in their structure. The population is then evaluated again and the whole process is repeated, until either the population is unable to improve or the algorithm is stopped. [4]

2.2.5 Hyperband

Another approach to hyperparameter optimization and model selection is the Hyperband algorithm, which is short for Hyperbandit. It aims to balance exploration and exploitation by dynamic allocation of training resources.

2.2.5.1 Successive halving

The core part of the Hyperband algorithm is Successive halving. This method considers exponentially more hyperparameter configurations than sequential training would, while increasing training time only marginally. It takes a given number of configurations and uniformly allocates part of a given budget to all configurations. The resource can be for example computation time or amount of datapoints used for training. It then estimates final loss of all the configurations with the allocated budget, discards a subset of configurations that performed poorly (the name suggests the bottom half, but the exact amount can be different), then allocates more resources to remaining configurations and repeats the process until only the best performing model with the best configuration remains.

This algorithm, however, presents a question of what the initial number of configurations should be. A large number of them will mean that there will be less resources for each configuration, thus vastly reducing the accuracy of the estimate of the real performance. This could lead to potentially better performing configurations being incorrectly eliminated early in the routine. Conversely, a small number leads to fewer configurations being probed. [2]

2.2.5.2 Hyperband

Hyperband addresses the problem with the optimal number of initial configurations in Successive halving by running Successive halving multiple times, with a different number of initial configurations in each iteration. These outer iterations are referred to as “brackets”. In each bracket, the minimum amount of resources allocated for each configuration is specified. Without the loss of generality, let us assume we start with the smallest number of resources allocated for each configuration and increase the number in every bracket. In the earlier brackets, there will be more configurations, with a lot of them stopping early, while in later stages, the pool of configurations will be smaller but the configurations will run with more resources. Thus, while in the beginning, the population drops rapidly, in the last bracket, every configuration is given the maximum amount of resources possible and is therefore reduced to a simple random search.

Hyperband also takes another argument which specifies what proportions of configurations should be eliminated in each iteration of Successive halving. This can be used to further adjust the exploration and exploitation trade-off. [2]

Algorithm 1 Hyperband

```
    Input $\eta$ - halving factor, $B$ - data budget
    Output
 1: Initialize $s_{max} := \lfloor \log_\eta(B) \rfloor$
 2: for $s \in \{s_{max}, s_{max} - 1, ..., 0\}$ do
 3:     $n = \lceil \frac{\eta^s}{s+1} \rceil$, $r = \eta^{-s}$
 4:     $T = n$ random configurations
 5:     for $i \in \{0, ..., s\}$ do                ▷ Run Successive halving
 6:         $n_i = \lfloor n\eta^{-i} \rfloor$
 7:         $r_i = rn_i^i$
 8:         Evaluate all configurations in $T$ on budget $r_i$
 9:         $T :=$ Top $\lfloor \frac{n_i}{\eta} \rfloor$ performing configurations from $T$
10:     end for
11: end for
```

2.2.6 Other methods

This section focuses on some other interesting hyperparmeter optimisation algorithms. They borrow from two different fields of computational intelligence.

2.2.6.1 MLplan

MLplan is a method that utilises algorithms used in automatic planning. It was proposed by Mohr et al. [5].

2.2.6.1.1 Planning problem A planning task is defined in a logic planning language L, which has to have first-order logic capabilities. The task is then a tuple (P, A, I, G) where P is a finite set of predicates, A is a finite set of operators, I is the formula describing the initial state and G is the formula describing the goal. An operator is a tuple $(name, pre, post)$ where *name* is name, *pre* is a formula in L describing the preconditions for applying the operator and *post* is a formula describing the resulting changes. An action is an operator, where variables in *pre* and *post* are replaced by constants. A state is a set of positive literals, literals not explicitly expressed in the state are assumed false. A state can also be defined to explicitly express both true and false literals. A plan is a sequence of actions that transform state I into state G.

Action a is applicable in state s if $s \models pre_a$. The successor state s' resulted application of a is s if a is not applicable in s and $(s \cup add) \setminus del$ otherwise, where *add* and *del* are positive and negative literals in $post_a$ respectively.

For the purposes of machine learning pipeline optimization, the operator is *OneHotEncode* or *SetDecisionTreeMaxDepth*. Terms can then describe state of the algorithm or dataset property, for example, when data contains categorical features or when a decision tree has a missing value for max depth.

MLplan solves the planning task using hierarchical task network (HTN). This network is a partially ordered set of tasks. A task is similar to an operator in a planning task, however, it does not have to be defined in A. A primitive task can be realised by a single operator, otherwise it is a complex task. HTN has a tree structure, where in the root there is the complex task representing the initial machine learning problem, *classification* or *regression*. The root task can then be decomposed into machine learning pipeline steps – tasks such as *preprocess* or *configureRandomForest*. In the leaf nodes of HTN, there is a plan consisting of only primitive tasks (OneHotEncode, etc.). It is similar to grammar derivation, where complex tasks are nonterminal symbols and primitive tasks are terminal symbols. MLplan can explore this tree using various methods and heuristics which we do not describe in mre detail, they are discussed in the citation.

2.2.6.2 AlphaD3M

AlphaD3M is a game theory-based approach inspired by Alpha Zero. It approaches hyperparameter optimisation as a single player configuration synthesis game. The intuition on mapping between primitives in the game and in model selection and HPO terms is in Table 2.1.

Table 2.1: Comparison between AlphaZero and AlphaD3M [6]

	AlphaZero	AlphaD3M
Game	Chess	Hyperparameter optimisation
Unit	Piece	Preprocessor, hyperparameter value
Sate	Board configuration	Pipeline, task, metadata
Acion	Move	Insert, Delete, Replace
Loss	Win/lose/draw	Performance

AlphaD3M has two parts: a Monte Carlo tree search and an LSTM neural network. Monte Carlo tree search randomly synthesises configurations and evaluates them, thus providing examples for the neural network to learn from. However, the variance in the randomness is mitigated, since the tree search also takes into account predictions made by the neural network, which predicts probabilities of actions the tree search should take.

The neural network aims to predict the performance of configurations as well as the probabilities of taking different actions on them. It is learning on evaluated examples provided by the tree search. The network parameters are optimised by minimising cross-entropy between real and predicted configuration probabilities and the means squared error between real and predicted performance. There are also two regularization terms, l_2 for network parameters to avoid overfitting and l_1 to avoid complex configurations. [6]

2.3 Meta-learning

While machine learning is a problem of manipulating and understanding data itself, meta-learning is, in our case, understanding the process of machine learning, so that we are able to do it more effectively, rather than brute-forcing the problems which are associated with it. With meta-learning, this is mainly done by looking into tasks that we have dealt with in the past and extracting information about how machine learning is progressing. We then use this metadata to extract meta-knowledge which helps us to determine a strategy for future tasks, hopefully helping us reach our goals faster and with a better result. Several definitions of meta-learning have been proposed, these are some picked from various sources by Lemke et al. [7] “

1. Meta-learning studies how learning systems can increase in efficiency through experience; the goal is to understand how learning itself can become flexible according to the domain or task under study.

2. The primary goal of meta-learning is the understanding of the interaction between the mechanism of learning and the concrete contexts in which that mechanism is applicable.

3. Meta-learning is the study of principled methods that exploit meta-knowledge to obtain efficient models and solutions by adapting machine learning and data mining processes.

4. Meta-learning monitors the automatic learning process itself, in the context of the learning problems it encounters and tries to adapt its behaviour to perform better.

”

For our purposes, we adopt definition number 3.

2.3.1 Meta-features and models

To use meta-learning, we have to define the means by which we can describe the characteristics of a machine learning problem. We can call these means *meta-features.* They can be categorised into the following groups [8] :

- *Simple meta-features*, which describe basic characteristics of the dataset for example number of records, number of features or number of classes.
- *Statistical meta-features*, which describe the data distribution by means of statistical measurements such as skewness or kurtosis.
- *PCA meta-features*, which are computed when PRINCIPAL COMPONENT ANALYSIS is applied to the dataset. Variables similar to *statistical meta-features* are then computed on the principal components.
- *Inforamtion-theroetic meta-features*, which are entropy of dataset classes or features.
- *Landmarking meta-features*, which are acquired by training a simple quick-learning machine learning model on the dataset. The model performance is then used as a *meta-feature.* This type of *meta-feature* appears to be highly relevant to the hyperparameter optimization problem, because it provides insight on how given simple approaches perform by themselves and in comparison to others.

With these variables, we are now able to describe a dataset. However, we also need to determine similarity between two datasets. Similarity can be expressed in two ways, we can ask "how similar" two datasets are, or whether two datasets are "similar enough". The former view leads to a regression task, the latter to a classification task. Clasification is the more popular approach and offers more possibilities [7]. Therefore, we will limit ourselves to viewing meta-learning as a classification problem.

Meta-learning can be implemented by common classification algorithms such as decision trees or support vector machines. However, in this thesis we will focus mainly on two methods – k-NEAREST NEIGHBOURS (kNN) and clustering.

2.3.1.1 Metric

Both kNN and clustering need to determine distances between elements. Distance functions are also called metrics. A metric is every function on a metric space V defined as

$$d := V \times V \to [0, \infty) \tag{2.2}$$

where $\forall x, y, z \in V$:

$$d(x,y) = 0 \iff x = y \tag{2.3}$$

$$d(x,y) = d(y,x) \tag{2.4}$$

$$d(x,y) \leq d(x,z) + d(z,y) \tag{2.5}$$

or positive-definiteness (2.2 and 2.3), symmetry 2.4 and triangle inequality 2.5 respectively.

The most commonly used metrics are L^p metrics, which are defined as:

$$d(x,y) = \sum_{i=1}^{n} \sqrt[p]{|x_i^p + y_i^p|} \tag{2.6}$$

for $x, y \in V$ where n is the dimension of space V. With $p = 2$, we get the most used Euclidean distance.

In meta-learning tasks, V is a space of meta-features and distance represents how similar two datasets are – the closer, the more similar.

2.3.2 *k*-nearest neighbours

This method is simple. We have a dataset which in our case consists of previously solved machine learning problems described by meta-features with their well-performing configurations. When we face a new machine learning problem, we find k problems from the dataset which are the closest to the new one and call them neighbours. We can then use the neighbours' configurations to help us solve the new machine learning problem. [9]

2.3.3 Clustering

Clustering is a method of unsupervised machine learning, which means it does not aim to predict a target variable, but rather to provide some structural information about the data. In our case, we aim to find groups of machine learning problems. The groups are determined by affiliation of the past problems to clusters. Clusters are subsets of a dataset, which should ideally meet two conditions: elements that are close to each other should be in the same cluster, while elements far from each other should be in different clusters. This definition is vague and leads to problems, which require some trade-offs to be made.

Once the clusters are determined, we can then assign any new machine learning problem to an existing cluster, thus declaring it similar enough to the subset of past problems in that cluster. We can then utilise the past problems' well-performing configurations to help us with the new problem. [10]

2.3.3.1 k-means

k-MEANS is one of the algorithms which are used in meta-learning. It defines clustering as an optimisation problem of finding a decomposition of the dataset $C = (C_1, ...C_k)$ to minimise loss function

$$L(C) = \sum_{i=1}^{k} \frac{1}{2|C|} \sum_{x,y \in C_i} d(x,y)^2. \tag{2.7}$$

We can rewrite the function as follows

$$L(C) = \sum_{i=1}^{k} \sum_{x \in C_i} d(x, \bar{x}_i)^2, \tag{2.8}$$

where $\bar{x}_i$ is the center of i-th cluster. Detailed derivation can be found in [10].

The first step of the algorithm is to randomly intialise the centroids of each cluster $(\mu_1, ...\mu_i)$. We now find a decomposition of $C = (C_1, ..., C_k)$ by assigning every x to a cluster C_i with the lowest value of $d(x, \mu_i)$.

The rest of the steps of the algorithm consist of iterating the following: Redefine the centroids $\tilde{\mu_i} := \bar{x}_i$ and find a new decomposition $\tilde{C} = (\tilde{C}_1, ..., \tilde{C}_k)$ by again finding the lowest value of $d(x, \tilde{\mu_i})$ This implies

$$\sum_{i=1}^{k} \sum_{x \in \tilde{C}_i} d(x, \tilde{\mu_i})^2 \leq \sum_{i=1}^{k} \sum_{x \in C_i} d(x, \mu_i)^2 \tag{2.9}$$

and therefore $L(\tilde{C}) \leq L(C)$. Iteration stops when $L(\tilde{C}) = L(C)$ and algorithm converges to a local optimum.

2.3.4 Ranking configurations

Another interesting problem can be determining which configurations are "the best" on a given dataset. It is straight-forward if our only metric is the value of the loss function of the problem for every configuration. The situation becomes more complicated if we add another metric – for instance, runtime. We may want to take runtime into consideration in case we want to balance good performance with reasonable training time. Two ways how to aggregate loss function and runtime cost to one metric are proposed in [11].

The first composed metric is *adjusted ratio of ratios* defined as

$$ARR_d(a_1, a_2) = \frac{\frac{P_{a_2}(d)}{P_{a_1}(d)}}{1 + AccD * \log(\frac{T_{a_2}(d)}{T_{a_1}(d)})}, \tag{2.10}$$

where a_1 and a_2 are compared algorithms, d is the dataset in question, $P_a(d)$ is performance of algorithm a on dataset d and $T_a(d)$ is runtime of algorithm a

on dataset d. *AccD* is a parameter which specifies how much accuracy we are willing to trade for a tenfold speedup. This function has two main advantages. Firstly, it is able to handle varying magnitudes of time and performance. Secondly, the *AccD* parameter allows us to specify the exact trade-off of time and performance we aim for.

However, this metric has proven to be problematic, as it was found that it is not monotonous in relation to the time ratio and, therefore, can produce incorrect rankings.

The second function is A3R defined as follows:

$$A3R_d(a_1, a_2) = \frac{\frac{P_{a_2}(d)}{P_{a_1}(d)}}{(\frac{T_{a_2}(d)}{T_{a_1}(d)})^q} \tag{2.11}$$

where now q is the parameter specifying the trade-off as well as adjusting the magnitude of time compared to performance. It should be a small number, value of $\frac{1}{64}$ is proposed in the paper. Although this function is less complex, it has the disadvantage of being less intuitive. Abdulrahman et al. [11] also provide evidence that incorporating A3R instead of simple performance in meta-learning leads to lower values of loss function earlier in the training process.

2.4 Combining meta-learning with hyperparameter optimization

According to Wistuba et al. [12], there is only one method to combine meta-learning and hyperparameter optimisation in tandem, which is to initialise a hyperparameter optimisation algorithm with the configurations proposed by meta-learning. The authors then continue to propose several strategies of how to choose configurations to initialise the hyperparameter optimisation algorithm.

Random best initialisation is a simple strategy which randomly selects k datasets from the past and uses their best-performing configurations to initialise the hyperparameter optimisation algorithm.

Nearest best initialisation uses k past datasets that have the most similar meta-features. It can be thought of as using the datasets which are given by the k-NN model.

Predictive best initalisation is similar to the previous strategy, only it determines similarity based on a regression model rather than distance. The regression model takes two meta-feature vectors and outputs a number representing their similarity.

Adaptive Predictive best initialization is again similar to the previous one, however, the regression model is updated with new meta-features during the initialisation process. The new meta-features are derived from the performance of evaluated configurations already used in the initialisation.

Active Adaptive Predictive best initialization alters the one above so that it uses first i initialisation configurations to find new meta-features.

Their experiments show that *Random best initialisation* is outperformed by every other and suggest that the performance of strategies grows with increasing complexity. However, their experiments did not include the usage of clustering.

2.4.1 Comparison of hyperparameter optimisation with and without meta-learning

Evidence has been provided by Feurer et al. [13] that meta-learning can significantly improve the performance of a hyperparameter optimization algorithm. Their experiment compared SMAC algorithm with and without meta-learning initialisation. To the best of our knowledge, there is little information in the literature on to what extent meta-learning can improve the performance of different types of algorithms.

Chapter 3

Realisation

The main goal of the experiment is to find out whether initialisation via meta-learning improves the performance of selected hyperparameter optimisation algorithms. We focus on hyperparameter optimisation algorithms SMAC, genetic programming and hyperband. These methods were chosen because they are commonly used for experimentation with good performance. We use Python because a lot of the methods needed are already implemented in it in commonly used machine learning libraries, such as scikit-learn. Therefore, the availability or ease of implementation in Python was a considerable factor. The experimental part has two main goals. One is to determine whether meta-learning initialisation improves the performance of hyperparameter optimisation algorithms. The other is to provide evidence for deciding which approach performs the best.

3.1 Experiment design

The experiment consists of two parts. The first, acquisition, consists of evaluating all datasets on all three hyperparameter optimisation algorithms without meta-learning initialisation. We save the best-performing configurations for each dataset.

For SMAC, the best performing configurations are the ones with the highest validation score. For genetic programming, the best performing configurations are Pareto front of validation score and complexity trade-off. The Pareto front was chosen to avoid overfitting and provide more diverse options for genetic programming initialised via meta-learning. We assume genetic programming is more sensitive to the lack of diversity than other methods since it derives new configurations from existing ones; therefore, the lack of diversity could lead to overfitting. For Hyperband, we save the configuration with the highest validation score in each bracket, since in most brackets, it is the only one trained on the biggest part of the datasets.

Table 3.1: Dataset properties

	Number of instances	Number of features	Number of classes
eeg-eye-state [14]	11235	14	2
phoneme	4053	14	2
satellite_image	48264	36	6
pol	11250	48	11
quake	1633	3	12
Heterogeneity	1200	20	2
Epistasis_2-Way	1200	1000	2
car-evaluation	1296	21	2
run_or_walk	66441	6	2
USPS	6973	256	10
bananas	3975	2	2
titanic	1650	3	2
coil2000 [14]	7366	85	2
delta_elevators	7137	6	26
ozone	1900	72	2
electricity	1455	33	2

The second part utilises the information acquired during the first phase. Again, all datasets are evaluated on all three hyperparameter optimisation algorithms. This time, however, hyperparameter optimisation algorithms are initialised via meta-learning. We use a leave-one-out cross-validation for evaluating this part.

For each algorithm we only consider configurations evaluated by that same algorithm in the acquisition phase. This decision was made because one of the goals of the thesis is to provide evidence of which approach performs the best overall. If one hyperparameter optimisation algorithm only had a superior performance when initialised by configurations found by a different algorithm, new datasets would have needed to be evaluated by both algorithms every time to maintain the consistency in the future.

A hard limit was set for each algorithm run to 1 hour for each dataset.

3.2 Components

3.2.1 Used datasets

All algorithms were evaluated on 16 classification datasets from openml.org. The criteria were mainly popularity defined by the number of posted usages and origin of the problem (artificial problems were excluded). More detailed description can be sen in table 3.1.

3.2.2 Used common libraries

3.2.2.1 Numpy

Numpy is a mathematical library used for scientific purposes, mainly array manipulation and random number generation.

3.2.2.2 Pickle

Pickle is used to serialise Python objects in order to save them into or load them from a file. We used it mainly to save configurations for further usage either in meta-learning initialisation or as data for evaluation of the experiment.

3.2.2.3 Pandas

Pandas is a widely used Python library for storing and manipulating table data. In Pandas, these tables are called Data frames, which retain most of the functionalities of two-dimensional numpy arrays, while also providing many other functions. These are mainly SQL-like commands, such as selecting and filtering, as well as certain aggregations. It has a built-in option to save to disc and load from it.

In the experiment, Data frames are used to store meta-features in tables and validation scores of evaluated configurations.

3.2.2.4 Scikit-learn

Scikit-learn is the core component of the experiment. It is one of the most commonly used machine learning libraries. It offers implementations of numerous machine learning algorithms and pre-processing methods with a consistent interface. It then allows the user to combine the algorithm with various pre-processors in a Pipeline class, which then represents one particular configuration.

3.2.3 Hyperparameter optimisation implementation

3.2.3.1 Auto-sklearn

Auto-sklearn is a library implementing SMAC, which supports initialisation via meta-learning and setting a time constraint. However, in its standard version, it does not support any alterations to its meta-dataset, as it uses its own database of solved machine learning tasks.

to the unavailability of an interface for meta-learning, the library is modified. The key class that require modification is *MetaBase*, which manages datasets, their corresponding meta-features, and past evaluations. In this part, only modifications relevant to building the meta-dataset are described. Changes for the meta-learning itself are discussed further in section 3.2.4.

The two important alterations were made to the constructor and *add_dataset* method. In the constructor, meta-features and configuration scores are loaded from the internal database. Post-modification they are loaded by Pandas from custom CSV files. *add_dataset* method in the unmodified version only adds meta-features of the currently evaluated dataset to a list of meta-features from the original database. The list is used only in this one evaluation and is not stored persistently. There is no way to use the newly added meta-features in future evaluations. Our modifications append current dataset meta-features to the existing meta-feature Data frame (if they were not already present) and the new Data frame is stored back on the filesystem.

3.2.3.2 TPOT

TPOT library provides an implementation of genetic programming for hyperparameter optimisation. It can be constrained by either the number of generations and individuals in each generation or time. Its main supporting library is deap, which implements a template for genetic algorithms, allowing the user to define various operations used during the algorithms. [15]

TPOT keeps Pareto-optimal cofigurations as an attribute by default.

3.2.3.3 Hyperband

Since we have not found any Hyperband implementation allowing to easily inject suggested configurations, Hyperband had to be implemented.

The first decision to be made was what to use as a resource. Due to low and inconsistent evaluation time for a single configuration, we have decided against time. Training iterations cannot be applied in our case, as different configurations are trained differently and sometimes even are not iterative. Therefore, the only remaining option was to use the number of datapoints. The halving factor was set to 3, as proposed by the authors in [2].

The implementation follows the pseudocode 1 from chapter 2, however, there are some notable implementation details. The sampling of configurations is done with the help of TPOT and its supporting library deap. First, we create a TPOTClassifier object and initialize it. This creates an object which has a method to generate a list of configurations of a given size. This allows us to generate an appropriate amount of configurations for each bracket using the TPOT generator. One minor modification has been made to the configuration space – limiting the maximum of hyperparameter k for k-nearest neighbours from 100 to 20 in order to prevent insufficient number of datapoints when evaluating configurations on very small subsets of datapoints.

Each evaluation is done by the evaluator of the TPOT object. Top configuration of each bracket is stored. The entire hyperband loop is repeated 20 times for one dataset.

3.2.4 Meta-learning

For meta-learning, we use the meta-learning infrastructure in auto-sklearn. The meta-learner takes dataset names and meta-features values from our modified MetaBase class 3.2.3.1, scales the input features using minmax scaling from 0 to 1 and uses them to fit a model. If any dataset name matches the currently evaluated dataset, its meta-features are discarded to avoid duplicity in the meta-feature dataset.

Meta-features used are all implemented in auto-sklearn. By default, some meta-features in auto-sklearn are excluded. That is because they were too computationally expensive for experiments done with auto-sklearn. In our experiment, however, slow computation of meta-features is not a factor, hence we included all meta-features provided.

3.2.4.1 *k*-NN

For *k*-NN we use the existing architecture of auto-sklearn. The *k* is set to 5. The number was chosen considering the total amount of datasets (16) and the need to suggest diverse options for the hyperparameter optimisation algorithm to avoid overfitting.

3.2.4.2 Clustering

Clustering uses the same architecture as *k*-NN. The main difference is that the model in class KNearestDatasets has been changed from scikit-learn.neighbors. NearestNeighbors to sklearn.cluster.KMeans. The number of clusters has been set to 3 so that the mean of the number of suggested configurations stays the same as when using *k*-NN. The minimum number of configurations in each cluster is 2 so that each dataset has at least one other dataset in its cluster.

3.2.5 Combining hyperparameter optimization with meta-learning initialisation

This subsection is dedicated to describing different initialisation methods for the hyperparameter optimisation algorithms. Since each algorithm is implemented by a different library, they warrant a different initialisation method.

3.2.5.1 SMAC

SMAC is initialised by the existing initialisation infrastructure in auto-sklearn. Again, the modifications were made mainly to the MetaBase class, namely the methods passing meta-features to the caller. The modification ensures compatibility with the custom meta-feature table.

Minor changes were made elsewhere so that the initialisation only takes into account the top 5 most well-performing configurations from each of the datasets suggested by meta-learning. This number was chosen because the default number of recommended configurations for auto-sklearn is 25, which is less than the number of datasets considered in this work.

3.2.5.2 Genetic programming

TPOT relies on deap to manage the configuration population. The population is created with a default deap *create_population* function. To initialise the population, *create_population* function was reimplemented to override default population.

The custom function reads the file with the names of the datasets recommended by the meta-learner in auto-sklearn and loads all their respective Pareto front configurations representations from the acquisition phase. It then compiles the representations into an array and returns it as the population in TPOT format. If there are more configurations than the required population size, some are randomly tossed. If there are not enough recommended configurations, the default deap function is used to generate random configurations to match the required population size. Default population size of 100 is used in the experiment.

3.2.5.3 Hyperband

Hyperband is specific in the sense that it does not evaluate all configurations in the same way. Configurations showing a good performance in brackets with a low s might perform poorly in brackets with a high s and a low budget. Similarly, configurations which win brackets with a low initial budget might take up precious space in brackets with a low number of configurations evaluated overall. Therefore, it warrants a different style of initialisation.

The difference in combining meta-learning with hyperband as opposed to the other algorithms is that it happens continuously across all brackets and not just in the beginning. At the start of each bracket, we add the winners of the respective bracket from the evaluations of datasets suggested by the meta-learner. The rest of the configurations in the bracket is then sampled randomly again using TPOT.

3.2.6 Evaluation methods

For comparing two algorithms we use the average difference of test or validation accuracies as well as the number of datasets where one method outperformed another one (*sign test* 3.1, where l_b and l_c are the loss function values of the baseline and the challenger respectively).

$$\sum_{i=0}^{n} sgn(l_b - l_c) \tag{3.1}$$

This decision was made because the average can be sensitive to outliers, while the sign test completely disregards the significance of how much better a method performed on individual datasets. More complex statistical tests can introduce arbitrary parameters or can have their assumptions violated, namely normality and commensurability. [16]

3.3 Results

In table 3.2, there are average test accuracies across all algorithms and meta-learning methods. As we can see, there is no improvement when algorithms are combined with meta-learning initialisation. For SMAC and GENETIC PROGRAMMING there is a very slight drop in accuracy and for HYPERBAND the performance drop is more apparent. The best-performing overall in the experiment is GENETIC PROGRAMMING,

Table 3.2: Average test scores of algorithms in %

	No meta-learning	With k-NN	With clustering
SMAC	80.578	80.277	80.159
Genetic programming	82.089	82.016	81.63
Hyperband	79.474	72.701	77.481

If we look at the results of the sign test in Table 3.3, we can see a similar pattern. Meta-learning initialisation mostly results in weaker performance, except GENETIC PROGRAMMING initialised using k-NN.

Table 3.3: Win/loss scores of meta-learning initialised algorithms against vanilla algorithms (on total of 16 datasets)

	With k-NN	With clustering
SMAC	-5	-2
Genetic programming	2	-4
Hyperband	-5	-1

This result likely has one of two causes. Either the chosen datasets are not similar enough so that the recommended configurations perform poorly on the new datasets, or the hyperparameter optimisation algorithms can reliably find well-performing configurations even without the help of meta-learning.

The former explanation can be evaluated by looking at the performance of the best configuration in early iterations of the algorithm. In Table 3.4 we can see the average of the best validation score from the first 25 evaluated configurations by SMAC. The difference is again very marginal and in favour of SMAC without the use of meta-learning. Please, note that neither GENETIC PROGRAMMING nor HYPERBAND can be evaluated like this, since GENETIC PROGRAMMING can be evaluated only after each generation, which does not provide enough granularity, and hyperband uses meta-learning continusously in each bracket.

Table 3.4: Average validation scores of the best of the first 25 configurations evaluated on each dataset by SMAC in %

No meta-learning	With k-NN	With clustering
81.717	81.599	81.565

We don't have enough evidence to conclusively decide whether the datasets are similar enough to be able to provide good configurations. If we added more datasets, we would be able to provide a more definitve anwser. However, that is beyond the scope of this work.

Another consideration is how common are configurations which perform nearly as good as the best configuration found. Figure 3.1 shows the accuracy of all configurations for each dataset individually evaluated sorted by performance. Genetic programming is not shown, since its population can be unrepresentative due to the fact that not all individuals are necessarily competitive.

We can see that in most cases, there is a large number of well-performing configurations. They are represented by the plateaus in the left part of the figure which contains well-performing configurations. This suggests that there is a large proportion of nearly optimal configurations which each algorithm is able to find.

If the hyperparameter optimisation algorithm can find the optimal configurations in a reasonable time by itself most of the time, the usage of meta-learning does not improve the performance. It only provides unnecessary complexity and can even lead to a slight performance drop overall.

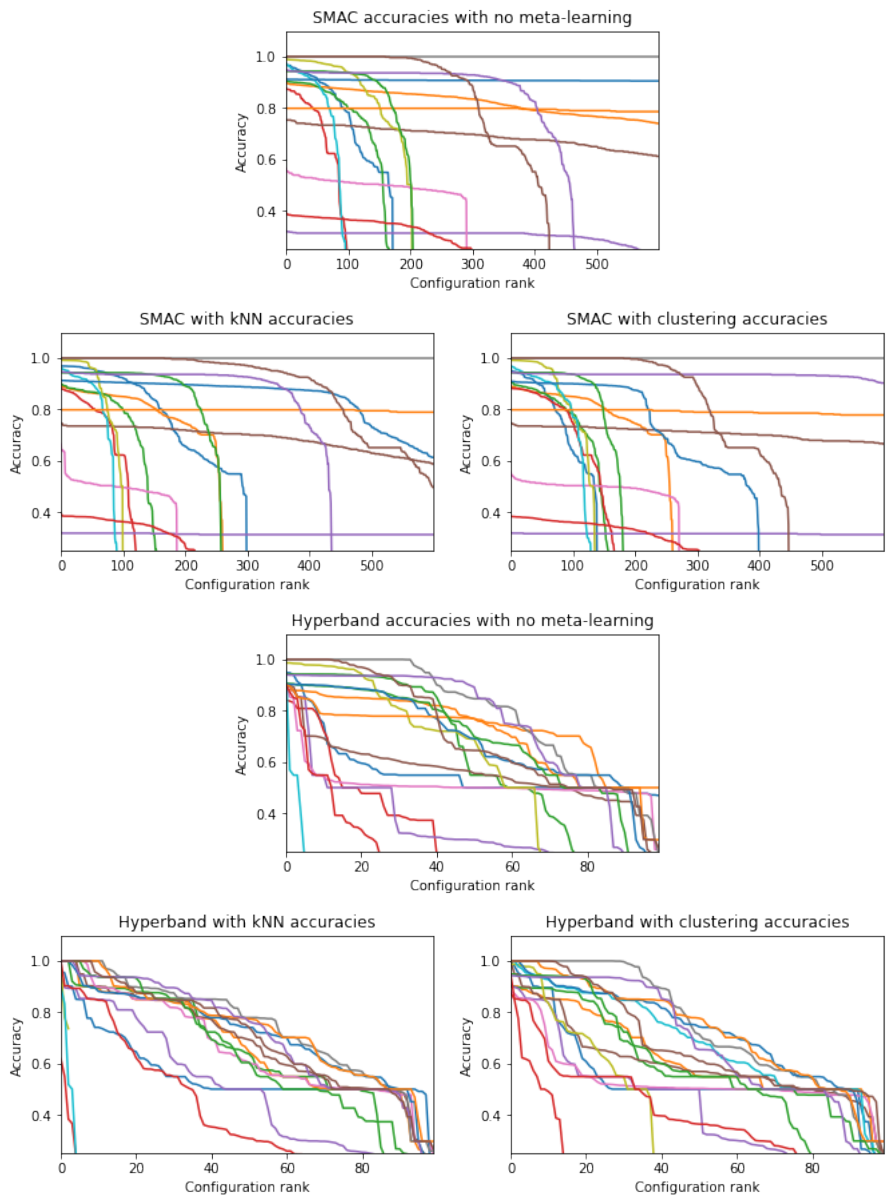
3.4 Discussion

The experiments have shown that the best performing algorithm for hyperparameter optimisation of the three evaluated in this thesis is genetic programming, since it has shown on average a better performance than the other two algorithms.

Unlike in [13], our results are much less optimistic regarding meta-learning. The authors' findings suggest that meta-learning improves the performance of SMAC during the early iterations, which we have been unable to reproduce. They also show that meta-learning can improve the overall performance of SMAC. The experiments have not found convincing evidence for this for none of the considered algorithms.

The varying results may be explained by two differences in methodology. Firstly, the paper in question uses more datasets in the experiment (57) compared to our relatively small number (16, which we have used for com-

Figure 3.1: Configuration performance distributions when using SMAC and HYPERBAND. Each line represents a single dataset. For HYPERBAND, only fully trained configurations are taken into account.



putational reasons). Therefore, there was a higher chance that meta-learning was able to find configurations that could perform well in the early stages as there was a higher change of a dataset with similar properties. Secondly, for evaluation the authors use the method of significant wins, which may not be suited for this type of problem according to [16] as it introduces an arbitrary parameter which creates an artificial filter for the results.

Further research may be needed in the area using a larger amount of datasets for meta-learning. Another direction the research can take is to experiment with meta-learning when optimising hyperparameters of a neural network. According to [13], performance improvement when using meta-learning increases in very high dimensional hyperparameter spaces, which can include the hyperparameter spaces of deep neural networks.

Conclusion

The thesis deals with the problem of optimizing hyperparameters of machine learning models and the usage of knowledge from previous tasks to speed up hyperparameter optimisation.

The theoretical part introduces machine learning problems, some of their terminology and the idea of optimising hyperparameters.

Throughout the thesis, we have looked at various hyperparameter optimisation and model selection methods. We have focused mainly on three of them, which we have used for the experiments: SMAC, which predicts which configurations have potential to improve the performance, GENETIC PROGRAMMING, which combines well-performing configurations to create potentially better ones, and HYPERBAND, which balances early-stopping advantages and drawbacks.

We have surveyed how knowledge from previous tasks can be utilised in machine learning. We have presented the idea of meta-features, their classification and how they can be used to determine the similarity of datasets. We have defined metric and explained how it is used in meta-learning algorithms. We have described two algorithms that can be used for meta-learning: clustering and k-NN. Then, we have discussed ways of combining these methods with hyperparameter optimisation algorithms and how runtime can be incorporated as a factor in optimization.

We have experimented using the meta-learning methods clustering and k-NN combined with SMAC, GENETIC PROGRAMMING and HYPERBAND. We have also designed and implemented integration of meta-learning with GENETIC PROGRAMMING and HYPERBAND.

The experiment which we ran has found no evidence that meta-learning initialisation of hyperparameter optimisation algorithms helps the overall performance. One possible explanation is that the meta-learning wasn't able to find well-suited configurations to recommend due to the relatively small number of datasets (used for computational reasons). However, further analysis implied that the reason is likely that the hyperparameter optimisation algorithm could find a relatively large number of configurations with nearly optimal performance in the time it was given even without the help of meta-learning.

In the case of HYPERBAND, meta-learning initialisation has even resulted in a decrease in performance. This implies the usage of meta-learning needs to be more carefully considered since it could hurt the overall performance if used inappropriately.

We have investigated the possibility that meta-learning initialisation can improve the early performance of hyperparameter optimisation algorithms. However, we have made no observations supporting the claim.

Our findings contradict some of the previous work done in this area and more research needs to be done on the issue.

Bibliography

1. KLOUDA, Karel. Organizace předmětu, představení tématu. *BI-VZD*. 2019.
2. LI, Lisha; JAMIESON, Kevin; DESALVO, Giulia; ROSTAMIZADEH, Afshin; TALWALKAR, Ameet. Hyperband: A novel bandit-based approach to hyperparameter optimization. *arXiv preprint arXiv:1603.06560*. 2016.
3. HUTTER, Frank; HOOS, Holger H.; LEYTON-BROWN, Kevin. Sequential Model-Based Optimization for General Algorithm Configuration. In: COELLO, Carlos A. Coello (ed.). *Learning and Intelligent Optimization*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 507–523.
4. OLSON, Randal S.; BARTLEY, Nathan; URBANOWICZ, Ryan J.; MOORE, Jason H. Evaluation of a Tree-based Pipeline Optimization Tool for Automating Data Science. In: *Proceedings of the Genetic and Evolutionary Computation Conference 2016*. Denver, Colorado, USA: ACM, 2016, pp. 485–492. ISBN 978-1-4503-4206-3. Available from DOI: `10.1145/2908812.2908918`.
5. MOHR, Felix; WEVER, Marcel; HÜLLERMEIER, Eyke. ML-Plan: Automated machine learning via hierarchical planning. *Machine Learning*. 2018, vol. 107, no. 8–10, pp. 1495–1515.
6. DRORI, Iddo; KRISHNAMURTHY, Yamuna; RAMPIN, Remi; LOURENÇO, Raoni; ONE, J; CHO, Kyunghyun; SILVA, Claudio; FREIRE, Juliana. AlphaD3M: Machine learning pipeline synthesis. In: *AutoML Workshop at ICML*. 2018.
7. LEMKE, Christiane; BUDKA, Marcin; GABRYS, Bogdan. Metalearning: a survey of trends and technologies. *Artificial intelligence review*. 2015, vol. 44, no. 1, pp. 117–130.

8. FEUERER, Matthias; SPRINGENBERG, Jost Tobias; HUTTER, Frank. Using meta-learning to initialize bayesian optimization of hyperparameters. In: *Proceedings of the 2014 International Conference on Meta-learning and Algorithm Selection-Volume 1201*. 2014, pp. 3–10.

9. KLOUDA, Karel; VAŠATA, Daniel. Metoda nejbližších sousedů, křížová validace. *BI-VZD*. 2019.

10. KLOUDA, Karel; VAŠATA, Daniel. Hierarchické shlukování a algoritmus k-means. *BI-VZD*. 2019.

11. ABDULRAHMAN, Salisu Mamman; BRAZDIL, Pavel; RIJN, Jan N van; VANSCHOREN, Joaquin. Speeding up algorithm selection using average ranking and active testing by introducing runtime. *Machine learning*. 2018, vol. 107, no. 1, pp. 79–108.

12. WISTUBA, Martin; SCHILLING, Nicolas; SCHMIDT-THIEME, Lars. Learning Data Set Similarities for Hyperparameter Optimization Initializations. In: *Metasel@ pkdd/ecml*. 2015, pp. 15–26.

13. FEURER, Matthias; SPRINGENBERG, Jost Tobias; HUTTER, Frank. Initializing bayesian hyperparameter optimization via meta-learning. In: *Twenty-Ninth AAAI Conference on Artificial Intelligence*. 2015.

14. DUA, Dheeru; GRAFF, Casey. *UCI Machine Learning Repository*. 2017. Available also from: `http://archive.ics.uci.edu/ml`.

15. OLSON, Randal S.; URBANOWICZ, Ryan J.; ANDREWS, Peter C.; LAVENDER, Nicole A.; KIDD, La Creis; MOORE, Jason H. Applications of Evolutionary Computation: 19th European Conference, EvoApplications 2016, Porto, Portugal, March 30 – April 1, 2016, Proceedings, Part I. In: ed. by SQUILLERO, Giovanni; BURELLI, Paolo. Springer International Publishing, 2016, chap. Automating Biomedical Data Science Through Tree-Based Pipeline Optimization, pp. 123–137. ISBN 978-3-319-31204-0. Available from DOI: `10.1007/978-3-319-31204-0_9`.

16. DEMŠAR, Janez. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine learning research*. 2006, vol. 7, no. Jan, pp. 1–30.

Appendix A

Acronyms

GP	Gaussian process
HTN	Hierarchical task network
***k*-NN**	*k* nearest neighbours
LSTM	Long short-term memory
ROAR	Random online aggressive racing
SMAC	Sequential Model-based Algorithm configuration
SMBO	Sequential Model-based Optimisation

Appendix **B**

Contents of enclosed SD card

```
readme.txt ...................... the file with CD contents description
experiment ........................... the directory with executables
  implementation.....................the directory wirh source codes
  Results.ipynb ...................... Jupyter notebook with results
paper ......................................... the thesis directory
  thesis.pdf.........................the thesis text in PDF format
  thesis .............the directory of LaTeX source codes of the thesis
```