Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science
Architecture of Information Systems Research Group

Automatic construction of machine learning pipelines

Master Thesis

Pieter Gijsbers

Supervisors:
dr. ir. J. Vanschoren
prof. dr. M. Pechenizkiy
dr. K.A. Buchin

Final Version

Eindhoven, October, 2017

Abstract

A lot of data is being collected, with the aim of developing good models from them. To construct a model, a machine learning pipeline is developed, which preprocesses the data and applies a machine learning algorithm. Unfortunately, this requires considerable knowledge about the use of machine learning algorithms.

Automated machine learning (AutoML) is an active research field, which in part aims to automate the process of finding good machine learning pipelines. This should allow non-experts to produce effective models. Additionally, machine learning experts can use these models to gain insight in their data, or have a starting point for further optimization.

One of the AutoML tools that has been developed recently is the Tree-based Pipeline Optimization Tool (TPOT). TPOT uses a genetic algorithm in order to optimize machine learning pipelines for any dataset. In this work two ways in which to improve TPOT are explored.

The first method is Layered TPOT (LTPOT). Due to the nature of the optimization algorithm used in TPOT, a lot of machine learning pipelines have to be evaluated. This is often an expensive operation. However, there is often correlation between the performance of a machine learning pipeline on a subset on the data, and that on the entire dataset.

This is used in Layered TPOT to find bad pipelines early, by evaluating each pipeline on a subset of the data. This is done in a layered structure, where the pipelines are evaluated on incrementally larger subsets of the data. There was no significant difference in the quality of pipelines found by TPOT and LTPOT. However, we find that LTPOT in many cases finds good pipelines faster. This makes LTPOT a useful improvement when working with time or resource constraints.

The second method is MetaTPOT. Meta-learning is the concept of learning from past experiments, and transferring this knowledge to new experiments. We conducted a small-scale benchmark of a few simple scikit-learn learning algorithms. Based on the results of this benchmark, we construct models which can recommend algorithms and hyperparameter settings to begin TPOT's pipeline optimization procedure with.

Three ways to construct recommendations are evaluated. Two of them recommend pipeline configurations based on a nearest neighbor approach. Given a new dataset, it finds the datasets most similar to it and recommends what worked best for them. The third approach trained a Random Forest Regressor for each baselearner, in order to predict the accuracy based on hyperparameter settings.

The nearest neighbor approaches worked best. They performed significantly better than random suggestions in many cases. However, it did not improve over the default version of TPOT which considers a larger array of learners and preprocessing steps. MetaTPOT results could improve further if the meta-models are trained with more data that includes learners and preprocessing steps currently included in TPOT but not in MetaTPOT. Nonetheless, this work shows that there is merit to incorporating meta-learning to warm-start TPOT.

Preface

This thesis presents the research I have done for my master thesis in the Data Mining group of Technische Universiteit Eindhoven. First and foremost I would like to thank my thesis supervisor, Joaquin Vanschoren. He helped through useful discussions and made sure to have time available to support me when needed.

The work I did builds on TPOT, it would not have been possible if not for its creators. In particular, I would like to thank Randy Olson for his insights during the development of Layered TPOT, and Weixuan Fu, who helped tremendously in solving bugs in TPOT.

This thesis is written using LaTeX with a template created and shared by J. Buijs and T. Nugteren, for which I would like to extend my thanks.

Finally I wish to thank everyone that supported me in other ways along the way, my family, my girlfriend, and my friends from all over the world.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Companies, individuals and institutions are collecting more data than ever before. The nature of this data varies wildly, from electrical signals in the brain to information about the demographics visiting a website. Not all this data is interpretable by humans, nonetheless there is valuable information latent in the data. For instance, brain signals can be interpreted in order to determine a crude approximation of what the brain is focusing on [11]. Information about the demographics on a website can be used to determine which advertisements should be shown, so that it is more likely that the visitor will click on them [48]. Clearly, constructing models from the data may create value. Unfortunately, the relationships in the data can be well-hidden and very complex, and problems may also arise through the sheer volume of the data. For many tasks, manually constructing good models, or even acceptable ones, is infeasible.

However, through the use of machine learning it becomes possible to find models which are useful. Machine learning has been applied to develop, among others, self-driving cars, automated medical diagnostics and recommendation systems.

1.1 Context

Machine learning is a technique which can automatically learn models from data. However, there are many machine learning algorithms to choose from. Not every algorithm will work well for all types of data, and it is not always clear which algorithm would work best for a specific dataset. On top of that, every machine learning algorithm has *hyperparameters* which define what the learned model can look like, how the model is learned, or both. These hyperparameters can heavily influence the performance of the resulting model. Picking the right algorithm and hyperparameters for your data can be hard, even for experts. The problem of choosing the best algorithm and hyperparameters is called the Combined Algorithm Selection and Hyperparameter optimization problem (CASH) [67].

Sometimes, it is helpful to first process the data before learning a model, for example by imputing missing values. The sequence of steps to go from input data to a learned model is also called a machine learning pipeline. Additional background knowledge about the choices made when creating a machine learning pipeline is presented in Section 2.1.

Recently, the combined problem of selecting preprocessing steps in addition to CASH is studied. New research often aims to automatically construct machine learning pipelines, and is often referred to automated machine learning (AutoML).

An aid in working towards solutions for the AutoML problem has been the concept of *meta-learning*. Meta-learning entails learning which machine learning algorithms work well on which datasets. In order to do this, first the performances of many learning algorithms are recorded on a wide array of datasets. Next, the datasets are characterized by their *meta-features*. Meta-features

are features of the dataset itself, such as the number of missing values found in a dataset, or the number of numerical attributes.

Knowing the meta-features of the dataset, as well as how various algorithms perform, allows a learner to generalize from this, and predict for a new dataset how well a certain learner will perform. Instead of comparing how various algorithms work on different data, the difference in performance for a single algorithm with various different hyperparameter configurations can also be observed. This way, in an analogous manner, a model can be learned of how hyperparameters affect the performance on a new dataset. Meta-learning has been successfully used in several applications for algorithm selection and hyperparameter optimization [26, 37, 67]. There are several ways to construct meta-features and create models from them, which will be discussed in Section 2.2.

One recent study which addresses the AutoML problem is TPOT, a Tree-based Pipeline Optimization Tool [54]. TPOT is a tool which automatically constructs a machine learning pipeline for any given dataset. When building this pipeline, TPOT considers several machine learning algorithms, as well as various preprocessing steps to perform on the data. Any combinations of preprocessors and learners may be used, allowing for great flexibility.

While the structure of pipelines in TPOT allows for great flexibility, TPOT strives to keep the number of steps in the pipeline small. This helps with keeping pipelines interpretable. Despite its upsides, TPOT leaves room for improvement. For instance, while meta-learning has successfully been used for AutoML tools, such as Auto-sklearn, TPOT does not use it. This study focuses on improvements to TPOT, therefore the next section will give more details about TPOT, to help understand the research problems. In Section 2.4, the inner workings of TPOT are explained in depth.

1.2 TPOT

TPOT was developed with two goals in mind. The first goal is to make machine learning accessible for people without expert knowledge in the field of machine learning. As it is, knowing which preprocessing steps, algorithm and hyperparameter configurations will work well on the data comes mostly with experience. With the rise in data collection, lowering the barrier to entry for machine learning will help more people and businesses create value from their data. The second goal is to speed up the work of machine learning experts. Even for experts it can sometimes be hard to tell what will work well for a given dataset. Using TPOT to generate one or several good models may immediately provide some insight on the data, or a starting point from which to tune the model.

At the core of TPOT is an evolutionary algorithm, a type of algorithm which is inspired by biological evolution. An evolutionary algorithm tries to find the best solution to an optimization problem. In biological evolution the aim is to create individuals most likely to procreate. In the case of TPOT, it aims to find the machine learning pipeline which optimizes a performance measure, such as the accuracy of the model constructed by the pipeline.

To find this optimized pipeline, the algorithm starts out with a *population* of pipelines and evaluates how well they score according to the performance measure which needs to be optimized. The pipelines that score poorly will be discarded, and those that perform well will be considered further. After picking the pipelines to keep, TPOT makes adjustments to them through either *mutation* or *crossover*. Which pipelines get modified, and in which manner, is decided randomly, but in the end there is a new pool of pipelines. From this point on, the process repeats itself, evaluating the pipelines, keeping some, and modifying them, iteratively trying to find the best pipeline.

1.3 Research Problem

The main goal of this work is to improve the performance of TPOT, both through the lens of computational efficiency, and that of resulting model performance. This will allow TPOT to find better models faster.

The effectiveness of the following ways to improve TPOT are explored:

- Discard bad pipelines after evaluating them on only a subset of the data.
  Machine learning algorithms and data preprocessing steps can be very slow, and datasets can be very large. The nature of the evolutionary algorithm means that many machine learning pipelines will be evaluated, where each might need substantial training time. In other work, the performance of algorithms on large datasets has been estimated based on how well they performed on a small part of the dataset [70]. It should be explored whether or not this concept works well in the context of TPOT, because if so, it can make TPOT less computationally demanding. The work on this problem resulted in Layered TPOT (LTPOT), and is presented in Chapter 3.
- Create pipelines based on meta-learning.
  Currently, candidate pipelines are created by randomly picking preprocessing steps and a learner. As has been shown in other work, meta-learning can be used to recommend learners with corresponding hyperparameters to achieve better performance than randomly picking a configuration [60, 1]. Instead of randomly generating new pipelines, they could be created based on recommendations from a meta-learner. The work on this problem resulted in MetaTPOT, and is presented in Chapter 4

1.4 Document Outline

First, background information will be presented in Chapter 2. Section 2.1 illustrates the various choices to be made when creating a machine learning pipeline. Section 2.2 will give a general overview of meta-learning research, where Section 2.3 delves into how it is used for various CASH solutions. Finally, Section 2.4 gives a thorough introduction to genetic algorithms and how they are applied in TPOT. Then, the work on discarding pipelines based on experiments on subsets of data is laid out in Chapter 3. The proposed changes to TPOT regarding this modification are laid out in Section 3.1, and the method of experimental evaluation and results follow in Section 3.2. The use of meta-learning to initialize the first initialization is discussed in Chapter 4. Finally, the work is summarized in the conclusion, with an outlook on future work.

Chapter 2

Preliminaries

This section gives an introduction to AutoML and provides the reader the background information that they should know to understand the thesis. First, a brief introduction to machine learning and machine learning pipeline construction is given. This helps the reader understand the problem automated machine learning tools, such as TPOT, aim to tackle. With that knowledge, the concept of meta-learning is explained and its literature is discussed, as one goal of this study is to improve TPOT through meta-learning. Then, various other approaches to algorithm selection, hyperparameter optimization and the CASH problem are reviewed. Finally, a more in-depth introduction to TPOT is given, so that the proposed modifications may be easier to understand.

2.1 Machine Learning Pipelines

In a machine learning problem, a machine learning algorithm is applied to automatically construct a model from data. For example, given a dataset with the height, weight and gender of a set of people, one can automatically construct a model predicting the gender based on height and weight.

This predictive model is created by a machine learning algorithm, such as the *k-nearest neighbors* algorithm (k-NN) [2]. When a k-nearest neighbor model is asked to predict the gender of a new height-weight pair, it will look up the k height-weight pairs that are most similar to it. These nearest height-weight pairs are also called the *neighbors*. The algorithm will look up the gender of these nearest neighbors and, for the new height-weight pair, predict the gender that is most frequent amongst the neighbors.

In some cases, a *preprocessing step* may be applied to the data, before using a machine learning algorithm. This preprocessing can help the machine learning algorithm learn faster or better. Examples include normalization or removing noisy outliers (eg. an entry of a person with a height of 4 meters is likely erroneous). A sequence of preprocessing steps and machine learning algorithms to create a final model is called a *Machine learning pipeline*. The following sections will give explanations of the various choices to be made in designing a machine learning pipeline.

2.1.1 Picking a Machine Learning Algorithm

In Figure 2.1, a fictional dataset is given with the height and weight of 30 people. The 20 blue markers represent the males, the 10 magenta ones represent females. While the males in the dataset are typically larger and heavier than the females, this is not always the case.

In order to learn a model, a choice of machine learning algorithm has to be made. In this example, we will use the k-NN algorithm described above. If one is given a new height-weight pair, illustrated in Fig. 2.1 by the green marker, the k-NN model can do a prediction for the gender of this pair.

At this point, the expert using the k-NN has to pick a value for k. If the expert picks k to be 1, then the algorithm will only look up the single point that's closest to it. In this case, the nearest neighbor is female, so the model predicts the new pair to be female too.

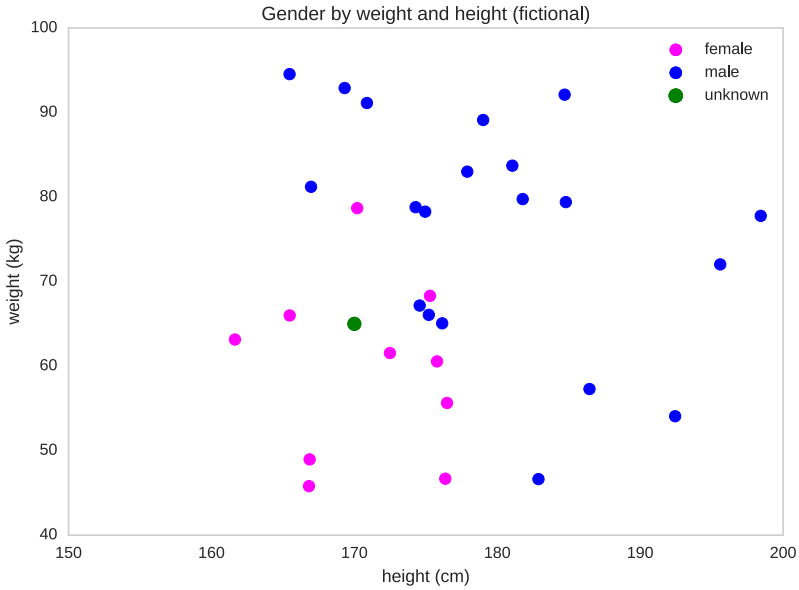


Figure 2.1: A fictional dataset of males and females by weight and height.

In creating a model using machine learning algorithms, choices have to be made. The first choice is to pick the algorithm, for example a *Decision Tree* [59], *Random Forest* [34] or *k-nearest neighbors* [2]. After picking the algorithm, a configuration of *hyperparameters* also has to be chosen. Hyperparameters are the parameters that the algorithm uses when learning the model. In the case of k-NN, k is a hyperparameter. The effect of the hyperparameter often affects the final model.

The hyperparameter k in k-NN determines the number of nearest neighbors to be looked up when making a prediction. In the gender prediction example, we saw that when $k = 1$, the new pair is predicted to be female.

Instead, if k was chosen to be at least 21, the model will always predict new pairs to be male. This follows from the fact that whatever the height-weight combination is, there are only 10 females in the dataset. Thus, when looking up the 21 nearest neighbors, at most 10 of them will be female, which means the majority prediction will be male. In reality, some versions of the k-nearest neighbor algorithm will take into account the distance to each neighbor, which can prevent these problems somewhat. In general however, this shows that performance can vary based on the hyperparameter configuration. This often results in studies about the optimization of hyperparameters for specific applications [17, 74, 3].

The model created by the machine learning algorithm is typically not perfect however. How good the model is depends on several factors. As illustrated above, hyperparameter values affect the performance of your model. Furthermore, the amount of data available also determines how good the model can be.

Indeed, two people can have the same height and weight, while one is male and the other is female. In that case, there would not be one perfect prediction for the given height-weight combination. Thus, in order to construct a better model, more attributes are needed.

In another case, it may be that the new sample is close to the decision boundary. In this case,

data about more people will help create a more defined boundary between males and females, also called the *decision boundary*.

In some cases useful information may already be present in the data, but the machine learning algorithm may not be able to use it directly. *Preprocessing* steps help transform the data, so that machine learning algorithms can potentially learn better models from it.

2.1.2 Preprocessing

Consider again the dataset which contains the height, weight and gender of 30 people. This time however, the height of the people in the example has been given in millimeters. Given a new height-weight pair (1630mm, 95kg), it would be in the top-left of Figure 2.1. Its nearest neighbor, had height been in centimeters, would be the male with height and weight roughly 167cm and 94kg, respectively - 4 centimeters taller and one kilogram lighter. In contrast, the female that has a height of 163cm, but weighs only 63kg, is of equal height but is 32 kilograms lighter.

However, the height was given in millimeters instead of centimeters. This means k-nearest neighbors sees the male as being 40 units of height away, and 1 unit in weight. On the other hand, the female is 0 units in height away, and 32 units in weight. In this case, the female is regarded as the closest neighbor, and the prediction (if k is 1), is that the new example is female.

Transforming the data, converting the height data from one representation to another e.g. from millimeters to centimeters, can affect model performance. The act of transforming the data is also called 'preprocessing the data', because it happens before the data is used as input to the machine learning algorithm. Picking one or more preprocessing steps is also complicated by the fact that, like the machine learning algorithms themselves, they often have hyperparameters too.

There are many ways to preprocess data. The example above used a scaling preprocessing step, of which there are many variants, such as 'min-max scaling' and 'normalization'. Other preprocessing steps include *feature extraction* and *feature selection*. Feature extraction aims to extract or create new features from the data, for example by multiplying values of two existing features. Feature selection aims to determine which features carry little to no information about the attribute to be predicted and discard them accordingly.

2.1.3 Creating Good Pipelines

In conclusion, creating a model from data using machine learning involves a lot of decisions. From picking the machine learning algorithm to choosing which preprocessing steps to apply, in addition to picking hyperparameter configurations for each step. The result of all these decisions is called a *Machine learning pipeline*. The machine learning pipeline specifies which preprocessing steps to take, in which order, which machine learning algorithm to apply, and with which hyperparameter configurations each step should be executed.

For example, "First normalize the data, then create a k-nearest neighbor model with k=3" is a machine learning pipeline with two steps: a preprocessing step (normalization) and creating the model (k-nn). Machine learning pipelines can become complex constructions, for example learning multiple models and combining their predictions into one.

Knowing which preprocessing steps work well with which machine learning algorithm comes, in part, with experience. However, even for experts it can be hard to predict which preprocessing steps or machine learning algorithm will create a good model for the data. Often, the optimization of a machine learning pipeline comes down to trying various configurations to see what works well. Human expertise can help decide which hyperparameter configurations or algorithms to evaluate somewhat. But even then, trying various algorithms and hyperparameter configurations is a time consuming and sometimes labor intensive task.

The field of automated machine learning (AutoML) aims to automate the creation of machine learning pipelines. It approaches the problem of pipeline creation as an optimization problem. The performance of the final model created by the pipeline is optimized by picking the right algorithms and hyperparameters. The automatic construction of pipelines can make machine learning solutions available to non-experts. For experts, AutoML tools can suggest good machine learning pipelines, which can be tuned further guided by human intuition. Research in AutoML is discussed with more depth in Section 2.3.

As discussed, when creating machine learning pipelines several configurations are often evaluated. During this process, one might find that an algorithm works well with a certain hyperparameter configuration on a given dataset. It stands to reason that if the same algorithm and hyperparameter configuration is used on other, but similar, data, the performance is likely similarly good. This concept of carrying over this information from one experiment to another is called meta-learning. The next section will give an introduction to meta-learning, and discuss its literature.

2.2 Meta-learning

Meta-learning is the process about learning how machine learning algorithms perform across a range of tasks. It aims to learn which algorithm will work well for a dataset with certain characteristics, or which hyperparameters will give a good performance.

In the last section, it has been illustrated that there are many decisions to make when creating a machine learning pipeline. Even when limiting the search space of potential pipelines, it will increase exponentially for each allowed preprocessing step or additional hyperparameter. This quickly makes trying all combinations intractable.

If a meta-model can predict what data characteristics make a certain machine learning algorithm work well, it can be used to prune the search space. It can immediately suggest potential good configurations or discard bad ones. Meta-learning is applied to perform algorithm selection, as well as hyperparameter optimization.

The workflow to create a meta-model is shown in Figure 2.2. In order to train a meta-model, a *meta-dataset* from which to learn is required. A meta-dataset contains information about machine learning experiments. For instance, it captures which machine learning algorithm is used, with which hyperparameter configurations, and how well the resulting model performed. Additionally, for each such experiment it also describes that dataset on which the experiment was performed. The description of the dataset is done by *meta-features* that capture information about the data, such as the number of attributes.

The first step to create a meta-dataset is to select a set of datasets to perform machine learning experiments on. This is illustrated as the first step in Figure 2.2.

In the second step two actions have to be performed on each dataset. On the one hand, the meta-features of the dataset need to be calculated. Meta-features describe the dataset in various ways. Examples include the number of classes in the dataset, the mean skewness of numerical features or the accuracy of simple classifiers. More information about the various meta-features follows in the next section. On the other hand, machine learning experiments have to be run on the datasets. The type of machine learning experiments that are run should be the same type you want the meta-model to make predictions about.

In the final step, the results of step two are combined into a single meta-dataset. This dataset contains a row for each machine learning experiment, describing the algorithm and hyperparameters used, the achieved performance and the meta-features of the dataset on which they were achieved.

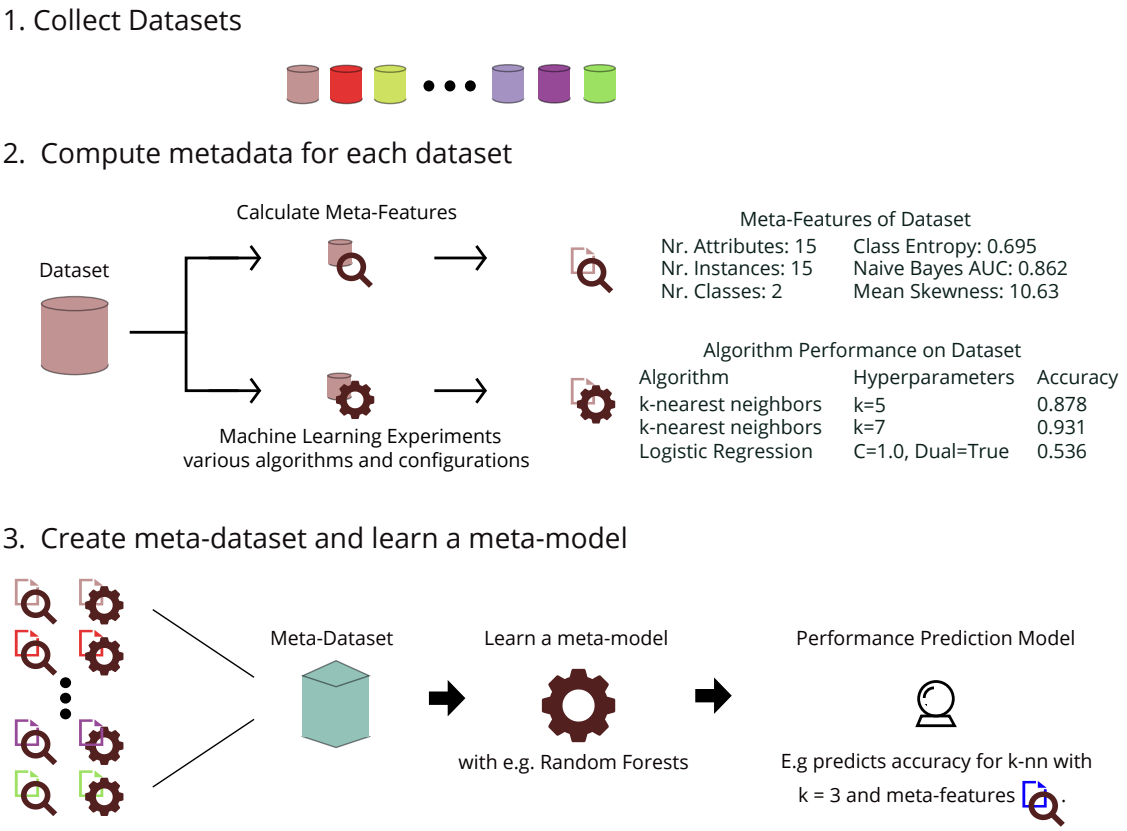


Figure 2.2: An overview of the steps to construct a meta-model.

We can then use a machine learning algorithm to create a model that predicts the performance of a learner, given the meta-features of a dataset and the desired hyperparameter configuration. In theory any learner can be used, but in practice Random Forests work well [69], and nearest neighbor algorithms are also sometimes used [27].

2.2.1 Meta-Features

To create an effective meta-model, the meta-features used have to describe the datasets well. The features should be good predictors of the relative performance of algorithms

Several different types of meta-features have been developed, ranging from simple features such as the number of samples in a dataset, to more complex ones. Most meta-features fall into one of the following groups [13, 57, 73]:

Simple, Statistical and Information-theoretic

These features sometimes appear separately, as they have mixed origins. Simple features are quick to extract, such as the number of samples in the dataset, or the proportion of samples in the majority class. Statistical features include information about the numeric features, such as the mean skewness of all features. Finally, mean entropy of attributes and class entropy are examples of information-theoretic features.

Landmarking

Characterizes the dataset by how simple machine learning algorithms perform on them. Examples include the performance of a decision stump, naive bayes classifier or linear discriminant analysis [57, 4].

Alternatively, more complex algorithms can be used on a subset of the data. To allow these meta-features to be calculated quickly, constant-sized training set is used and the performance is measured based on a fraction of the data. These types of landmarks are called *sampling-based landmarks* [64]. Algorithms used in [64] for sampling-based landmarking include decision trees, neural networks and an ensemble of boosted trees.

Good landmarkers should probably have a runtime complexity of at most $O(n \log n)$. Otherwise, calculating landmark features could become more expensive than executing the complex learner itself [57].

Moreover, when using multiple landmarkers, they should have different biases [57, 4]. If two distinct landmarkers have similar performance across all datasets, then it probably suffices to only use one of them [31].

Model-based

Similar to landmarking, model-based features are constructed by running machine learning algorithms on the dataset. However, the features are based on the model constructed by the learner.

For example, the decision tree algorithm can be used to construct model-based meta-features. First, a decision tree is learned from the data. Then, characteristics of this decision tree are the meta-features. Examples include its depth or the number of leaves. Moreover, there are several characteristics of which the minimum, maximum, mean and standard deviation are used, such as the branch-length, the number of nodes per level or the number of times an attribute is used in a split [5, 56, 61].

2.2.2 Meta-Learners

Finally, after constructing a meta-dataset by calculating meta-features and running machine learning experiments on a set of datasets, a final model has to be learned. The algorithm used will depend on the desired output of the model, the *meta-target*. In the context of using meta-learning for algorithm selection, [14] describes four different kind of meta-targets are described:

- Best algorithm in a set. The only task of the meta-learner is to predict which algorithm will perform best on the task. The advantage formulating the target this way, is that this turns the problem into a classification problem - a topic which has been studied extensively and for which a wide arrange of algorithms and implementations are available. The disadvantage is that there is no guarantee that the recommended algorithm is indeed the best one, and if it performs poorly there are no recommendations on what algorithm to try next [14].
- Subset of algorithms. Instead of recommending a single algorithm, recommend all algorithms expected to perform well. Often, some measure is used to determine if the expected performance of an algorithm is not much worse than that of the best one. If the expected performance is close enough to the expected performance of the best algorithm, it is included in the recommendation. The advantage of this method is that multiple algorithms are recommended, which means one algorithm performing poorly does not create a problem. The disadvantage is that it can not tell which algorithm to try first [14].
- Ranking of algorithms. Similarly to the subset of algorithms, multiple algorithms are recommended. However, a ranking is constructed for these algorithms, indicating the expected order of performance. Several types of rankings can be considered.

  Rankings can be complete or incomplete. A complete ranking assigns a rank to each algorithm, while an incomplete rank only assigns a rank to some algorithms. Sometimes the

expected performance of an algorithm is very uncertain, in which case it might be better to not assign it a rank.

Rankings can also be linear or quasi-linear. A linear ranking has a strict ordering, where each algorithm is assigned a distinct rank. In a quasi-linear ranking, also called a weak ranking, some algorithms may share a rank. This allows the metalearning model to express when two algorithms are expected to have similar performance.

The advantage of a ranking is that there is an ordered set of suggestions to try. Additionally, a user may have preferences about the algorithm used, based on e.g. computational efficiency or model interpretability. Based on the ranking the user may make an informed decision about using their preferred algorithm, if its performance is ranked not much worse than the top ranked suggestion [15]. A disadvantage of using ranking algorithms is that there is significantly less work on learning rankings [14].

In [13], a C4.5 decision tree algorithm is used to construct rules for each algorithm. Then, each branch in the tree is a rule, and each rule is given an information score based on its usefulness. An example of such a rule would be "If the number of samples in the dataset is smaller than 1000, the Discriminant Algorithm is applicable". Finally for a new dataset, all rules are considered and the sum of information scores for rules that are satisfied indicates the usefulness of each algorithm.

Ranks can also be constructed based on k-nearest neighbor [15, 26].

- Estimates of performance. The final meta-target is to predict an estimate of performance. This makes the problem a regression problem, which also has been studied extensively. To predict the performance of the machine learning algorithms, one regression model per algorithm has to be trained. This has the advantage that it is easy to add or remove algorithms from consideration, by training a new model, or discarding an old one, respectively. Moreover, it gives more information to the user, such as what performance to expect or how big the difference is expected to be between algorithms [14].

In the context of using meta-learning for hyperparameter optimization the techniques can be adapted to pick algorithm settings instead of algorithms. For selecting good initial sample points for *Bayesian optimization*, a technique discussed in the next section, nearest neighbors is used [27].

Meta-learning has also been used with algorithm runtime as meta-target. For this purpose, ridge regression and random forest regressors were found to be the best meta-learners [38]. The result is a model with good, robust performance when predicting the runtime of an algorithm.

Moreover, meta-learning has also been applied in the context of algorithm selection for data-streams [69]. In this case, meta-features are calculated on a window of 1,000 observations in the data stream. A Random Forest classifier is used as meta-learner to predict which algorithm will perform best for a given data stream window.

2.3 Automated Machine Learning

Automated machine learning (AutoML) consists of automatically choosing the right algorithm, performing hyperparameter optimization and possibly pick preprocessing steps for a machine learning problem. At first, hyperparameter optimization and algorithm selection were often researched as separate problems.

When the problems were first combined, it was often named Combined Algorithm Selection and Hyperparameter optimization (CASH), a term introduced in [67]. More recently, the problem receives attention under the name AutoML, short for automated machine learning, at Machine Learning conferences such as ECML-PKDD and ICML. Recently, the focus has shifted to include

preprocesing steps in addition to algorithm selection and hyperparameter optimization.

As discussed before, algorithm selection is often done with a meta-learning approach. Hyperparameter optimization for machine learning algorithms is also a well researched topic, and the next section will discuss its literature. The section after that will give an overview of the work on the combined problem of algorithm selection and hyperparameter optimization.

2.3.1 Hyperparameter Optimization

Hyperparameter optimization is the act of finding the hyperparameter settings which optimize the performance of an algorithm with respect to a certain performance measure. There are many factors which make hyperparameter configuration for machine learning algorithms a challenge [20]:

- Hyperparameters can exhibit nonlinear and non-convex behavior. Having very small k in the k-nearest neighbor algorithm may overfit, reducing performance. On the other hand, having high k may underfit, again reducing performance. Somewhere in-between is where performance is most likely to be best. Even then, in some cases varying k by small amounts has no measurable impact on performance.

- Hyperparameters can be integer, continuous or non-numeric. Examples of integer hyperparameters are the value of k in k-nearest neighbors and the maximum depth of a decision tree. Continuous hyperparameters include regularization for Support Vector Machines and the learning rate for neural networks. Finally, non-numeric hyperparameters exist in e.g. the k-nearest neighbor algorithm there can be multiple ways to calculate distance, such as the 'Manhattan Distance' and 'Euclidean Distance'. For decision trees, there are multiple ways to measure the quality of a split, like 'Gini impurity' and 'Entropy'.

- Learning algorithms can have conditional hyperparameters. For instance, when using scikit-learn's implementation of Logistic Regression, one has to specify which solver is used, which in turn can limit the available options for the 'penalty' parameter.[1]

- Function evaluations can be expensive. Depending on the size of the dataset and the machine learning algorithm or its configuration, evaluating many different hyperparameter settings may be prohibitive. While some models train in minutes, others can take days or even weeks.

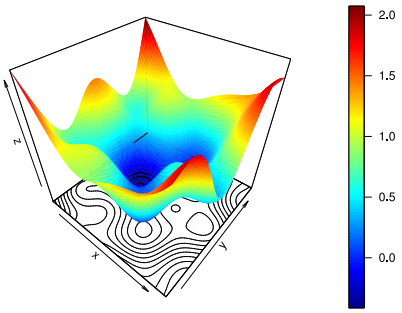
Multiple approaches to hyperparameter optimization can be found in the literature. Figure 2.3 gives a visualization of three hyperparameter optimization methods discussed hereafter.[2] The visualization uses the function $f(x, y) = sin(x) * 0.5 * cos(y) + 0.04 * (x^2 + y^2)$, but it is analogous to a possible performance response from changing two hyperparameter settings. In Figure 2.3a values for the function f are shown for $x, y \in [-5, 5]$. In Figures 2.3b-2.3d, the black dots indicate points which are sampled by the optimization method. For each method, 25 sample points are illustrated.

Grid Search

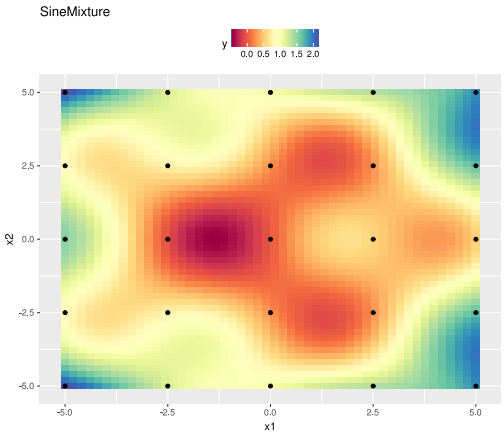
Grid-search is the act of performing an exhaustive search over all possible discretized combinations [36]. Figure 2.3b shows an example for a search grid, where every combination $(x, y) \in \{-5, -2.5, 0, 2.5, 5\}^2$ is tried. Good configurations are found, notably $(-2.5, 0)$ and $(0, 0)$, but the grid misses some of the best configurations. At the same time, many function evaluations take place in bad areas of the search space (where either x or y is -5 or 5).

[1] http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

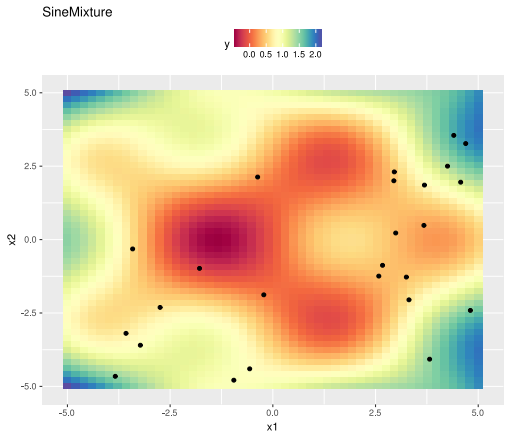
[2] Figures with permission from https://mlr-org.github.io/First_release_of_mlrMBO_the_toolbox_for_Bayesian_Black_Box_Optimization/



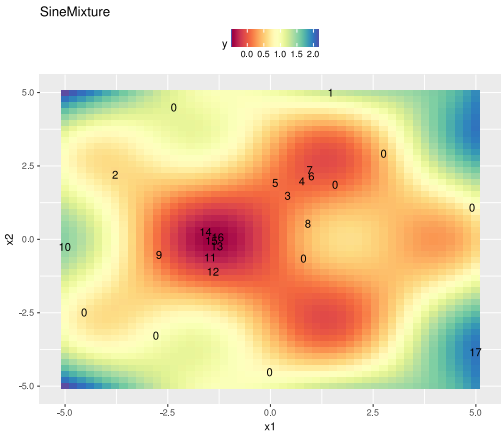
(a) Function to optimize: $f(x, y) = sin(x) * 0.5 * cos(y) + 0.04 * (x^2 + y^2)$.



(b) Example sample points of grid search.



(c) Example sample points of random search.



(d) Example sample points of search with Bayesian optimization.

Figure 2.3: Examples of sample points for various hyperparameter optimization methods.

Grid-search is sure to find good hyperparameter settings if the grid contains enough sample points. However, the number of possible settings grows exponentially with the amount of hyperparameters. Thus, the grid needs to be very coarse for a large search space, and can be infeasible for algorithms with a large number of hyperparameters. As the grid becomes more coarse, it becomes easier to miss good hyperparameter configurations with grid-search. Moreover, since some machine learning algorithms take a lot of resources to train, exhaustively evaluating a bad part of the search space only wastes a lot of resources.

Random Search

An alternative to Grid Search is *Random Search*, which explores hyperparameter settings at random. It has been shown that Random Search is more efficient for hyperparameter optimization than Grid Search, both empirically and theoretically [6]. This partially stems from the fact that for most datasets, only a few hyperparameters matter. The hyperparameters that matter differ per dataset, so grid search can't be adjusted accordingly. An example of possible sample points for Random Search is shown in Figure 2.3c.

Bayesian Optimization

A more advanced method for hyperparameter optimization is *Bayesian Optimization* [63]. Bayesian optimization iteratively chooses sample points based on earlier results, balancing between exploration and exploitation.

In Figure 2.3d, you see an example of points sampled by Bayesian optimization. Each point is represented by a number that indicates at which iteration it was sampled. The initial iteration included 8 samples (denoted by zeros), each subsequent iteration only had one sample.

Bayesian optimization constructs a probabilistic surrogate model for the optimization function. It models the likelihood of various optimization functions, so that the expected performance based on the hyperparameter configuration can be determined. There are many possible surrogate models, but a Gaussian-process is often used.

These surrogate models are updated based on evidence of the true optimization function. This evidence comes from performing a function evaluation (machine learning experiment). Every time a function evaluation presents new evidence, the beliefs of the possible optimization functions and their likelihoods are updated.

The initial belief is constructed by taking at least two samples randomly. This allows the construction of a distribution of values the optimization function is likely to have. Then, an *acquisition function* will determine which hyperparameter settings to try next, based on the beliefs about the optimization function. The acquisition function has to balance exploration and exploitation. On the one hand, it should explore areas which are likely to contain good hyperparameter settings. At the same time it should make sure not to leave areas with possibly bigger improvements unexplored.

An example of Bayesian optimization on a 1D problem can be seen in Figure 2.4 from [18].

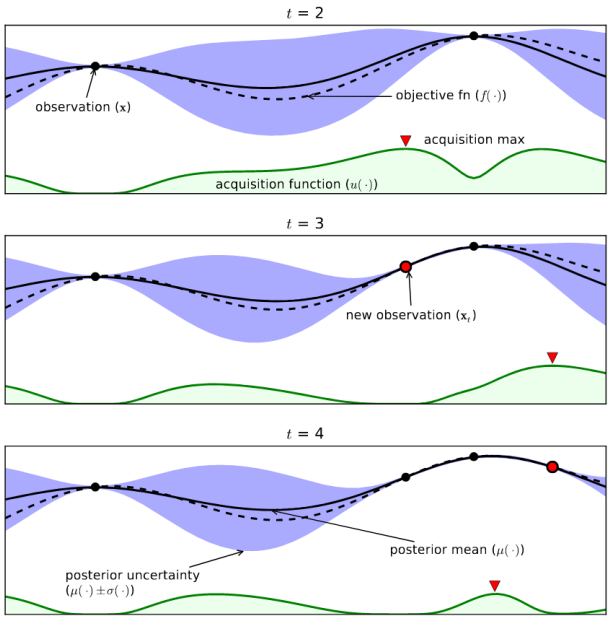


Figure 2.4: An example of Bayesian optimization on a 1D problem.

In each subfigure, the solid black line indicates the mean prediction of the objective function, and the blue shaded area is the mean plus and minus the variance. The dotted line is the true objective function. In the bottom, the acquisition function values are plotted in green.

In the first subfigure of Figure 2.4, we see that a prediction of possible objective functions is made based on two sample points. The belief is that the objective function is maximized close to the left or right of the observation on the right. The acquisition function suggests to try a new hyperparameter value to the left of the observation.

Based on the observation from this sample, the prediction of possible objective functions is now updated, as seen in the middle figure. At this point, the acquisition is maximized by picking a value on the right. Again, the predictions are updated. In the last subfigure we see that even though most of the left is unexplored, just enough is known to determine it is unlikely to be better than hyperparameter values on the right.

Bayesian Optimization has been shown to outperform Random Search [10]. These results are obtained even though the addition of an acquisition function makes it less efficient to determine the next configuration to sample. This is easily explained however, as [63] suggests: "when evaluations of f(x) are expensive to perform - as is the case when it requires training a machine learning algorithm - it is easy to justify some extra computation to make better decisions.".

Bayesian Optimization is a Sequential Model Based Optimization (SMBO), because it uses a model of possible objective functions to determine what good hyperparameter values may be, and sequentially updates this model based on new findings. Originally SMBO techniques were not suitable for algorithm selection due to limitations, such as only supporting numerical parameters.

In [37], SMBO was adapted to make it work for the algorithm configuration problem, in particular to find configurations for SAT solvers. They introduced two methods, Random Online Aggressive Racing (ROAR) and Sequential Model-based Algorithm Configuration (SMAC). SMAC was able to find state-of-the-art algorithm configurations.

An SMBO method similar to SMAC is Spearmint. Like SMAC, Spearmint also models the

probability of the loss given a hyperparameter configuration. However, unlike SMAC, it does not allow for conditional hyperparameters.

A surrogate model that does not model the loss given a hyperparameter setting explicitly is the Tree-structured Parzen Estimator (TPE) [9]. Whereas a Gaussian-process surrogate models the probability of a certain loss given a hyperparameter setting $p(y|x)$, the TPE models both $p(y)$ and $p(x|y)$. That is, the TPE models both the probability a certain loss is achieved and how likely a certain loss is caused by a certain hyperparameter setting. This also allows a term proportional to the expected improvement to be calculated, to help determine the best hyperparameter configuration to sample next.

In a comparison between Spearmint, SMAC and TPE, [25] found that Spearmint performs best for low-dimensional continuous problems, while SMAC and TPE performed better for high-dimensional problems.

Other methods

Genetic Algorithms have been used for tuning hyperparameters for many machine learning algorithms. Among them, most work is focused on using it for neural networks [68, 44, 46] and support vector machines [62, 29, 47].

A bandit-based approach has also been explored with Hyperband [45]. Where SMBO methods iteratively suggest new hyperparameter settings to evaluate, Hyperband considers a predefined selection of hyperparameter settings. However, Hyperband does not evaluate all these configurations with an equal amount of resources. Initially, Hyperband evaluates all hyperparameter configurations with limited resources. For example, it can cut off training after a short timespan, or only using a small subset of the data. Based on these results, Hyperband adapts the resource allocation, so that more resources will be used for hyperparameter configurations that are likely to be good. In a specific selection of datasets, for some learning problems, Hyperband improved over Random Search, SMAC and Tree-structured Parzen Estimators.

2.3.2 Algorithm Selection and Hyperparameter Optimization

Starting with Auto-WEKA, several Combined Algorithm Selection and Hyperparameter optimization (CASH) solutions for machine learning were constructed. For many CASH solutions, hyperparameter optimization techniques are extended to include the choice of algorithm and preprocessing steps. This is done by including them as additional parameters to the optimization problem, thus extending the hyperparameter naturally [67, 16, 26].

Auto-WEKA uses the algorithms available in WEKA, a machine learning package for Java [33]. It will perform algorithm selection and hyperparameter optimization for any given dataset, and it also can automatically decide to use one of a few feature selectors and evaluators available in the WEKA package. During the optimization method, internal models are constructed to find promising algorithm and hyperparameter combinations. Auto-WEKA used two different SMBO techniques, SMAC and TPEs, to optimize the algorithm and hyperparameter configurations. The authors showed that Auto-WEKA often outperformed random search.

Hyperopt is a python library that can be used for any SMBO problem[8]. It has been designed to have the SMBO algorithm as an interchangeable component, currently supporting simulated annealing, TPE and Random Search[7]. It had initially been developed for optimizing deep neural networks and convolutional neural networks. Hyperopt-Sklearn uses the Hyperopt library for scikit-learn algorithms [16]. It allows for the construction of machine learning pipelines with any of six preprocessing algorithms and seven classification algorithms, by introducing a parameter for

each method which indicates whether or not it is used in the pipeline.

Auto-sklearn[26] is another solution to the CASH problem. It is a Python reimplementation of Auto-WEKA, but is based on scikit-learn[55]. Moreover, it explores two new additions. The first is to use meta-learning to warm start their Bayesian optimization. In particular, it used a dataset containing meta-features of other datasets, alongside the best ML framework found for each respective dataset in an earlier experiment. Given a new dataset, its meta-features would be calculated, and ML frameworks of the top 25 most similar datasets (according to L_1 distance) would be used to start the Bayesian optimization procedure. The second addition is to construct an ensemble of classifiers based on classifiers that had already been evaluated. When compared to vanilla auto-sklearn, a version of auto-sklearn without the improvements, both of the improvements were found to improve the performance.

Vanilla auto-sklearn was compared to Auto-WEKA, and was found to be significantly better on 6 datasets, worse on 3, and tied on 12. The authors of auto-sklearn compared Hyperopt-Sklearn to vanilla auto-sklearn. They found Hyperopt-Sklearn to be more of a proof-of-concept, as they encountered crashes when sparse data and missing values were presented [26]. Still, out of the sixteen datasets which were fully evaluated, Hyperopt-Sklearn only performed significantly better on one, and scored similarly on nine. No comparison between Hyperopt-Sklearn and auto-sklearn with improvements was provided.

Another solution to the CASH problem is TPOT, a Tree-based Pipeline Optimization Tool [54]. It is different from the above methods in that it does not use SMBO, but instead uses an evolutionary algorithm to optimize machine learning pipelines. A more in-depth look at TPOT is given in the next section.

A different AutoML tool is RECIPE (REsilient ClassifIcation Pipeline Evolution) [21]. Like TPOT, it is also based on genetic programming. When RECIPE was developed, it distinguished itself mainly from TPOT by including the use of a grammar. While TPOT could create invalid pipelines - pipelines which did not have a classifier, or have invalid hyperparameter configurations - RECIPE's grammar would make sure that invalid pipelines could not be constructed. This saved considerable resources, as the number of invalid pipelines in TPOT could be as high as 30% [21].

The second way in which RECIPE differs is by it's optimization goal. While TPOT tried to keep the number of components in the pipeline small while optimizing performance, RECIPE's only focus was the performance of the pipeline. In its evaluation RECIPE was compared to both auto-sklearn and TPOT. RECIPE did not have a significantly different performance than either method in most cases. In the cases that it did, RECIPE was significantly better [21].

However, since the release of RECIPE, TPOT has continued development and now includes a grammar. Because of this, TPOT is no longer able to construct invalid pipelines. Unlike RECIPE, only some of TPOTs grammar is configurable. To the best of my knowledge, no recent evaluation has been performed to see how this impacted the relative performance.

Automated algorithm selection is done in the context of data streams in [69]. In that paper, smaller fixed size windows of the data streams are characterized by meta-features, after which various algorithms are evaluated. A meta-learner is then trained on this data to perform algorithm selection. This proved to be competitive with state-of-the-art ensembles.

An AutoML competition has been organized by ChaLearn, and completed over the course of 19 months [32]. In this competition, participating teams would submit AutoML solutions which would be tested on various datasets containing classification problems. A team lead by Frank Hutter won the competition with auto-sklearn, though they were head to head with a proprietary solution of a team from Intel. The solution of Intel was based on gradient boosting of trees, with no preprocessing except for feature selection.

2.4 Structure of TPOT

TPOT is an AutoML solution using genetic programming, an evolutionary optimization approach to optimize computer programs, or machine learning pipelines in the case of TPOT. This section will first give an introduction to genetic algorithms and genetic programming, and then address how genetic programming is used specifically within TPOT.

2.4.1 Genetic Algorithms for Evolutionary Optimization

Genetic Algorithms are biologically inspired algorithms for optimization [42]. They mimic the biological process of natural selection, where organisms which are better able to survive and procreate, due to differences in DNA, are then also more likely to propagate this DNA to a next generation. This search through a big space of possibilities to find the best solution to an optimization problem, such as DNA sequences which lead to procreation, is used as inspiration to solve analogous complex optimization problems.

DNA is made up of building blocks called genes, which determine the behavior of the DNA, for example having an effect on the appearance of the organism. There are multiple ways in which new DNA structures can be introduced to the population. Whenever an animal organism has offspring, the DNA of the offspring consists of a mix of the DNA of both parents. Additionally, DNA can change over time due to mutation, which causes changes in one or more genes. If this new DNA introduces or improves a trait beneficial to survival and procreation, it is more likely that it will be passed on to the next generation, because it is more likely that the individual having that DNA is able to procreate. In this way biological evolution finds solutions, in the form of new DNA sequences, to the optimization problem of finding the best way to ensure offspring. Genetic algorithms emulate this biological process.

In genetic algorithms, the aim is to find an optimal solution to a defined function. In the context of genetic algorithms, a candidate solution is called an *individual*, the function that expresses how good a candidate solution is, is called a *fitness function*. The first task is then to find a representation for such an individual. The representation generally has clear individual components, much like genes in DNA. This naturally gives rise to *mutation* and *crossover* operators, which can replace a gene or switch it with a gene from a different candidate solution, respectively. Different variations of mutation and crossover operators are possible, [66] differentiates 9 types of mutation operators, and [65] differentiates 8 types of crossover operators. Examples for mutation include changing a gene's value, or removing it completely. Examples of crossover include swapping genes between two or more parents. Finally, to decide which individuals of a generation can propagate to the next, a *selection* procedure needs to be chosen, of which there are again many variations [12]. A common one is proportional selection, where the probability of an individual creating offspring for the next generation is proportional to its fitness.Tournament selection holds tournaments between s individuals, where s is the tournament size, determines the winner based on who has the best fitness, and places that in the next generation [49]. For both of these techniques, neither guarantee that the best individual(s) are selected. Because of this, some schemes deterministically put the best individual(s) in the next generation. This is also called elitism.

Once the representation of individuals is defined, as well as the mutation, crossover and selection functions, the algorithm executes the following steps:

1. Initialize a group of candidate solutions. This can either be done manually or randomly.

2. Evaluate the candidate solutions, determining their fitness.

3. Perform selection, to determine which individuals in the population will proceed to the next generation.

4. Perform crossover between individuals that are in the new generation. Not each individual in the generation needs to be affected by this step.

5. Perform mutation on individuals in the generation. Not each individual in the generation needs to be affected by this step.

6. Repeat steps 2-5 until a specified termination condition is reached. These can include a time limit being met, a set amount of generations performed, a solution of desired quality found or not enough improvement is found for a number of generations.

An example of this procedure is given in Figure 2.5. In this example we aim to minimize the objective function in Equation 2.1 which takes two input variables, which makes it our fitness function. We represent these input variables with seperate genes, and thus define our individual to be a pair of two genes.

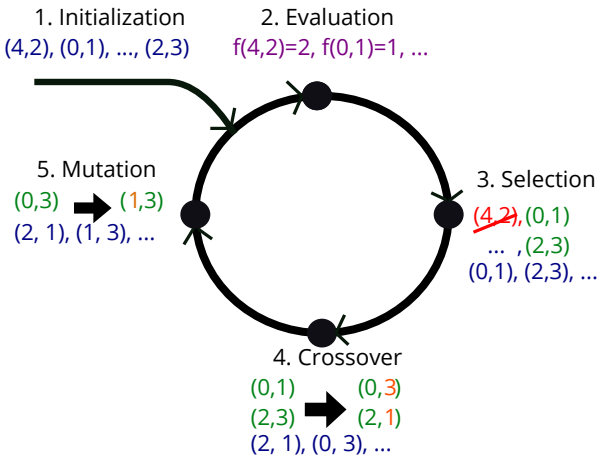


Figure 2.5: An overview of the steps of a genetic algorithm as applied to a simple example.

$$f(a,b) = a^2 - 4 \cdot a + b \quad \text{such that } a, b \in [0, 10] \tag{2.1}$$

The initial generated population is chosen to contain individuals $(4,2)$, $(0,1)$ and $(2,3)$ alongside others. First, all individuals are assigned a fitness score based on the fitness function. The two named individuals have a respective fitness score of $f(4,2) = 2$,$f(0,1) = 1$ and $f(2,3) = -1$. Choosing proportional selection, let's assume that the two best individuals of the three, $(0,1)$ and $(2,3)$, move on to the next generation as that is most likely. Then crossover happens between individuals. For example, switching out the second gene in the two individuals would result in $(0,3)$ and $(2,1)$ as new individuals. Finally, mutation happens, altering genes of some individuals, for example altering the first gene of the individual $(0,3)$ to $(1,3)$, but not altering individual $(2,1)$. The new generation is now final, ready to be evaluated again. This way genetic algorithms iteratively search the solution space, and converge to a (local) optimum.

In practical terms however, genetic algorithms struggle with *premature convergence*. Premature convergence occurs when the objective function has local optima. When the population has converged to a local optimum, it is unlikely for it to converge to the global optimum, as the only way for the population to move away from the local optimum is if crossover or mutation creates a change which results in the new individual having a fitness that is competitive to the local optimum. A technique that is often used is to add a termination condition if there is not

enough improvement in the population for a set amount of generations, as this likely means a (local) optimum has been found, and to consequently restart the genetic algorithm with a new first generation [30].

The fact that the fitness function, design of an individual, mutation, crossover and selection can all be adapted to the problem makes genetic algorithms very flexible. Moreover, it is able to optimize functions which are ill-conditioned or where no derivatives are available [42]. Genetic algorithms lend themselves to effective use of parallelism [50], for example by parallelizing the evaluation procedure.

Not every single problem is easily expressible by optimizing a single objective. As [41] states, in many real-life problems, objectives under consideration conflict with each other, and optimizing a particular solution with respect to a single objective can result in unacceptable results with respect to the other objectives.

For example, when constructing machine learning pipelines, it might be easier to get better model performance by including many steps in the pipeline. However, a secondary objective might be to keep the number of steps in the pipelines down. Limiting the number of steps in the pipeline benefits the interpretabilty as well as lowering resources required to execute it.

Many versions of multi-objective genetic algorithms have been developed, each with their own advantages and disadvantages. In [41], an overview is given for 15 different multi-objective algorithms, each of them well-known and thoroughly tested.

2.4.2 Genetic Programming and TPOT

Genetic programming is the adaptation of genetic algorithms to evolve computer programs. To make this possible, individuals are represented as tree-like structures, where each internal node in the tree is a *primitive* (also called *function*), and each leaf is a *terminal* [58]. Primitives, or functions, represent functions which take input in the form of variables, constants or output of other functions, such as "$f(a, b) = a \cdot b$" or "$f(a, b) = a$ if b is True else 1" . Terminals are constants or variables of the program, and do not take any input, examples include "3", "True" or "x". The set of all defined terminals and primitives is called the *primitive set*. Primitives and terminals can be typed, so that upon creation or mutation of individuals it is enforced that they result in valid expressions. For example, the terminal "3" can be typed as integer, while the terminal "True" can be typed boolean. The primitive "$f(a, b) = a$ if b is True else 1" is specified to have integer input a and boolean input b, and an integer output.

TPOT uses genetic programming and thus represents machine learning pipelines as a tree-like structure. In the case of TPOT, the primitives are various algorithms, from preprocessing to machine learning algorithms, the terminals are then variables such as (preprocessed) datasets or hyperparameters. An example of such a tree-based pipeline is shown in Figure 2.6 taken from [51]. Terminals which specify the hyperparameters are omitted. At the root of the tree is the machine learning algorithm which is able to do the final prediction to a classification or regression problem. The tree does not have to be balanced, so it can for example also represent a pipeline with two consecutive preprocessing steps and a machine learning algorithm. TPOT is written in Python, making use of the genetic algorithm framework *Distributed Evolutionary Algorithms in Python* (DEAP) [28] to perform the evolutionary optimization. The machine learning and preprocessing algorithm implementations are all from scikit-learn [55], with the exception xgboost [19]. The primitive set, defining all valid algorithms and their hyperparameter configurations, is defined by the user in a configuration file, though TPOT also provides a default configuration.

TPOT uses three types of mutation operators; *insertion*, *replacement* and *shrinking*. Figure 2.7 illustrates an example for each of them. In the top left corner one tree-based machine learning pipeline is shown, which uses the Bernoulli Naive Bayes classifier on MinMax-scaled data. The

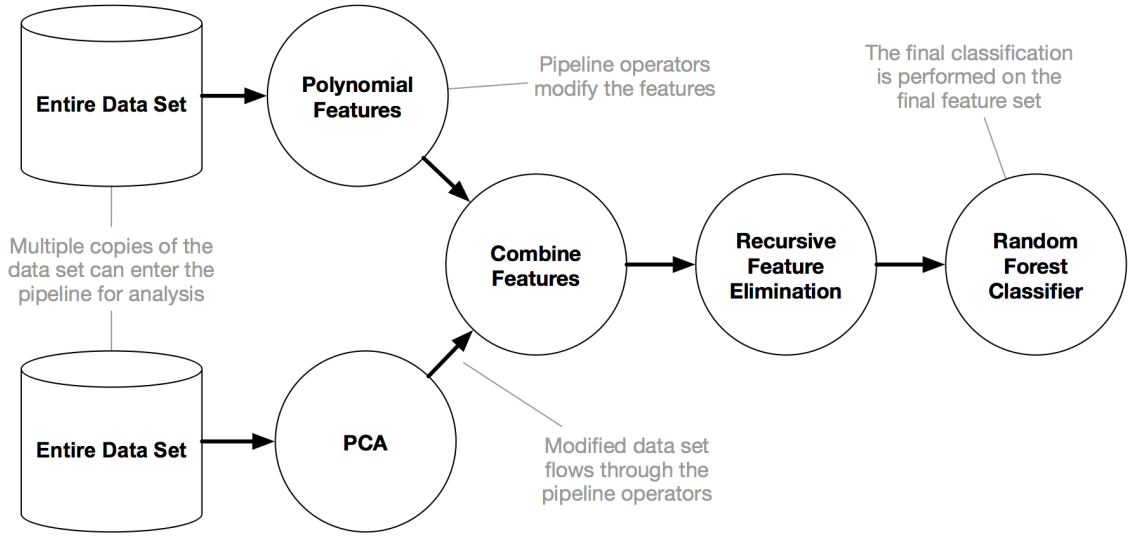


Figure 2.6: An example of a machine learning pipeline generated by TPOT.

hyperparameters 'fit_prior' and 'alpha' for the Bernoulli Naive Bayes classifier are set to 'True' and '1.0', respectively.

The replacement operator chooses a random terminal in the tree, such as a hyperparameter, and replaces it with another valid one. For example, the terminal which specifies the 'fit_prior' value can be changed from 'True' to 'False'.

The insertion operator will insert a new subtree in a random position in the tree, with as a child the subtree that started at the chosen position. For example, the subtree which specifies MinMax-scaling is performed on the terminal 'data' is chosen as the point of insertion. In this example, a subtree consisting of only the 'Binarizer' primitive is inserted, which takes as input the output of the MinMax-scaling subtree.

Finally, the shrinking operator picks a primitive and its subtree, and replaces it with a terminal. For example, again the subtree which specifies MinMax-scaling is performed on the terminal 'data' is chosen. The only primitive in this subtree is the 'MinMax' scaler, so it is removed, as the 'data' terminal is also valid input to the Bernoulli Naive Bayes classifier. This operator can not be chosen if the tree only has one primitive, because this would mean the removal of the only machine learning algorithm in the pipeline.

After selecting an individual for mutation, the choice of which operator is used is uniformly random among available options.

Crossover can take place between any two individuals which can exchange two subtrees with each other and still remain valid pipelines. There is one exception, where both subtrees can not be the single-node 'data' subtree, as the offspring would be identical to its parents. Out of all valid exchanges of subtrees between the two individuals, one is performed, creating two new individuals.

Figure 2.8 gives two examples of crossover. The example on the first row demonstrates crossover where two individuals are combined by taking a subtree of one individual using it to replace a subtree of the other individual. The second example demonstrates crossover where a terminal node of one parent replaces a terminal node of the other parent.

The goal of TPOT is to not only find a pipeline which has good accuracy, but also to keep the total size of the pipeline as small as possible. One reason is the assumption that less-complex pipelines will also generalize better [51]. To achieve this, the selection procedure of the multi-objective genetic algorithm Non-dominated Sorting Genetic Algorithm II (NSGA-II) is used [22]. It finds the Pareto fronts of the trade-off between performance and pipeline length, considering

both the individuals of the current generation as well as the previous one's. Moreover, it also considers a crowding distance for individuals in the Pareto front, so that it will aim to select individuals spaced out along the Pareto front. Based on this selection, the top 20% individuals will have 5 copies each in the next generation [52].

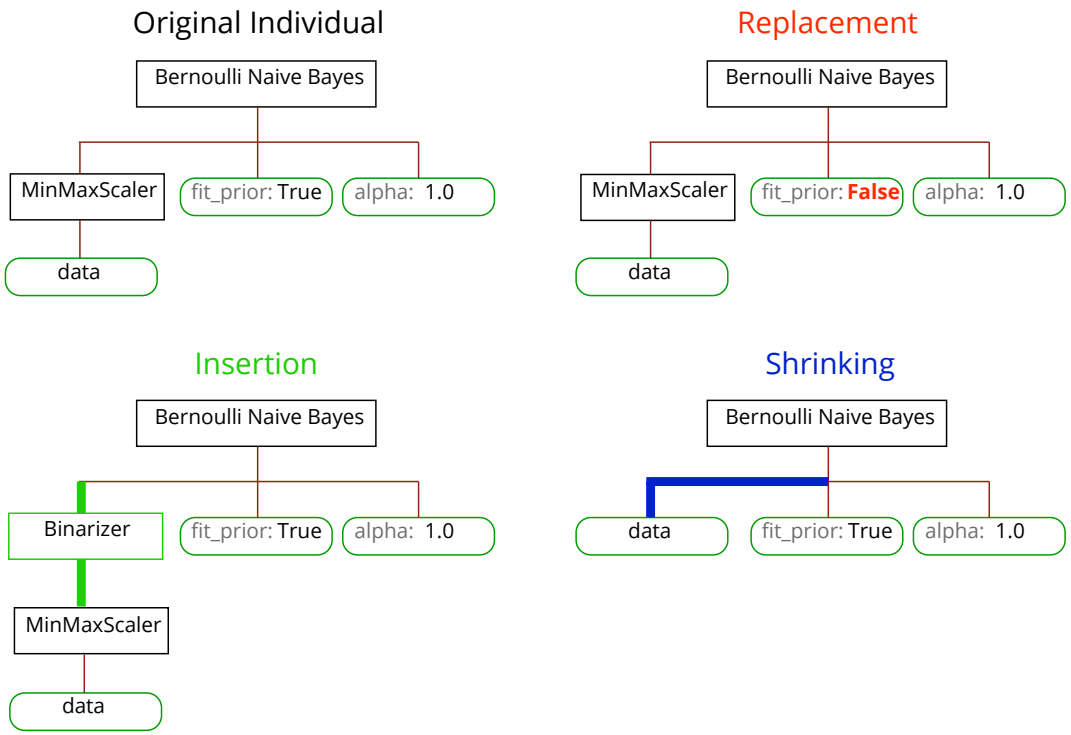


Figure 2.7: Example of mutation operators used in TPOT.

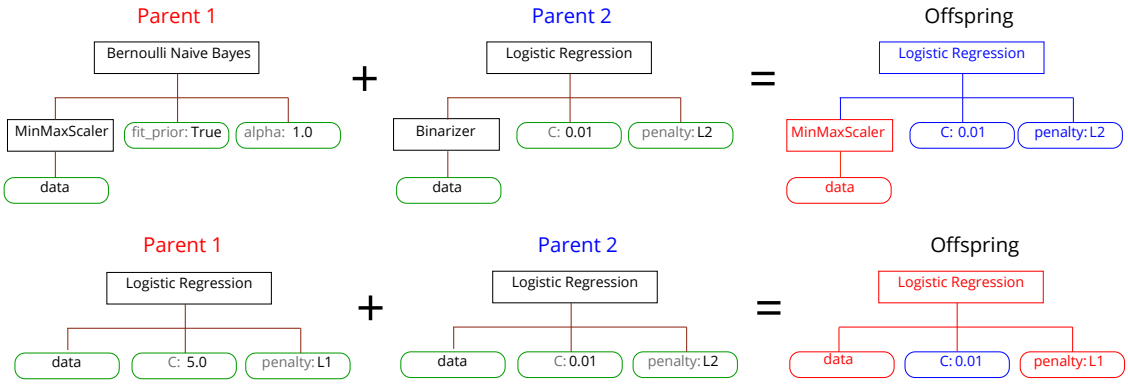


Figure 2.8: Examples of crossover in TPOT.

Chapter 3

Layered TPOT

The work in this chapter was published as part of the AutoML workshop at EMCLPKDD 2017. While I did most of the work, the two co-authors Randy Olson and Joaquin Vanschoren were invaluable to the process. Both contributed in insightful discussions, and Joaquin Vanschoren also helped in shaping the final paper through feedback and editing. Passages are adapted or extended to match the context of the thesis, but most work is taken directly from the paper.

3.1 Concept

This chapter describes the research done to improve TPOT by evaluating pipelines on subsets of the data to discard bad pipelines early on. We introduce a novel improvement of TPOT, aimed at reducing the time needed to evaluate pipelines, without reducing the quality of the final pipeline. Indeed, the most time-consuming part in the optimization process is evaluating the performance of candidate machine learning pipelines. In our modification, this time is reduced by initially evaluating the pipelines on a small subset of the data, and only allowing promising pipelines to be evaluated on the full dataset. In order to do this in a fair manner, modifications to the evolutionary algorithm are implemented to prevent direct comparison between pipelines which are evaluated on different subsets of the data. As such, we aim to find pipelines of similar quality in much less time, making the tool more accessible and practical by requiring less computational time. We call this improvement Layered TPOT (LTPOT).

Age-Layered Population

Individuals that have been trained and evaluated on different sized subsets can not be compared directly, as generally performance improves with the amount of training data. To incorporate a fair evaluation of pipelines on subsets gradually increasing in size, we designed a layered structure to separate competition among individuals. In this, we were inspired by the Age-Layered Population Structure (ALPS) [35], where individuals are segregated into layers to reduce the problem of premature convergence. The problem of premature convergence occurs when the individuals in the population converge to a good local optimum, which means that any new individuals are unlikely to be competitive with this local optimum, and through selection are filtered out of the population before they themselves can converge to a local optimum. In ALPS, all individuals were given an *age* which would increase as an individual or its offspring would remain in the population, and perform breeding and selection only in separate age groups called layers. This segregation is important, because otherwise the old, locally well optimized, individuals would often prevent young individuals from being able to survive the multiple generations they needed to get closer to their local optimum.

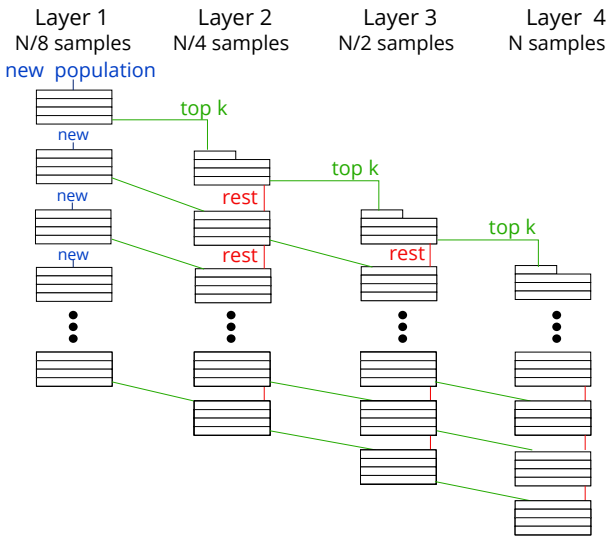


Figure 3.1: A visual overview of the Layered TPOT structure.

3.1.1 Layers in LTPOT

In LTPOT, we wish to evaluate pipelines on subsets of the data. However, the performance of a machine learning pipeline is influenced by the amount of training data it receives. This means that when evaluating individuals on different subsets of the data, their performance cannot directly be compared to one another. Therefore, the individuals are segregated into *layers*.

At each layer, individuals will be evaluated on a subset of different size. The layers are ordered, such that in the first layer the subset used to evaluate the individuals is the smallest, and every layer above that will increase the subset size. When an individual performs well in one layer, it will eventually be transferred to the next. This way, only the best pipelines will be evaluated on the entire dataset in the last layer. A visual overview of the structure and flow of LTPOT is given in Fig.3.1.

Correlation of performance between layers

In the extreme, the selection procedure implicitly assumes that the relative performance of two pipelines is the same when evaluated on a subset of the dataset as it is on the entire dataset. However, this assumption does not always hold.

The learning curves for two pipelines may cross, meaning that one pipeline performs better after being trained on a small subset of the data, while the other performs better when trained on the full dataset. Generally speaking, as the pipelines are evaluated on more data, the relative performance correlates more strongly with the relative performance obtained when they are trained on the entire dataset. This is why our design will evaluate the pipelines on gradually larger subsets of data, so that when a pipeline performs worse than expected as the dataset increases, it need not be evaluated on bigger datasets. Unfortunately, in the case where a pipeline has poor performance on a small subset, but good performance on the entire dataset, LTPOT will not pick up this pipeline.

We set up an experiment to verify whether or not there is indeed a correlation between the performance of a pipeline on a sample of the dataset and its performance on the entire dataset. In this experiment, we evaluated 50 pipelines on 12 datasets with ten times 10-fold cross-validation, with various samples sizes of the dataset as well as the entire dataset. All datasets were part of

dataset	instances	features	$\rho_{\frac{N}{2}}$	$\rho_{\frac{N}{4}}$	$\rho_{\frac{N}{8}}$	$\rho_{\frac{N}{16}}$	$\rho_{\frac{N}{32}}$
satimage	6435	36	0.984	0.903	0.833	0.759	0.641
clean2	6598	168	0.972	0.946	0.890	0.825	0.765
ann-thyroid	7200	21	0.982	0.967	0.946	0.924	0.828
twonorm	7400	20	0.978	0.929	0.898	0.759	0.829
mushroom	8124	22	0.973	0.962	0.935	0.867	0.781
agaricus-lepiota	8145	22	0.989	0.980	0.920	0.877	0.805
coil2000	9822	85	0.939	0.878	0.815	0.568	0.529
pendigits	10992	16	0.984	0.983	0.945	0.844	0.771
nursery	12958	8	0.995	0.995	0.857	0.917	0.917
magic	19020	10	0.988	0.974	0.961	0.947	0.934
letter	20000	16	0.990	0.959	0.918	0.899	0.801
krkopt	28056	6	0.984	0.942	0.917	0.913	0.750

Table 3.1: An overview of the correlation between the ranking of pipelines trained on subsets of the data compared to the entire dataset, $p < 0.0001$ in all cases. The subscript in the column denote the size of the subset.

the Penn Machine Learning Benchmark (PMLB) [53]. The average AUROC of each pipeline for each sample size was determined for each dataset. Sample sizes were $\{N/2^1, \ldots, N/2^5\}$, where N is the number of instances in the dataset. Because the evolutionary algorithm performs pipeline selection based on the ranking of the algorithms, rather than the value of the score metric, we ranked the averaged scores and computed the correlation of the rankings.

In Table 3.1 the Spearman ρ-values are displayed, that signify the correlation between the ranking of pipelines trained on a sample of the dataset, and the ranking of pipelines when trained on the entire dataset. The p-values are omitted because in all cases they are smaller than 0.0001 and all correlations are thus significant. From Table 3.1 we see that that there is a positive correlation between the rankings for all sample sizes and datasets, and the correlation gets stronger as the sample size gets closer to the full dataset size.

We experimented with various curve fitting methods to extrapolate the learning curves of pipelines so that crossing learning curves might be predicted earlier, but they did not improve the results. In future work the use of meta-learning for learning curve extrapolation, such as in [70], will be tried.

3.1.2 Layered TPOT Algorithm

We will now give a more in-depth break down of the algorithm used in LTPOT. Algorithm 1 shows the core of the layered algorithm LayeredEA, and Algorithm 2 gives descriptions of the subroutines called from LayeredEA.

Selecting parameter values

Before calling LayeredEA, the number of layers as well as their *sample size* is defined. The sample size of a layer dictates how many instances are sampled from the dataset, to create the subset that the pipelines in that layer will be evaluated on. The subset is created by stratified uniform random sampling without replacement, and pipelines will be evaluated on this subset with 5-fold cross-validation. In this study, the sample sizes used in each layer are dictated by the size of the dataset. Let the dataset contain N instances, then the final layer will always train on the entire dataset, and each subsequent layer will use half of the data the layer above did. In this study, the number of layers used is 4, for every dataset. The respective sample sizes used at each layer are thus $\frac{N}{8}$, $\frac{N}{4}$, $\frac{N}{2}$ and N. We use the term *higher* layer loosely to denote layers which sample more of the entire dataset (i.e. the layer with sample size $\frac{N}{2}$ is higher than the layer with

Algorithm 1 Layered Evolutionary Algorithm

function LayeredEA(population, S, g, G, D)
population: a set of pipelines that will be the first generation
S: set containing the sample size for each layer
g: interval in generations for when a transfer should occur
G: total number of generations
D: dataset to construct a pipeline for
    $M \leftarrow \|S\|$ ▷ Denote the number of layers.
    $P \leftarrow \|population\|$ ▷ Denote population size.
    $L_1 \leftarrow$ population
    $L_2, \ldots, L_M \leftarrow \emptyset$
    for i in $1..G$ **do**
        for l in $1..M$ **do**
            if $L_l \neq \emptyset$ **and** $i \bmod g < 2^{(M-l+1)}$ **then**
                offspring $\leftarrow$ VarOr(L_l, P)
                Evaluate(offspring, S_l, D)
                $L_l \leftarrow$ Selection($L_l \cup$ offspring, P)
            end if
        end for
        if $i \bmod g = 0$ **then**
            for l in $(M-1)..1$ **do**
                $L_{l+1} \leftarrow L_{l+1} \cup$ Top(L_l, $P/2$)
            end for
            $L_1 \leftarrow$ NewPopulation(P)
        end if
    end for
    return Top(L_M, 1)
end function

Algorithm 2 Functions called in Layered EA

1: **function** VarOr($Population, N$)
Performs mutation and crossover on the individuals in $Population$, creating N new individuals.
end function

2: **function** Evaluate($Population, s, D$)
Evaluates each individual on a subset of dataset D, created by taking s instances by stratified sampling. Individuals are evaluated based on 3-fold CV accuracy and number of components in the pipeline. Results are saved as attributes of individuals.
end function

3: **function** Selection($Population, p$)
Creates pareto-fronts based on the accuracy-pipeline complexity trade-off. Then takes the first p individuals after ordering the population by which Pareto-front they are in (individuals in the first front come first).
end function

4: **function** Top($Population, k$)
Returns the k best individuals of the population, by constructing a pareto-front based on the accuracy-pipeline complexity trade-off.
end function

5: **function** NewPopulation(P)
Creates a new population of P individuals.
end function

sample size $\frac{N}{4}$).

There are two ways to specify for how long the main loop of lines 7 through 20 should run: a set amount of generations G, or an amount of time. We chose not to work with a specific number of generations G, but instead let the main loop in lines 7 through 20 run for eight hours. For parameter g, the amount of generations between transfer, we experimented with values 2 and 16. With a g value of 2, LTPOT acts almost as a filter, allowing only very little optimization in early layers. A value of 16, however, allows many generations of early optimization, before passing individuals through to the next layer.

For the final part of the initialization, a population of size P is generated randomly.

LayeredEA

When calling LayeredEA, as shown in Algorithm 1, the first step is to denote constants based on the input, specifically the number of layers M (line 1) and the size of the population P (line 2), as well assigning the initial population to the first layer (line 4) and marking all other layers as empty (line 5).

Then, the evolutionary algorithm will start iterating through the generations (line 6-20), at each generation again progressing the evolutionary algorithm in each layer (line 7-13) and then if required transferring individuals from one layer to the next (line 14-19).

Progressing the evolutionary algorithm in each layer (line 7-13), only happens for layers which are *active*. This means that there must be a population in the layer (first clause on line 8), which it may not yet have if not enough transfers have taken place yet. Additionally, layers which evaluate on more data are not active every generation (second clause on line 8), this is motivated below.

When progressing the evolutionary algorithm, it executes the same steps as TPOT would. First, a new population is created from the individuals evaluated during the last generation in the same layer, by performing mutation and cross-over (line 9). However, every time a layer is

passed new individuals from a previous layer, as well as in the very first generation, the provided population is taken as is without creating offspring.[1] The new individuals are then evaluated based on the sample of the data as defined by their layer (line 10). Finally, based on the Pareto-front of the trade-off between the performance score of the pipeline as well as the pipeline length, the best individuals are picked among the new individuals as well as last generation's (line 11). Then, every g generations, the best individuals from each layer get passed to the next one. In our configuration we chose to transfer half of the layer's population.
The final pipeline chosen by LTPOT is the pipeline which has the best score in the highest layer (line 21).

Optimizations

There are a few scenario's that either require some additional clarification, or deviate from the above algorithm:

The first time a layer receives individuals from the layer before it, the selection procedure will oversample from this population so that the population in the layer will also grow to size P (line 11). This is done so that in subsequent generations, more variations of the original pipelines will exist, allowing for a better search for optimal pipelines.

Secondly, if LTPOT runs for a specified amount of generations, layers will be turned off whenever their population can no longer reach the highest layer. For example, let LTPOT be configured with 4 layers. When LTPOT is less than $3 * g$ generations away from completion, any individual in the lowest layer will never reach the highest layer, thus rendering any results obtained in this layer useless. Whenever this happens, the respective layer will no longer have their individuals evaluated or transferred to a next layer.

Next, there is the earlier mentioned restriction on activating layers as shown in the second clause on line 8. LTPOT has a selection process in place for which pipelines will be evaluated on the entire dataset. This means that at higher layers, the pipelines in the population are likely already quite good. Because of these two factors, we want to limit the exploration in higher layers. To do this, instead of running the evolutionary algorithm in each layer every generation, higher layers can be turned off for some generations. In this study, for a layered structure with M layers, layer l is progressed for $\min(2^{(M-l+1)}, g)$ generations every g generations, with $l \in \{1, \cdots, M\}$. This is demonstrated with $g = 12$ and $M = 4$ in Fig. 3.2, and checked in the second clause on line 8. We have not yet evaluated if this leads to significant improvements.

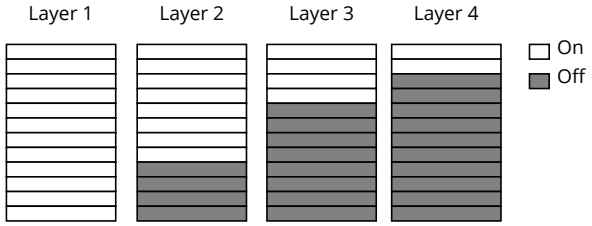


Figure 3.2: An illustration showing that some layers will be 'turned off', meaning that no iterations of the evolutionary algorithm are executed.

Finally, to prevent any one pipeline from halting the algorithm, a pipeline's evaluation is automatically stopped if it exceeds a given time limit. If the evaluation is stopped this way, the

[1] This is not incorporated in the pseudo-code of algorithm 1, to keep the general structure clear.

pipeline is marked as failure and will not be considered as parent for the next generation. This behavior is present in the original TPOT, and adopted to LTPOT by further decreasing the time limit by layer. In the top layer, each individual is allowed the same evaluation time as it would in TPOT. However, for lower layers, the time allowed is decreased quadratically proportional to the sample size (a layer with half the data gets a fourth of the time per individual).

3.2 Empirical evaluation

3.2.1 Experimental Questions

The goal of LTPOT is to find pipelines at least as good as TPOT's, but in less time. This also means that, given the same amount of time, LTPOT could very well find better pipelines. To assess whether or not this is achieved, we will evaluate LTPOT in three ways.

First, we want to evaluate if, given the same amount of time, LTPOT will outperform TPOT when their best found pipelines are compared. To do this, a ranking is constructed between TPOT and LTPOT for each dataset over time, by ranking the performance of the best pipeline found so far at regular time intervals. We omit a comparison to Random Search, as TPOT compared favorably to Random Search in earlier work [52].

Secondly, to quantify how much faster LTPOT is, we compare the time needed for LTPOT to find a pipeline at least as good as the best pipeline found by TPOT. We then compare it to the time TPOT needs to find this pipeline.

Finally, we will also compare the Area Under the Receiver Operating Characteristic curve (AUROC) of the final pipelines found by each method, so we can quantify the difference in model quality between the methods.

3.2.2 Experimental Setup

We compare LTPOT to the original TPOT by evaluating both on a selection of 18 large datasets. We specifically chose larger datasets so that there will be a distinct difference in time needed to evaluate individuals on the entire dataset versus just a subset. All datasets in the selection contain at least one hundred thousand instances, though most contain exactly one million. The selection includes pseudo-artificial datasets described in [71]. The datasets are available for download and inspection on OpenML[2], an open database for machine learning experiments [72]. We previously evaluated LTPOT on a selection of datasets from the Penn Machine Learning Benchmark [53], which TPOT was initially evaluated on. Because those datasets were relatively small, pipeline evaluations were quick even on the full dataset, so there was no significant benefit of using LTPOT. On each dataset, each method is evaluated nine times, starting with a different initial generation each time.

As described earlier, there are many hyperparameters with which to tune LTPOT. In this study, we only experiment with g, the amount of generations between transfer. The choices for g will be 2 and 16, and these configurations will be referred to as LTPOT-2 and LTPOT-16, respectively. This is meant to give insight in the effectiveness of two functions of LTPOT: filtering and early optimization. For LTPOT-2, layers act almost solely as a filter, by passing the best individuals to the next layer every other iteration, not much optimization takes place in lower layers, but it does allow for the early discarding of pipelines which seem to perform poorly. With LTPOT-16, we instead see that a lot of optimization can take place based on results found in lower layers, as relatively more time is spent evaluating and improving individuals in lower layers compared to LTPOT-2. In other words, LTPOT-2's first layer allows for more early exploration, while LTPOT-16's first layer is more focused on exploitation.

[2] https://www.openml.org/s/69

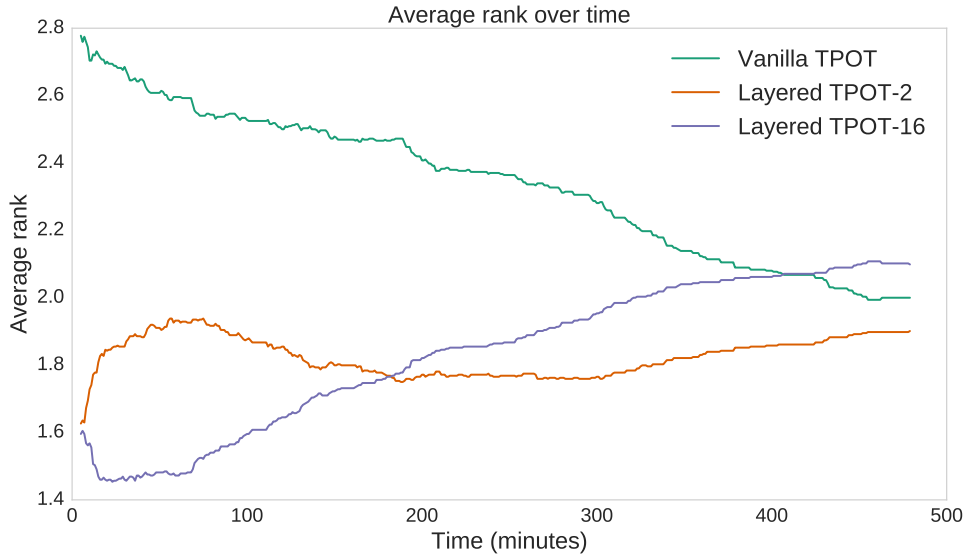


Figure 3.3: Ranking of each method averaged over all datasets based on internal AUC scores of the best individual found so far.

The amount of individuals transferred, k, will be set to 15, which is half of the total population size $P = 30$. Each LTPOT configuration, as well as TPOT, is run nine times per dataset, each time with a different *random seed*, guaranteeing a different initial population and subsequent choices for crossover and mutation. Each run set to last 8 hours, but each individual pipeline may only be evaluated for at most 10 minutes. We explored different values for P and different amounts of evaluation time per individual, while keeping the total run time constant at 8 hours, and found that for the chosen datasets these values strike a balance between having a diverse enough population and being able to evaluate enough generations.

3.2.3 Results

First, we compare the various configurations by their average rank over time, which can be seen in Fig. 3.3. In this figure, for each configuration, for every dataset and seed, the best found pipelines so far are ranked against each other every minute. For each method, the average Friedman rank [23] across all these datasets and seeds is calculated based on the highest AUROC score achieved by the best pipeline so far, using the fixed hyperparameter values stated above. To calculate the rank, we consider the result a tie if the difference in AUROC values is smaller than 0.01. A lower rank is better.

In Fig. 3.3, we see that LTPOT on average achieves the best scores throughout the entire 8 hour period. LTPOT-16 starts by outperforming LTPOT-2 slightly, but as time goes on its relative performance drops, even being surpassed by TPOT. From this, it seems that while LTPOT works well as a filter, optimization in early layers does not pay off beyond the early stages. Thus, a lower value for g is better based on these results. Furthermore, we see that towards the end TPOT is decisively improving over LTPOT-16, but only very slowly over LTPOT-2, as the average rank of LTPOT-2 increases only very slowly.

A different rank does not necessarily mean the found model performance differs a lot. To clarify this, we look at the difference in AUROC by dataset per configuration, as seen in Fig. 3.4 and Fig. 3.5.

In Fig. 3.4 and 3.5 we show a boxplot that describes the distribution of AUROC scores of the

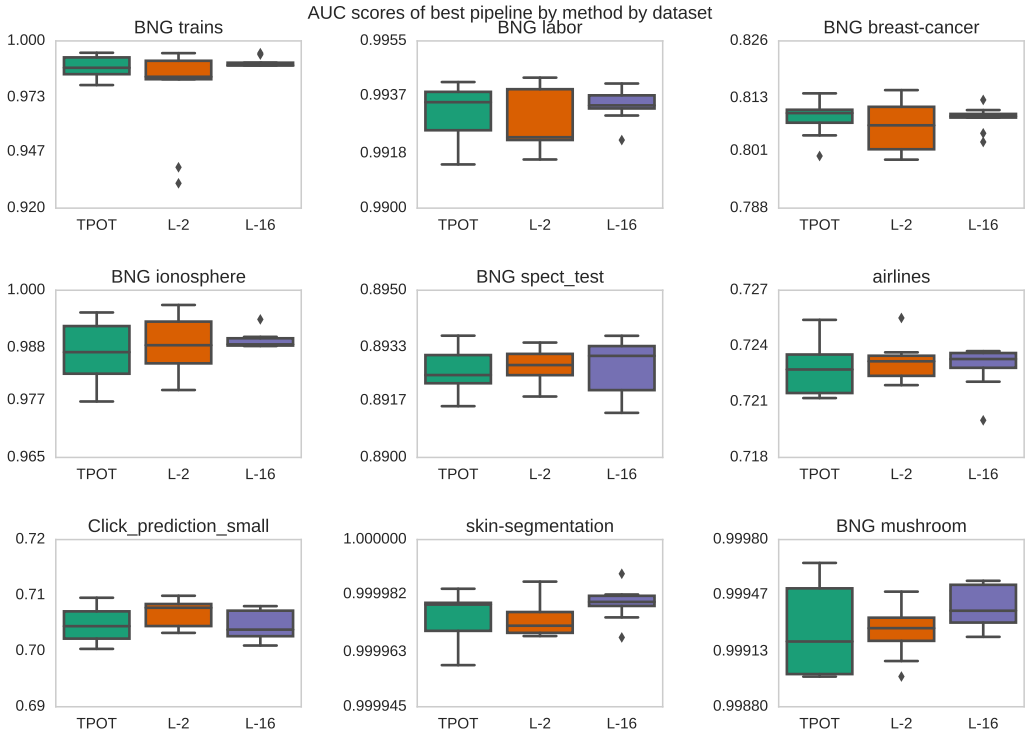


Figure 3.4: An overview of achieved AUROC score for each run of each method by dataset.

final pipelines by each method. No single method is dominant over the others, and differences in AUROC scores are small for almost all datasets. Using a student t-test, we determined that there is no statistically significant difference ($p < 0.05$) between the final pipelines (after the full 8 hour time budget).

However, looking at the average ranking by configuration in Fig. 3.3, we see that under smaller time budgets LTPOT-2 often finds pipelines which are better than TPOT. In this scenario, it is interesting to see how much time LTPOT needs to find a pipeline at least as good as the best pipeline TPOT found. We compared LTPOT-2 to TPOT for each dataset and seed, and looked at how long it took for the method which found the best pipeline to find a pipeline at least as good as the best found pipeline by the other method. Figure 3.6 shows the time difference (in minutes) between finding these equally good pipelines. Positive values indicate that LTPOT is faster. Yellow distributions correspond to seeds where LTPOT eventually found the best pipeline, and show how much sooner LTPOT found a pipeline at least as good as the best pipeline of TPOT. Blue distributions correspond to cases where TPOT eventually found the best pipeline, and show how much later TPOT found a pipeline at least as good as LTPOT's best.

In general, when LTPOT finds the best pipeline, it finds a pipeline at least as good as TPOT's best pipeline much sooner. In particular for LTPOT-16 we see that it often is at least 200 minutes faster. Even in the cases where TPOT eventually finds the best pipelines, we see that it often finds a pipeline at least as good as LTPOT's best only after LTPOT already found it. This again is especially true for LTPOT-16, where almost all yellow distributions are entirely positive.

However, even when one method finds a pipeline at least as good as the other method's eventual best pipeline, it can still be the case that the eventual worst method at that same time already has quite a good pipeline. Therefore, we compare the performance of the best pipeline of each method found at time t, where time t is the time where the best method finds a pipeline at least as good as the eventual best pipeline of the worst method. Between 18 datasets and 9 seeds, there

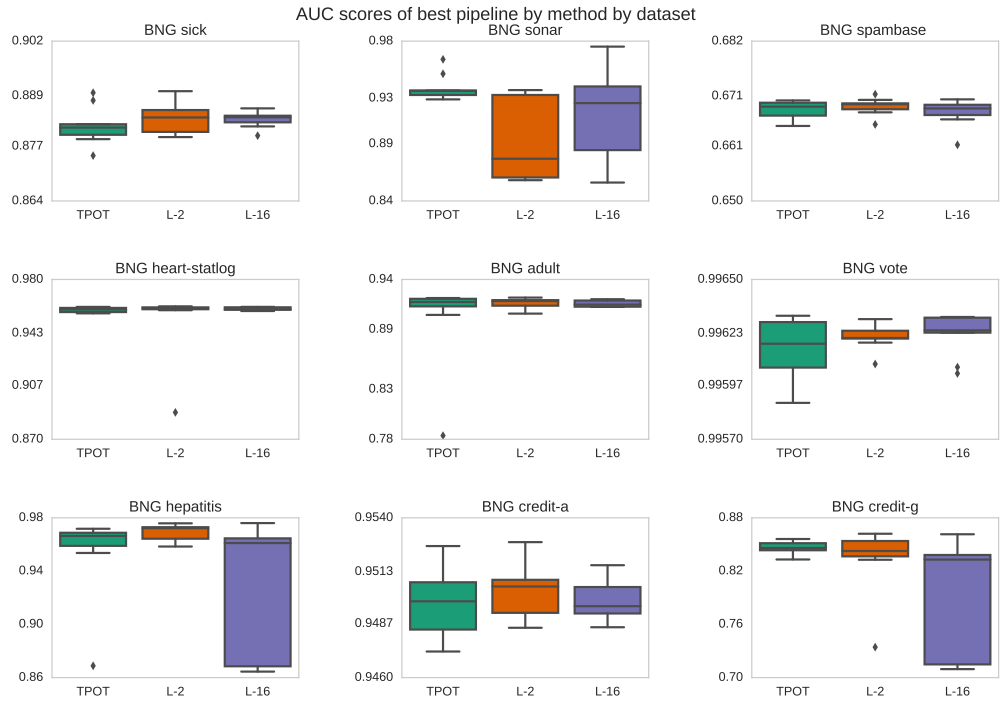


Figure 3.5: An overview of achieved AUROC score for each run of each method by dataset.

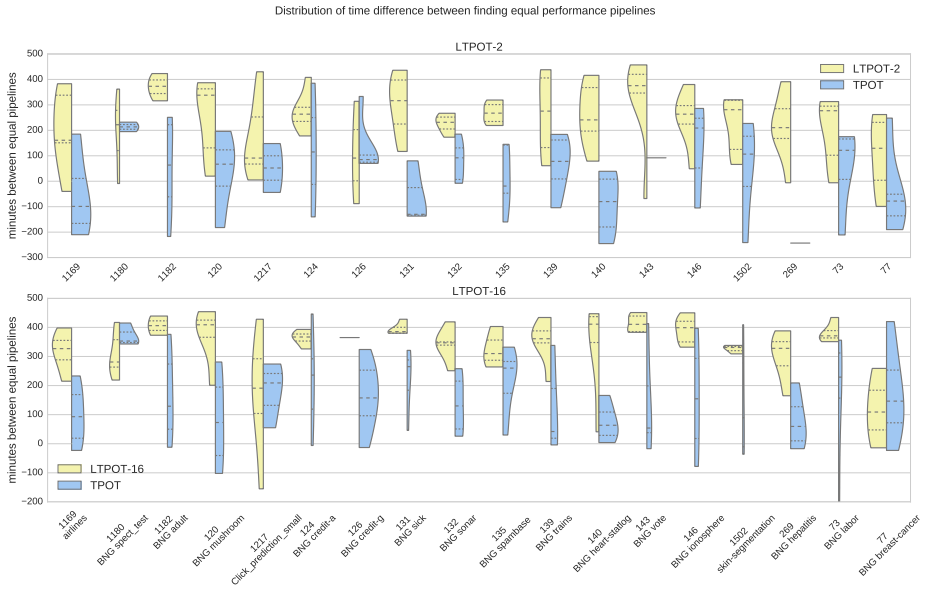


Figure 3.6: Violin plots of the time difference between finding two equally good pipelines. Datasets are shown together with their OpenML ID number.

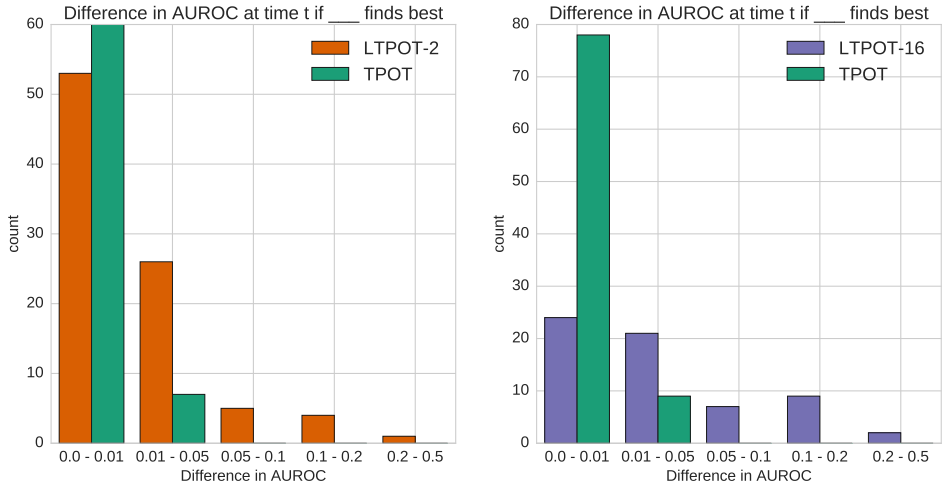


Figure 3.7: Shows the AUROC difference at time t, which is the time the best method (color coded) finds a pipeline at least as good as the other method will find.

are 162 comparisons between TPOT and LTPOT-2 or LTPOT-16. The comparison in AUROC at time t is shown in Figure 3.7.

We see that when TPOT finds a pipeline at least as good as LTPOT's best, in most cases it does so when LTPOT already has found a pipeline at most 0.01 AUROC worse. However, when LTPOT finds a pipeline at least as good as TPOT's best, TPOT has relatively worse pipelines more often, and in some cases as much as over 0.2 AUROC worse.

From this we conclude that in many cases LTPOT finds good pipelines faster. While LTPOT-2 does not always find the best pipeline, when it doesn't, it finds comparable pipelines at least as quickly as TPOT. LTPOT-16 finds comparable pipelines even quicker, although it becomes less competitive under larger time budgets.

Chapter 4

Initializing TPOT with meta-learning

Meta-learning has been an important approach to automating algorithm selection and hyperparameter configuration. Using meta-learning to warm-start the Bayesian optimization in auto-sklearn improved its performance [9]. In the context of genetic algorithms, meta-learning has also been applied for hyperparameter optimization [60]. This chapter explores three meta-learners for suggesting good individuals for the first generation of TPOT optimizations.

4.1 Method

Three meta-learners are evaluated for the use of suggesting good individuals for the first generation. The first method is to train a Random Forest Regressor on the meta-dataset, in order to create a model which can predict the accuracy for each algorithm-hyperparameter configuration. The second and third methods are adaptations the method of [60] for TPOT, which suggests individuals based on nearest neighbors.

4.1.1 Meta-dataset

A meta-dataset is created for the OpenML-100 datasets[1]. Metadata features for each dataset could be retrieved directly from OpenML. This gave a set of 118 metafeatures, which includes features from each group discussed in Section 2.2. We ran experiments to determine the 10-fold crossvalidation accuracy of several algorithm configurations for each of these datasets. All hyperparameter combinations for each baselearner considered can be found in Table 4.1, and form a subset of the algorithms and hyperparameter combinations considered by default in TPOT.

The selection is very limited compared to the full configuration space that is used by default in TPOT. This is scope was picked due to resource and time constraints.

4.1.2 Meta-learners

Two different types of meta-learners are used, a Random Forest Regressor, and the k-NN approach discussed in [60], which is adapted in two ways.

k-nearest neighbor ranking

We will first explain the approach used in [60], and then describe how it is adapted for TPOT. In [60], meta-learning was used to determine good starting points for the genetic algorithm used to perform hyperparameter configuration for two learners, the Support Vector Machine and Random

[1] www.openml.org/s/14

learner	hyperparameter	values
GaussianNB	-	-
MultinomialNB	fit_prior	False, True
	alpha	$\{0.001, 0.01, 0.1, 1, 10, 100\}$
BinomialNB	fit_prior	False, True
	alpha	$\{0.001, 0.01, 0.1, 1, 10, 100\}$
Nearest Neighbor	k	$\{1, 3, \cdots, 99\}$
	voting weight	uniform, distance
	Minkowski p	1, 2
Logistic Regression	C	$\{0.001, 0.01, 0.1, 0.5, 1, 5, 10, 15, 20, 25\}$
	dual	True, False
	penalty	l1, l2
Decision Tree	criterion	gini, entropy
	max_depth	$\{1, 3, 5, 7, 9, 11\}$
	min_samples_split	$\{1, 5, 9, 13, 17\}$
	min_samples_leaf	$\{1, 5, 9, 13, 17\}$

Table 4.1: The baselearners and their considered hyperparameter values.

Forest classifiers.

The meta-dataset used consisted of 15 meta-features describing 102 datasets, and results of a gridsearch over a discretized hyperparameter space for each learner. To create a population of n individuals, the n nearest neighbors are looked up, and the hyperparameter configuration that worked best for them are recommended.

However, they would not directly perform nearest neighbor in the space defined by the 15 meta-features. As the authors point out, the number of features is relatively high for the number of samples, and the nearest neighbor algorithm is sensitive to irrelevant features. Instead, first a feature selection technique is performed, and the nearest neighbors are determined in the space defined by those features.

The optimal feature set for each learner was determined separately, as [40] found that a set of meta-features suitable for predicting the performance of one learner may not be suitable for predicting another learner. The goal of the feature selection is to minimize the sum of distances between optimal hyperparameter values of the nearest neighbors and the dataset itself. The pseudo code for the feature-selection technique can be found in Algorithm 3.

Algorithm 3 Meta-Feature Selection

function META_FEATURE_SELECTION(X, H, k)
X: set of all datasets with their meta-features
H: set of all feature selection functions, one for each possible subset
k: the number of nearest neighbors to find
  for feature selection functions $h \in H$ **do**
    $d_h \leftarrow 0$
    for datasets $x \in X$ **do**
      $N \leftarrow k$-nearest datasets of x in the space of the meta-features selected by h.
      $d_\theta \leftarrow$sum of distances between optimal parameter values of x and N
      $d_h \leftarrow d_h + d_\theta$
    end for
  end for
  return h that minimizes d_h
end function

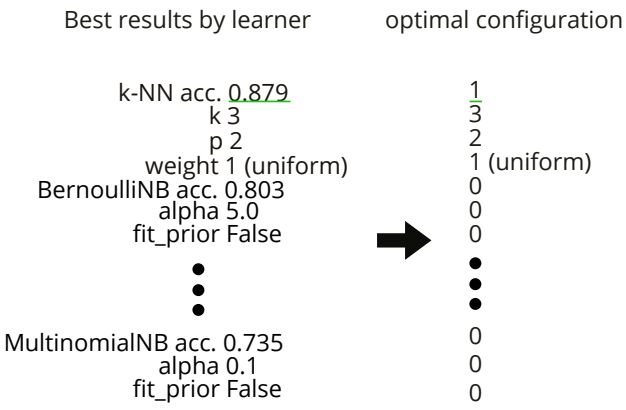


Figure 4.1: Encoding benchmarks results to a single configuration vector.

This technique is not directly applicable to our meta-dataset for TPOT for two reasons. First, we are not aiming to optimize the hyperparameters for a given classifier, instead TPOT aims to determine the best pipeline. This means, at the very least, the classifier needs to be recommended in addition to the best hyperparameters for it. Second, because there are 128 meta-features available, considering every possible subset of them is not feasible.

To make the search of a meta-feature space possible, we first reduce the meta-feature space by applying Principle Component Analysis (PCA). PCA is a dimensionality reduction technique which reduces the number of dimensions in the data while retaining as much as possible of the variation present in the data [39]. This is done by transforming to the Principle Components, which are uncorrelated and ordered by how much variance is retained. We reduce the space to only the first 15 principle components.

This allows the search of a subset of principle components by Algorithm 3 in reasonable time.

Because we need to be able to recommend the base learner in addition to its hyperparameter configurations, the following two adaptations are explored.

The first way is similar to [60], and one feature selection per base learner is learned. This results in one meta-model per learner. Learning a separate subset of features for each learner means that nearest neighbor distance can not be used directly across models to construct a ranking.

To construct a recommendation for 10 individuals, each model determines the accuracy of its nearest neighbor, and the neighbor with the highest accuracy is added as a recommendation. This step repeats itself, each model only considering the nearest neighbor that is not yet included in the recommendation. We will refer to this method as 'NN-CLF'.

The second way to adapt the nearest neighbor recommendations to the TPOT setting is to instead construct one hyperparameter vector per dataset. The vector consists of all hyperparameters of all base learners in addition to an indicator column for each base learner. The values of this vector are then determined by the configuration that achieved the best performance for the given dataset. The vector indicates, for a given dataset, which algorithm achieved the best performance, and the hyperparameters with which it was achieved. An example is given in Figure 4.1.

We then use Algorithm 3 again to find the feature space for which determining the nearest neighbors gives hyperparameter configurations closest to the true best configuration for each dataset. Thus, we find one space in which recommendations can be given for any learner and hyperparameter setting. We will refer to this method as 'NN-ALL'.

Random Forest Regressor Ranking

The second method to construct a set of recommended pipelines is to use Random Forest Regressors to predict the accuracy of each algorithm-hyperparameter configuration. To do this, one Random Forest model was trained on a per-learner basis, on all meta-data for that specific learner. Then, for each algorithm-hyperparameter combination that was present in the benchmark, the accuracy is predicted. The ten configurations with the highest predicted accuracy will form the initial generation. We will refer to this method as 'RFR'.

A model was also learned from the dataset which combined the results of all learners, as the k-NN approach. However, this leads to a model which only predicted the same 10 configurations for all datasets in most cases. This model was left out of evaluation due to giving too little dataset-specific recommendations.

Baselines

We compare these three of warm-starting methods to three baselines:

- Rand-base. This method suggests 10 configurations at random, but, like the meta-learners, does not consider preprocessing steps in the first generation. This should allow us to determine whether the meta-learner is able to learn a model which is useful.
- Rand-pipe. This method suggests 10 configurations at random, but, unlike the meta-learners, does consider including preprocessing steps in the first generation. This should help determine whether recommending just base-learners can be competitive with pipelines including preprocessing from the first generation.
- TPOT. A normal unmodified version of TPOT with the default configuration, meaning it considers a wider array of learners and hyperparameter configurations than the above three methods. This baseline is included to put the results into context with the current state of TPOT.

Additionally, we compare the warm-starting methods to an 'Oracle'. The Oracle will start with the 10 true best configurations found in the benchmark for the dataset. Comparison to the oracle should help determine how much room for improvement there is in the way the meta-model is learned.

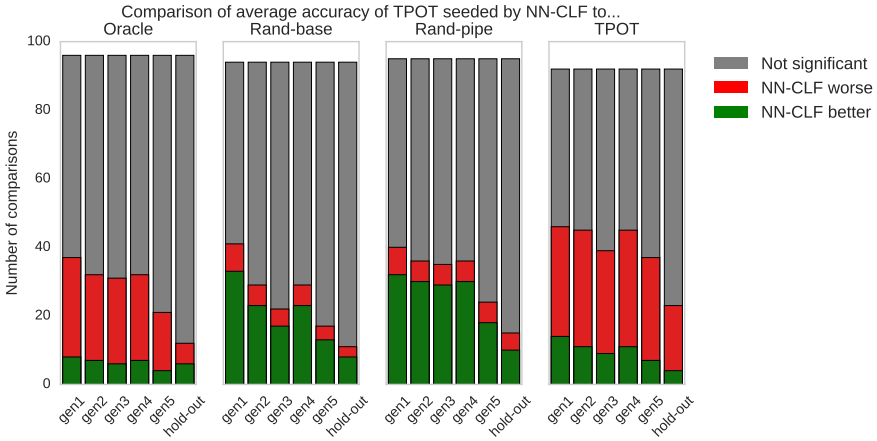
4.2 Results

We compare the proposed methods based on the performance of the pipeline which achieved the highest accuracy. Each method ran for 5 generations with a population size of 10.

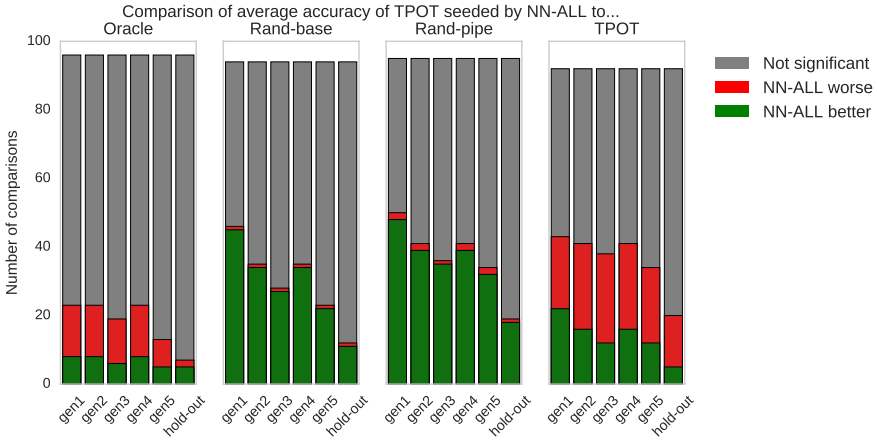
Because the meta-learner only has a direct effect on the initial generation, we want to monitor the effect of the recommendations over time. To do this, we compare the accuracy of the best pipeline so far at each generation after the first, in addition to the hold-out accuracy achieved by the best pipeline after 5 generations. The accuracy of the best pipeline in each generation is determined based on the internal score assigned by TPOT, which is determined by 5-fold cross-validation on the training data. Finally, the generalization performance of the best pipeline found after five generations is evaluated using hold-out with 75% training data and 25% test data.

To account for the random behavior of the genetic algorithm, as well as the effect of the initial random population of some methods, we repeat each experiment 10 times.

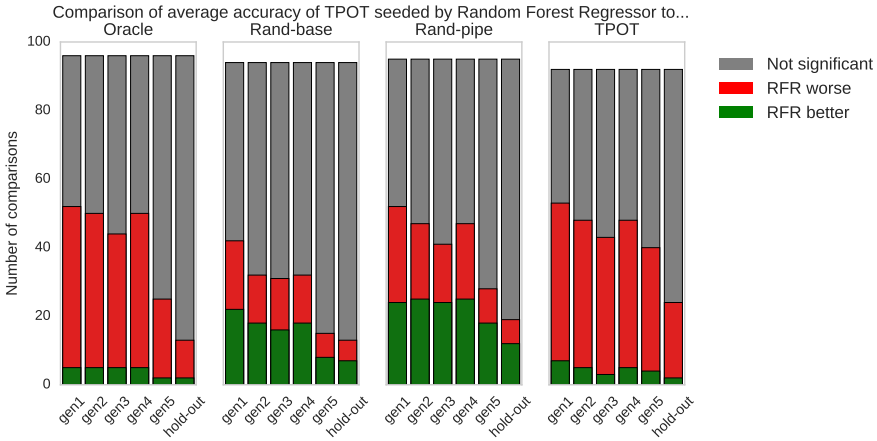
The results can be seen in Figure 4.2. In each subfigure, one meta-model is compared to the baselines as well as the Oracle. Per comparison, you see the amount of times that the performance for a given dataset was significantly worse, better, or not significantly different. A change was considered significantly different if it rejects the student t-test null hypothesis with $p < 0.05$. From



(a) Comparison of NN-CLF to the baselines and Oracle.



(b) Comparison of NN-ALL to the baselines and Oracle.



(c) Comparison of RFR to the baselines and Oracle.

Figure 4.2: Result comparison of various approaches to train a meta-model for population suggestions.

the results we see that all methods have no significant improvement for the majority of datasets. However, there seems to be a clear distinction between the NN and RFR approach, in that RFR seems to have bad suggestions much more often than NN.

Overall, we see that NN-ALL seems to create the best meta-model. While it does not outperform either TPOT or Oracle, it does perform better than both Rand-base and Rand-pipe in many cases. While there are scenarios where its recommendations lead to worse performance, there are many more cases where performance is significantly improved by the recommendations.

However, using the default configuration space of TPOT with more classifiers and preprocessing steps still leads to better pipelines, on average. Even when only considering the limited configuration space, the comparison to Oracle suggests it might be possible to further improve on the way to construct a meta-model. Still, it should be noted that in most cases, there is no significant difference between NN-ALL and Oracle.

While according to [40], different features are important for different classifiers, learning a single representation was still better. This can be explained by the way the predictions of the classifier-specific meta-models were used to generate a recommendation.

The decision to then recommend based on achieved accuracy might lead to recommending classifiers which perform well generally more than data specific ones.

We observe that in almost all cases, the number of significant differences declines by generation. This is likely because changes over time reduce the effect of the initial population. For example, TPOT is able to find better pipelines than originally proposed, by tuning the hyperparametersettings and preprocessing steps further.

We see that in all cases, there are fewer significant differences in the hold-out accuracy compared to even internal test accuracy in the 5^{th} generation. However, the best pipeline of the 5^{th} generation is the best pipeline found by TPOT, and thus also the one tested with hold-out. Thus, the difference in significant differences is not explained by the pipeline evaluated.

The difference is likely caused by the fact that the accuracy within the generations is determined based on a 5-fold cross-validation test. Compared to the hold-out accuracy, this should have smaller variance, as it already is the average of five splits. Because of this smaller variance, it should be easier to determine if there is a significant difference in the means of the best accuracies obtained. If the generalization performance of the best pipeline were to be evaluated with 10-fold cross-validation, the results might have been more similar to that seen in the 5^{th} generation.

In order to have an additional measure of impact of the initial suggestions, we also look at the amount of times one of the initial suggestions is present in the final best pipeline. In some cases, it can be that one of the suggested pipelines is exactly the best pipeline found after 5 generations. In others, the suggested pipeline is extended with one or more preprocessing steps.

Table 4.2, the degree to which initial suggestions are present in the final best pipeline are shown. The column 'exact' displays the fraction of times the final best pipeline is exactly the same as one of the initial suggestions. The column 'preprocessing' displays the fraction of times the final best pipeline is the same as one of the recommendations, but extended with at least one preprocessing step. For example, we see that for the NN-ALL method, in 27.7% of cases, a recommendation ends up being the best pipeline found, and in 26.7% of cases the best pipeline is a recommendation with one or more preprocessing steps added.

From Table 4.2, we deduce that the suggestions are often used in the final pipeline, either with preprocessing or without. For both NN methods, over half of the best pipelines are formed with one recommendation as the final learner. In about half those times, a preprocessing step is added.

This is in contrast with the random method, for which the recommendations were almost never used. Thus the suggestions given by the meta-learners were useful.

Method	Exact	Preprocessing
NN-ALL	0.277	0.267
NN-CLF	0.254	0.256
RFR	0.191	0.285
Perfect	0.323	0.329
Rand-base	0.008	0.012

Table 4.2: The fraction of recommendations which were used in the final best pipeline, with and without preprocessing steps added.

In conclusion, various methods to construct a meta-model to recommend good pipelines were evaluated. One method in particular, based on a nearest neighbor approach, proved to be better than random initialization in many cases. However, the comparison is based on a very restricted version of TPOT, only considering a few simple base learners, due to time and resource constraints. The default version of TPOT which considers a wide array of learners and preprocessors still performs significantly better in most cases. However, this work indicates that incorporating meta-learning to warm-start TPOT could lead to significant to improvements.

4.3 Future Work

There are many ways to extend the use of meta-learning for TPOT. One way is to expand the meta-dataset from which the meta-learners can learn. This can be done by including more algorithms and hyperparameter configurations, including preprocessing steps and evaluating the configurations on more datasets.

Another avenue to expand Meta-TPOT is to guide mutation and crossover using meta-learning. Pipelines experience mutations and crossover when going from one generation to another. Currently the decision on which pipeline will incur which change is decided randomly. Potentially, meta-learning might improve this process in one of two ways. It can be used to select which candidates are most likely to benefit from mutation or crossover or it can be used to decide which parts of the pipeline should be mutated and in which manner. This can be done in conjunction with hyperparameter optimization methods such as Bayesian optimization.

Over the course of the genetic algorithm, many pipelines will be evaluated on the dataset, and this gives new information about the data and each learner's capacity on that data. This information can be used to update the meta-model during the optimization process and improve its accuracy. An example of this type of online model adjustment is seen in Active Testing [43, 1]. The updated model can be used to improve on the above proposed meta-learning guided cross-over and mutation, but it can also be used to introduce some new individuals to the population.

Chapter 5

Conclusions

This thesis explores two ways in which to improve the AutoML tool TPOT. TPOT constructs, evaluates and optimizes machine learning pipelines automatically by genetic programming.

The first improvement is Layered TPOT (LTPOT). LTPOT aims to reduce the time spent evaluating bad pipelines. It does this by introducing a layered structure and evaluating pipelines on incrementally larger subsets of the data, only allowing the best performing pipelines at each layer to be evaluated on more data.

The second improvement is MetaTPOT. Meta-learning is applied to learn a model which can recommend promising algorithm configurations for unseen datasets. Using this meta-model's recommendations, the optimization process is warm-started.

5.1 Layered TPOT

LTPOT introduces a layered structure to TPOT. The layered structure separates the competition between individuals trained on different amounts of data. At each layer, the genetic algorithm is ran for a specified amount of generations. At each subsequent layer, individuals are evaluated on increasingly larger subsets. After a set amount of generations, pipelines which worked well in one layer will be transferred to the next, so it may be evaluated on more data. This allows LTPOT to find and discard pipelines which are unlikely to perform well, before training them on all the data.

LTPOT was evaluated on a selection of 18 large datasets. We found that while it does not produce significantly better or worse pipelines than TPOT, it did find good pipelines earlier. This means that it can find better pipelines given a limited time budget, which makes LTPOT suitable for scenario's where there are time or resource constraints. Examples where LTPOT might be preferred include quick data exploration by an expert, or finding better pipelines for the novice that does not have a significant computational budget.

However, because LTPOT discards pipelines based on the performance on a subset of the data it might discard good pipelines too. While all learners are dependent on the amount of data to create a good model, some are more so than others. One pipeline might learn a mediocre model given little data, but stay mediocre with more data to learn from. Another pipeline might learn a bad model given little data, but learns a great one given enough data. In this case, LTPOT will discard the second one, based on experiments done on the small subset. So, LTPOT might miss out on finding some good machine learning pipelines.

In future work, to mitigate the problem of discarding 'eventually-good' pipelines early, different ways to integrate learning curve prediction can be explored. The most promising of which seems to be learning curve prediction using meta-learning [70, 24].

In this work, there is not much research done on the effect of the different hyperparameters that LTPOT introduces. For example, given very large datasets, it might be useful to introduce additional layers, further reducing the duration of the initial evaluations. It is also possible to further explore the effect of the number of generations between transferring individuals, or determine what the right amount of individuals to transfer to the next layer is. It may be useful to adapt these hyperparameters based on dataset characteristics.

5.2 MetaTPOT

MetaTPOT improves on TPOT by warm-starting the optimization by recommending which algorithm configurations to include in the first generation. These recommendations are given by a model learned through meta-learning. Different ways to construct meta-learned recommendations are evaluated.

Two of them used a nearest neighbors approach, based on nearest neighbors after a feature selection technique discussed in [60]. The third approach predicted the accuracy per configuration using a Random Forest Regressor model, and recommended configurations based on predicted accuracy.

In our initial experiments we explored MetaTPOT in a limited setting.The recommendations for individuals were based on only a small selection of learners and hyperparameter values that TPOT considers by default. This was done to obtain the most realistic results under time constraints. Moreover, preprocessing steps were also not considered, meaning that the initial suggestions always consisted of solely the baselearner with its hyperparameter configuration.

MetaTPOT was compared to three baselines and an Oracle. It is compared to two methods which randomly recommend configurations, one with preprocessing steps, the other without. MetaTPOT is also compared to TPOT that is not restricted to the limited selection of algorithms. Finally, the oracle suggested the best known algorithm configurations.

Although our experiments were limited to a constrained version of TPOT, they showed that the kNN approaches improved over initiating TPOT with random suggestions under the same constraints, and in most cases were not significantly worse than the Oracle baseline. In particular, the nearest neighbor approach which modeled algorithm selection and hyperparameter configuration together as a single problem achieved the best performance of the three proposed methods. The results of this work show that using meta-learning to warm-start the genetic optimization in this setting also has merit.

This work can be naturally extended by including all learners and hyperparameter configurations, as well as considering preprocessing steps. Another way to use meta-learning in TPOT is to guide the evolutionary process, like suggesting mutations or individuals for crossover. Finally, a lot of machine learning experiments are run on the data during the optimization process of TPOT. One could aim to learn from these experiments in a similarly to Active Testing [43]. In TPOT, the meta-model can be updated based on evaluations and propose new pipelines to try out as the evolutionary algorithm progresses.

Bibliography

[1] Salisu Mamman Abdulrahman, Pavel Brazdil, Jan N Van Rijn, and Joaquin Vanschoren. Algorithm selection via meta-learning and sample-based active testing. *MetaSel@ PKDD/ECML*, 1455:55–66, 2015. 3, 41

[2] N. S. Altman. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician*, 46(3):175–185, 1992. 5, 6

[3] Rimah Amami, Dorra Ben Ayed, and Noureddine Ellouze. Practical selection of svm supervised parameters with different feature representations for vowel recognition. *arXiv preprint arXiv:1507.06020*, 2015. 6

[4] Hilan Bensusan and Christophe Giraud-Carrier. *Discovering Task Neighbourhoods through Landmark Learning Performances*, pages 325–330. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000. 10

[5] Hilan Bensusan, Christophe Giraud-Carrier, and Claire Kennedy. A higher-order approach to meta-learning. Technical report, University of Bristol, Bristol, UK, UK, 2000. 10

[6] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13:281–305, February 2012. 14

[7] James Bergstra, Brent Komer, Chris Eliasmith, Dan Yamins, and David D Cox. Hyperopt: a python library for model selection and hyperparameter optimization. *Computational Science & Discovery*, 8(1):014008, 2015. 16

[8] James Bergstra, Dan Yamins, and David D. Cox. Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms. In Stéfan van der Walt, Jarrod Millman, and Katy Huff, editors, *Proceedings of the 12th Python in Science Conference*, pages 13 – 20, 2013. 16

[9] James S. Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs 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. 16, 35

[10] Bernd Bischl, Jakob Richter, Jakob Bossek, Daniel Horn, Janek Thomas, and Michel Lang. *mlrMBO: A Modular Framework for Model-Based Optimization of Expensive Black-Box Functions*, 2017. 15

[11] Benjamin Blankertz, Guido Dornhege, Steven Lemm, Matthias Krauledat, Gabriel Curio, and Klaus-Robert Mller. The berlin brain-computer interface: Machine learning based detection of user specific brain states. *Journal of Universal Computer Science*, 12:581–607, jun 2006. http://www.jucs.org/jucs_12_6/the_berlin_brain_computer. 1

[12] T. Blickle and L. Thiele. A comparison of selection schemes used in evolutionary algorithms. *Evolutionary Computation*, 4(4):361–394, Dec 1996. 18

[13] Pavel Brazdil, João Gama, and Bob Henery. *Characterizing the applicability of classification algorithms using meta-level learning*, pages 83–102. Springer Berlin Heidelberg, Berlin, Heidelberg, 1994. 9, 11

[14] Pavel Brazdil, Christophe Giraud-Carrier, Carlos Soares, and Ricardo Vilalta. *Development of Metalearning Systems for Algorithm Recommendation*, pages 31–59. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. 10, 11

[15] Pavel B. Brazdil, Carlos Soares, and Joaquim Pinto da Costa. Ranking learning algorithms: Using ibl and meta-learning on accuracy and time results. *Machine Learning*, 50(3):251–277, Mar 2003. 11

[16] Brent Komer, James Bergstra, and Chris Eliasmith. Hyperopt-Sklearn: Automatic Hyperparameter Configuration for Scikit-Learn. In Stéfan van der Walt and James Bergstra, editors, *Proceedings of the 13th Python in Science Conference*, pages 33 – 39, 2014. 16

[17] Thomas M. Breuel. The effects of hyperparameters on SGD training of neural networks. *CoRR*, abs/1508.02788, 2015. 6

[18] Eric Brochu, Vlad M. Cora, and Nando de Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *CoRR*, abs/1012.2599, 2010. 14

[19] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. *CoRR*, abs/1603.02754, 2016. 20

[20] Marc Claesen and Bart De Moor. Hyperparameter search in machine learning. *CoRR*, abs/1502.02127, 2015. 12

[21] Alex G. C. de Sá, Walter José G. S. Pinto, Luiz Otavio V. B. Oliveira, and Gisele L. Pappa. *RECIPE: A Grammar-Based Framework for Automatically Evolving Classification Pipelines*, pages 246–261. Springer International Publishing, Cham, 2017. 17

[22] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, Apr 2002. 21

[23] J. Demsar. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine Learning Research*, 7:1–30, 2006. 30

[24] Tobias Domhan, Tobias Springenberg, and Frank Hutter. Extrapolating learning curves of deep neural networks. In *ICML 2014 AutoML Workshop*, 2014. 43

[25] Katharina Eggensperger, Matthias Feurer, Frank Hutter, James Bergstra, Jasper Snoek, Holger Hoos, and Kevin Leyton-Brown. Towards an empirical foundation for assessing bayesian optimization of hyperparameters. In *NIPS workshop on Bayesian Optimization in Theory and Practice*, volume 10, 2013. 16

[26] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2962–2970. Curran Associates, Inc., 2015. 2, 11, 16, 17

[27] Matthias Feurer, Jost Tobias Springenberg, and Frank Hutter. Initializing bayesian hyperparameter optimization via meta-learning. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, pages 1128–1135. AAAI Press, 2015. 9, 11

[28] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, jul 2012. 20

[29] Frauke Friedrichs and Christian Igel. Evolutionary tuning of multiple svm parameters. *Neurocomputing*, 64:107–117, 2005. 16

[30] Alex S. Fukunaga. Restart scheduling for genetic algorithms. In Agoston E. Eiben, Thomas Bäck, Marc Schoenauer, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature — PPSN V: 5th International Conference Amsterdam, The Netherlands September 27–30, 1998 Proceedings*, pages 357–366, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg. 20

[31] Johannes Fürnkranz and Johann Petrak. An evaluation of landmarking variants. In *Proceedings of the ECML/PKDD Workshop on Integrating Aspects of Data Mining, Decision Support and Meta-Learning (IDDM-2001)*, pages 57–68, 2001. 10

[32] Isabelle Guyon, Imad Chaabane, Hugo Jair Escalante, Sergio Escalera, Damir Jajetic, James Robert Lloyd, Núria Macià, Bisakha Ray, Lukasz Romaszko, Michèle Sebag, Alexander Statnikov, Sébastien Treguer, and Evelyne Viegas. A brief review of the chalearn automl challenge: Any-time any-dataset learning without human intervention. In Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren, editors, *Proceedings of the Workshop on Automatic Machine Learning*, volume 64 of *Proceedings of Machine Learning Research*, pages 21–30, New York, New York, USA, 24 Jun 2016. PMLR. 17

[33] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009. 16

[34] Tin Kam Ho. Random decision forests. In *Document Analysis and Recognition, 1995., Proceedings of the Third International Conference on*, volume 1, pages 278–282. IEEE, 1995. 6

[35] Gregory S. Hornby. ALPS: The age-layered population structure for reducing the problem of premature convergence. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, GECCO '06, pages 815–822, New York, NY, USA, 2006. ACM. 23

[36] Chih-Wei Hsu, Chih-Chung Chang, and Chih-Jen Lin. A practical guide to support vector classification. Technical report, Department of Computer Science, National Taiwan University, 2003. 12

[37] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration (extended version). Technical Report TR-2010-10, University of British Columbia, Department of Computer Science, 2010. Available online: http://www.cs.ubc.ca/~hutter/papers/10-TR-SMAC.pdf. 2, 15

[38] Frank Hutter, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. Algorithm runtime prediction: The state of the art. *CoRR*, abs/1211.0906, 2012. 11

[39] Ian T Jolliffe. Principal component analysis and factor analysis. In *Principal component analysis*, pages 115–128. Springer, 1986. 37

[40] Alexandros Kalousis and Melanie Hilario. Feature selection for meta-learning. In *Proceedings of the 5th Pacific-Asia Conference on Knowledge Discovery and Data Mining*, PAKDD '01, pages 222–233, London, UK, 2001. Springer-Verlag. 36, 40

[41] Abdullah Konak, David W. Coit, and Alice E. Smith. Multi-objective optimization using genetic algorithms: A tutorial. *"Reliability Engineering & System Safety"*, 91(9):992 – 1007, 2006. Special Issue - Genetic Algorithms and Reliability. 20

[42] Oliver Kramer. *Genetic Algorithm Essentials*. Springer International Publishing, 2017. 18, 20

[43] Rui Leite, Pavel Brazdil, and Joaquin Vanschoren. *Selecting Classification Algorithms with Active Testing*, pages 117–131. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. 41, 44

[44] Frank Hung-Fat Leung, Hak-Keung Lam, Sai-Ho Ling, and Peter Kwong-Shun Tam. Tuning of the structure and parameters of a neural network using an improved genetic algorithm. *IEEE Transactions on Neural networks*, 14(1):79–88, 2003. 16

[45] Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Efficient hyperparameter optimization and infinitely many armed bandits. *CoRR*, abs/1603.06560, 2016. 16

[46] Sai-Ho Ling, Frank Hung-Fat Leung, Hak-Keung Lam, Yim-Shu Lee, and Peter Kwong-Shun Tam. A novel genetic-algorithm-based neural network for short-term load forecasting. *IEEE Transactions on Industrial Electronics*, 50(4):793–799, 2003. 16

[47] Ana Carolina Lorena and Andr C.P.L.F. de Carvalho. Evolutionary tuning of svm parameter values in multiclass problems. *Neurocomputing*, 71(16):3326 – 3334, 2008. Advances in Neural Information Processing (ICONIP 2006) / Brazilian Symposium on Neural Networks (SBRN 2006). 16

[48] H. Brendan McMahan, Gary Holt, D. Sculley, Michael Young, Dietmar Ebner, Julian Grady, Lan Nie, Todd Phillips, Eugene Davydov, Daniel Golovin, Sharat Chikkerur, Dan Liu, Martin Wattenberg, Arnar Mar Hrafnkelsson, Tom Boulos, and Jeremy Kubica. Ad click prediction: A view from the trenches. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 1222–1230, New York, NY, USA, 2013. ACM. 1

[49] Brad L Miller, David E Goldberg, et al. Genetic algorithms, tournament selection, and the effects of noise. *Complex systems*, 9(3):193–212, 1995. 18

[50] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA, 1998. 20

[51] R. S. Olson, R. J. Urbanowicz, P. C. Andrews, N. A. Lavender, L. Kidd, and J. H. Moore. Automating biomedical data science through tree-based pipeline optimization. *CoRR*, 2016. 20, 21

[52] Randal S. Olson, Nathan Bartley, Ryan J. Urbanowicz, and Jason H. Moore. Evaluation of a tree-based pipeline optimization tool for automating data science. *CoRR*, abs/1603.06212, 2016. 22, 29

[53] Randal S. Olson, William La Cava, Patryk Orzechowski, Ryan J. Urbanowicz, and Jason H. Moore. PMLB: A large benchmark suite for machine learning evaluation and comparison. *CoRR*, abs/1703.00512, 2017. 25, 29

[54] Randal S. Olson and Jason H. Moore. Tpot: A tree-based pipeline optimization tool for automating machine learning. In Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren, editors, *Proceedings of the Workshop on Automatic Machine Learning*, volume 64 of *Proceedings of Machine Learning Research*, pages 66–74, New York, New York, USA, 24 Jun 2016. PMLR. 2, 17

[55] 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. 17, 20

[56] Yonghong Peng, Peter A Flach, Pavel Brazdil, and Carlos Soares. Decision tree-based data characterization for meta- learning. 10

[57] Bernhard Pfahringer, Hilan Bensusan, and Christophe G. Giraud-Carrier. Meta-learning by landmarking various learning algorithms. In *Proceedings of the Seventeenth International Conference on Machine Learning*, ICML '00, pages 743–750, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc. 9, 10

[58] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A Field Guide to Genetic Programming*. Lulu Enterprises, UK Ltd, 2008. 20

[59] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986. 6

[60] Matthias Reif, Faisal Shafait, and Andreas Dengel. Meta-learning for evolutionary parameter optimization of classifiers. *Machine Learning*, 87(3):357–380, 2012. 3, 35, 37, 44

[61] Matthias Reif, Faisal Shafait, Markus Goldstein, Thomas Breuel, and Andreas Dengel. Automatic classifier selection for non-experts. *Pattern Analysis and Applications*, 17(1):83–96, Feb 2014. 10

[62] F. Samadzadegan, A. Soleymani, and R. A. Abbaspour. Evaluation of genetic algorithms for tuning svm parameters in multi-class problems. In *2010 11th International Symposium on Computational Intelligence and Informatics (CINTI)*, pages 323–328, Nov 2010. 16

[63] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 2951–2959. Curran Associates, Inc., 2012. 14, 15

[64] Carlos Soares, Johann Petrak, and Pavel Brazdil. *Sampling-Based Relative Landmarks: Systematically Test-Driving Algorithms before Choosing*, pages 88–95. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. 10

[65] Nitasha Soni and Tapas Kumar. Study of various crossover operators in genetic algorithms. *International Journal of Computer Science and Information Technologies*, 5(6):7235–7238, 2014. 18

[66] Nitasha Soni and Tapas Kumar. Study of various mutation operators in genetic algorithms. *International Journal of Computer Science and Information Technologies*, 5(3):4519–4521, 2014. 18

[67] Chris Thornton, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 847–855, New York, NY, USA, 2013. ACM. 1, 2, 11, 16

[68] Jinn-Tsong Tsai, Jyh-Horng Chou, and Tung-Kuan Liu. Tuning the structure and parameters of a neural network by using hybrid taguchi-genetic algorithm. *IEEE Transactions on Neural Networks*, 17(1):69–80, Jan 2006. 16

[69] Jan N. van Rijn, Geoffrey Holmes, Bernhard Pfahringer, and Joaquin Vanschoren. *Algorithm Selection on Data Streams*, pages 325–336. Springer International Publishing, Cham, 2014. 9, 11, 17

[70] J.N. van Rijn, S.M. Abdulrahman, P. Brazdil, and J. Vanschoren. Fast algorithm selection using learning curves. In *Lecture Notes in Computer Science (IDA 2015)*, volume 9385, pages 298–309, 2015. 3, 25, 43

[71] J.N. van Rijn, G. Holmes, B. Pfahringer, and J. Vanschoren. Algorithm selection on data streams. *Lecture Notes in Computer Science (Proceedings of Discovery Science 2014)*, 8777:325–336, 2014. 29

[72] Joaquin Vanschoren, Jan N. van Rijn, Bernd Bischl, and Luis Torgo. OpenML: Networked science in machine learning. *SIGKDD Explorations*, 15(2):49–60, 2013. 29

[73] Ricardo Vilalta, Christophe G. Giraud-Carrier, Pavel Brazdil, and Carlos Soares. Using meta-learning to support data mining. *IJCSA*, 1(1):31–45, 2004. 9

[74] B Yekkehkhany, A Safari, S Homayouni, and M Hasanlou. A comparison study of different kernel functions for svm-based classification of multi-temporal polarimetry sar data. *The International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, 40(2):281, 2014. 6