**Technical University of Berlin**

Institute of Software Engineering and Theoretical Computer Science
Database Systems and Information Management Group

Faculty EECS (IV)
Einsteinufer 17
10587 Berlin
http:/www.dima.tu-berlin.de


Master Thesis

# Efficient Automatic Machine Learning using Experiment Databases

Milad Abbaszadeh Jahromi

Student Number: 389928
19.06.2020

Supervisor:
Prof. Dr. Volker Markl

Advisors:
Behrouz Derakhshan
Dr. Alireza Rezaei Mahdiraji


DFKI Institute
Alt-Moabit 91c
10559 Berlin

This dissertation originated in cooperation with the German Research Center for Artificial Intelligence (DFKI).

First of all, I would like to thank Prof. Dr. Volker Markl at the DFKI for giving me the opportunity to carry out the state of the art research in this field.

Special thanks to Mr. Derakhshan and Dr. Rezaei Mahdiraji for their guidance. They were always open whenever I ran into a trouble spot or had a question about my research or writing. They consistently allowed this thesis to be my work but steered me in the right direction whenever they thought I needed it.

Finally, I must express my very profound gratitude to my parents and my friends to provide me with unfailing support and continuous encouragement throughout my years of study and through researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

I hereby declare that the thesis submitted is my own, unaided work, completed without any unpermitted external help. Only the sources and resources listed were used.

The independent and unaided completion of the thesis is affirmed by affidavit:

. . . . . .

*Milad Abbaszadeh Jahromi - Berlin, 19.06.2020*

**Abstract**

Machine learning has become increasingly successful during the last decades, and this success brings a lot of non-experts to the field. Automatic Machine Learning (AutoML) is the phenomena that makes the process of machine learning easy for novice users and as its name suggests, tries to automate the machine learning steps. The goal of AutoML is to find the best pipeline components (i.e., preprocessing, feature extraction, feature and model selection, hyperparameter tuning, and model interpretation) that minimize an error, or maximize the quality of the provided solution based on ML metrics. AutoML faces big challenges, because finding the best algorithm and tuning hyperparameters for selected models is hard even for humans. However, some AutoML systems have been successful during recent years.

Bayesian Optimization is the key success for AutoML. It is an iterative process that takes into account the information provided in each iteration, and uses this knowledge to find a valuable part of search space and eventually provides the best-found configuration. Bayesian Optimization is different in the way that they formulate the optimization. In this thesis we focus on TPE (Tree-structured Parzen Estimator) which is an algorithm that uses the insight from former configurations to provide a new set of parameters.

Current AutoML tools have two significant drawbacks. First, the space of available pipeline components must be defined beforehand. It is not possible to import experimental databases before the process starts. Thus, domain knowledge cannot be incorporated in the design process. Secondly, the process is very time-consuming and resource-intensive due to the large size of the search space.

This thesis examines the possibility of warm-starting Bayesian Optimization with experimental databases like OpenML. Our goal is to enable the AutoML to use experimental databases (OpenML) to design the search space and guide the search process and improve AutoML performance. Moreover, we compared the performance of our tool, which is called *HyperOpenML*, that enables using an experimental database with the vanilla version of hyperopt. Besides, we have studied the exploited pipelines from OpenML due to pruning the required number of instances for warm-starting.

**Abstrakt**

Machine learning ist in den letzten Jahrzehnten immer erfolgreicher geworden, und dieser Erfolg führt viele Nicht-Experten in den Bereich. Automatisches machine learning (AutoML) ist ein Phänomen, das Anfängern den maschinellen Lernprozess erleichtert und, wie der Name schon sagt, versucht, die maschinellen Lernschritte zu automatisieren. Das Ziel von AutoML besteht darin, die besten Pipeline-Komponenten (d. h. Vorverarbeitung, Merkmalsextraktion, Merkmal- und Modellauswahl, Hyperparameter - Optimierung und Modellinterpretation) zu finden, die einen Fehler minimieren oder die Qualität der bereitgestellten Lösung basierend auf ML-Metriken maximieren. AutoML steht vor grossen Herausforderungen, da es selbst für Menschen schwierig ist, den besten Algorithmus zu finden und Hyperparameter für ausgewählte Modelle abzustimmen. Einige AutoML-Systeme waren jedoch in den letzten Jahren erfolgreich.

Bayesian Optimization(Bayes'sche Optimierung) ist der Schlüsselerfolg für AutoML. Es ist ein iterativer Prozess, der die in jeder Iteration bereitgestellten Informationen berücksichtigt und dieses Wissen verwendet, um einen wertvollen Teil des Suchraums zu finden und schliesslich die am besten gefundene Konfiguration bereitzustellen. Bayesianische Optimierungen unterscheiden sich in der Art und Weise, wie sie die Optimierung formulieren. In dieser Arbeit konzentrieren wir uns auf TPE (Tree-Structured Parzen Estimator), einen Algorithmus, der die Erkenntnisse aus früheren Konfigurationen verwendet, um einen neuen Satz von Parametern bereitzustellen.

Aktuelle AutoML-Tools weisen zwei wesentliche Nachteile auf. Als Erstes muss der Platz der verfügbaren Pipelinekomponenten im Voraus definiert werden. Es ist nicht möglich, experimentelle Datenbanken zu importieren, bevor der Prozess beginnt. Daher kann Domänenwissen nicht in den Entwurfsprozess einbezogen werden. Zweitens ist der Prozess aufgrund der Grösse des Suchraums sehr zeitaufwändig und ressourcenintensiv.

Diese Arbeit untersucht die Möglichkeit einer warmstartenden Bayes'schen Optimierung mit experimentellen Datenbanken wie OpenML. Unser Ziel ist es, AutoML die Verwendung experimenteller Datenbanken (OpenML) zu ermöglichen, um den Suchraum zu entwerfen, den Suchprozess zu steuern und die AutoML-Leistung zu verbessern. Darüber hinaus haben wir die Leistung unseres Tools namens *HyperOpenML* verglichen, das die Verwendung einer experimentellen Datenbank mit der Vanille-Version von hyperopt ermöglicht. Außerdem haben wir die genutzten Pipelines von OpenML untersucht, da die erforderliche Anzahl von Instanzen für den Warmstart entfernt wurde.

# Contents

# List of Figures

# List of Tables

# 1 Introduction

Machine learning (ML) creates a paradigm shift from traditional rule-based programming and enables the programmers to teach the machine, instead of programming the logic explicitly. Machine learning's objective in contrast to the traditional programming, is learning the mathematical function that maps an input to a specific output. More precisely, the data is a multidimensional array known as features, and the output could be a real number, a class or a structure [1].

In addition to the learning algorithms, we need to specify several choices before solving a given ML problem. The first choice is to select the preprocessor or data transformer, so that the machine learning algorithm gets the optimal result. The second choice is to find the model or learning algorithm that solves the given problem efficiently. Although the machine learning model learns some parameters during the training phase, there are still some configurable parameters that we need to specify before the beginning, which we call hyperparameters. Finally, the last step is to tune the hyperparameters of the selected algorithm.

The goal of any machine learning pipeline is to provide the solution as accurately as possible. Selecting each of the mentioned choices significantly affects the final result. There are many data preprocessors and machine learning algorithms that we can apply to solve a specific problem. At the same time, these algorithms have hyperparameters, and each of these hyperparameters can have a finite set or sometimes infinite set of options. Therefore, the search space of these configurations is enormous and finding the best configuration is challenging.

Most commonly, finding the hyperparameters for each algorithm or, in general, finding the best configuration for a given problem, is known as a human task. In practice, choosing one configuration at a time and checking its result was the most common approach until very recently. Some researchers still use this approach in the research community. Consequently, researchers need to spend a major time to find the best configuration and at the same time explore the values for each hyperparameter [2].

Machine learning has been successful in a variety of fields in recent years. Nowadays, it is not only in the research and development applications, but also available in the enterprise domain. This diversity of applications introduces the idea of using machine learning by non-experts. Automatic machine learning (AutoML), as its name suggests, tries to abstract and automate the processes of machine learning for novice users. AutoML provides (semi-)automated end-to-end processes for applying machine learning to

real-world problems by designing pipelines and tuning the hyperparameters for models [3].

AutoML defines its goal to find the best pipeline elements (i.e., preprocessing, feature extraction, feature and model selection, hyperparameter tuning, and model interpretation) to minimize an error or enhance the quality of the provided solution based on ML metrics. AutoML wants to automatically achieve a result as accurate as possible. Since each step in the machine learning pipeline could affect the final result, choosing a good configuration is crucial for finding a favorable result [2].

## 1.1 Problem Statement

The goal of AutoML is to automatically configure different ML pipeline steps, which is very challenging. Designing the optimal pipeline is extremely difficult even for humans due to the number of available options for each step in the ML package. Consequently, in recent years some AutoML systems were developed.

Auto-sklearn [4] and Auto-WEKA [5] are two examples of AutoML libraries, which focus on finding a suitable model and hyperparameters with the help of Bayesian Optimization. Moreover, Bayesian Optimization is one of the three common approaches (i.e., grid search and random search) of performing hyperparameter optimization. Bayesian Optimization keeps track of past evaluation results, in contrast to random and grid search. Bayesian Optimization uses stored knowledge, to form a probabilistic model for mapping the hyperparameters to the past evaluation results. This probabilistic model, enables Bayesian Optimization to propose new hyperparameters and evaluate results in finding a high-quality model in a shorter amount of time [6].

Current AutoML tools have two significant drawbacks. First, It is not possible to import experimental databases before the process starts. Thus, existing tools [4, 7, 5] cannot utilize domain knowledge in the design process. It means the space of available pipeline components (i.e., the types of data operations and training algorithms) must be defined beforehand (regardless of experimental database insight). Second, the process is very time-consuming and resource-intensive due to the large size of the search space (i.e., all the available data operations, training algorithms, and hyperparameters).

This thesis examines the possibility of warm-starting Bayesian Optimization with experimental database like OpenML [8]. We enable AutoML to use this knowledge to define the search space and guide the search process, which finally leads to better results. To achieve this approach, we utilize OpenML, an online repository of training datasets, and user-defined machine learning pipelines. More precisely, OpenML enables AutoML to track the execution of the user-defined pipelines on the training datasets and captures several publicly available metrics such as execution time and evaluation scores to offer a new configuration. Moreover, after an investigation on the exploited pipelines

from OpenML, we have tried different strategies to prune the exploited pipelines from OpenML to reduce the required time in Bayesian Optimization.

## 1.2 Scope

In this thesis, we focus on classification tasks. However, using the experimental database is not limited to classification and the proposed solution could easily be extended to other tasks as well. Also, similar to Auto-sklearn, we focus on the scikit-learn [9] package for all steps. We only exploit pipelines from OpenML that have three steps (data preprocessing, feature preprocessing, and classifier) for a given dataset.

## 1.3 Outline

The remaining part of this thesis is organized as follows. Chapter 2 presents the background of the field, including a different approach of hyperparameter optimization in-depth, then introduces AutoML and its tools architectures. This chapter continues with Hyperopt and OpenML as two bases of this thesis. Chapter 3 presents the HyperOpenML, proposed approach for warm-starting the Bayesian Optimization, and describes the characteristic of the exploited experimental databases from OpenML. Chapter 4 describes the details of how HyperOpenML is implemented. In Chapter 5 the comparison between different approaches in HyperOpenML are shown in a range of problems with varying datasets. Finally, Chapter 6 gives final remarks and suggests future work lines based on this work.

# 2 Background

This chapter covers the necessary background of this thesis. It starts with hyperparameter optimization and its algorithms, and then continues with the essential approaches in more details. Moreover, it follows with well known AutoML systems and their architectures. This chapter concludes with OpenML as an online experimental database.

## 2.1 Hyperparameter Optimizations and Algorithms

Hyperparameters are specific values that control the behavior of a typical ML algorithm, and they are set before the training process starts. Parameters are values that the algorithm learns during the training phase.

Hyperparameter optimization (HPO) improves the performance of a machine learning model by choosing a right-set of hyperparameters. The goal of HPO is to decrease a specific loss on the output. Although there are several ways of achieving HPO, some of them treat the problem as black-box optimization. In more details, recently scientists consider HPO as a black-box that receives the search space of hyperparameters, and gives a precise output (loss on validation dataset) [5].

HPO has many challenges. Finding the best combination of hyperparameters is complicated, and iterating over lots of candidate sets is time-consuming [2]. There is always a risk of model overfitting [10] on the training data and ultimately reaching high performing results on the training set. In contrast, the designed model has a significant loss on the validation dataset. Although there are various regularization techniques [11] to overcome overfitting, it is still a common problem when tuning the hyperparameters of the machine learning algorithms.

Traditionally, HPO was the job of humans because they can be efficient and wisely change the hyperparameters. These days, with increase in the size of the search space and at the same time complexity in the models, we need to automate this process. Fortunately, it is possible to automate the HPO and find better results, outperforming humans [12]. In the following, we review the most popular HPO approaches.

### 2.1.1 Grid Search

Grid search is the most straightforward approach for hyperparameter optimization. This approach creates a grid out of the values that the user defines for each hyperparameter

and tries every single combination to find the combination which achieves the minimum loss on the validation set.

The grid search needs to examine all combinations, the Cartesian product [13] of value sets of all hyperparameters. As a result, the size of the search grid increases exponentially when the number of hyperparameters rises linearly. Although this algorithm searches all parts of the search space blindly and is computationally inefficient, it guarantees to find the best possible combination of the given candidates. Moreover, it is easy to implement, and it is a straightforward algorithm to parallelize [12], as each hyperparameter combination can be executed independently. Figure 2.1 left side, shows an example layout of trial points of grid search in two dimensions.

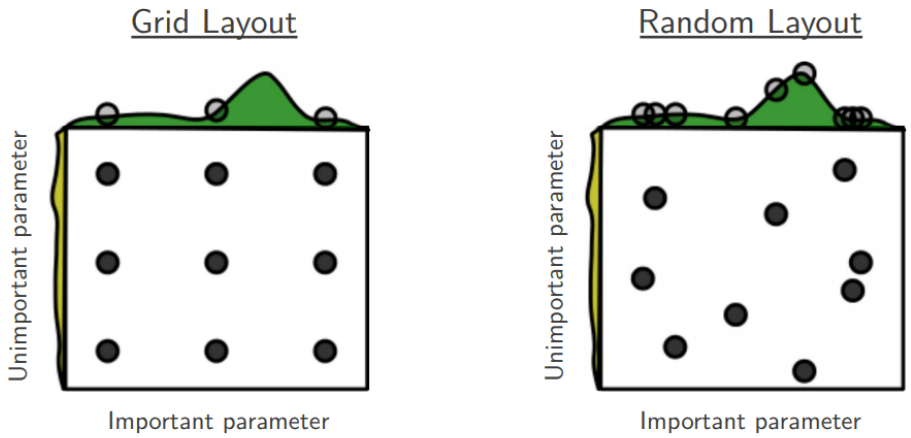



Figure 2.1: Example Layout of Trial Points of Grid and Random Search in Two Dimensions [12]

### 2.1.2 Random Search

Random search [12] follows the same path as grid search in that it executes HPO processes independent from each other. However, it does not sequentially test all combinations, instead trying random combinations among the range of values that are specified for hyperparameters.

The main advantage of random search in comparison to grid search is that it tries a vast range of values and it gives the user control over the number of candidates to try based on time and resources. These features in random search become even more critical when dealing with many hyperparameters. Of course, all of the hyperparameters do not have the same impact on the final metric. In this situation, the random search would try a new value for each hyperparameter in each iteration and ends up to sufficiently large search space. In contrast to random search, grid search works on a relatively small

number of different values for a hyperparameter [12].

Even though grid search allocates many more resources to explore part of the search space (it is not essential), it finds the best combination. Meanwhile random search, as its name suggests, may not reach the best combination due to its randomness. Therefore, it does not benefit from each combination sight.

Figure 2.1 represents a grid and random search for nine combinations. This example represents two hyperparameters. One hyperparameter (top of each square, in green) has a higher impact on the result compared to another (left of the square, in yellow). As we can see in Figure 2.1, grid search only tried three distinct places on the important hyperparameter while random search tried all nine trials to explore the important spot.

In short, while grid search and random search both have pros and cons, it is also possible to use them in a row. Instead of using them individually, it is possible to use each of them in a scenario. The random search could be used as the first step to speed up the exploration of a broader range of values, and then grid search could be used as the second step to search the desired spot more precisely. Figure 2.2 demonstrates this approach visually. It shows, first random search is run to find a suitable range for each hyperparameter, and then those ranges are searched more closely by grid search. Although the combination of random search and grid search could save computational resources and time, the main limitation of both approaches still exists. It means neither grid search nor random search considers the effect of each combination for proposing the set of the configuration. To overcome this limitation, Subsection 2.1.3 introduces Bayesian Optimization.

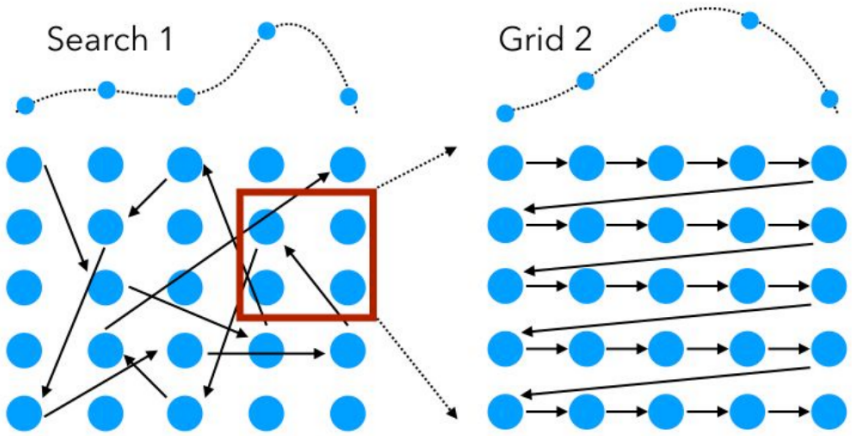



Figure 2.2: Use random search and grid search in a row. It is possible to find the interesting part of the search space faster by random search and precisely search it by grid search [14].

### 2.1.3 Bayesian Optimization

Bayesian Optimization is a more sophisticated method for hyperparameter optimization. Unlike grid and random search, Bayesian Optimization takes into account the information provided in the past evaluation of the hyperparameters. In other words, it offers the hyperparameters based on recent reports in the sequence manner. For this reason, the Bayesian Optimization approach is known as sequential model-based optimization (SMBO) or adaptive configuration methods. This method tries to adapt the configuration sequentially based on the previous results [15].

Training different configuration sets of hyperparameters to find the best (good enough) configuration is an expensive task. SMBO has two essential parts to counterbalance this cost.

Firstly, this method predicts the result of a new set of hyperparameters by a function that is much cheaper than the main one (to evaluate it on the validation dataset). This alternative function is known as Surrogate Function [16] and receives a set of hyperparameters as input, and finds the prediction of the loss on the validation dataset as an output. Surrogate function is continuously updated based on past information after training and testing on a previous set of hyperparameters. This approach makes it possible to learn how changing the hyperparameters could change the final result. This learning makes it possible to predict which part of the search space is more promising and prevent SMBO from exploring the non-promising part.

Secondly, there is another part that decides about the next set of hyperparameters, which we know as the acquisition function [17, 18].

**Algorithm 1** Sequential Model-based Optimization

1: Input: f, $M_0$, T, S
2: $H \leftarrow \emptyset$
3: **for** t $\leftarrow$ 1 to T **do**
4: $x^* \leftarrow \arg\min_x S(M_{t-1})$
5: Evaluate $f(x^*)$ ▷ (Expensive step)
6: $H \leftarrow H \cup (x^*, f(x^*))$
7: Fit a new Model $M_t$ to H
8: return H

Algorithm 1 describes the behavior of the SMBO algorithm in pseudo-code [19]. As mentioned, this algorithm mostly is used when evaluating the actual function is costly. SMBO approximates the expensive function with a surrogate that is much cheaper to evaluate. In more details, this algorithm receives the real fitness function f, the initial model $M_0$, the number of trials $T$, and surrogate function $S$ [19] (Line 1). It starts with null history (Line 2) and tries to find the best possible solution in the limited amount of

iteration $T$ (Line 3). In each iteration, based on the surrogate function, the acquisition function [19] proposes a set of configurations that one of them amounts to the best loss (minimum loss) and saves as $x^*$ (Line 4), the fitness function $f$ evaluates the proposed configuration (Line 5), and algorithm adds this configuration to history (Line 6). As the last step in each iteration, model $M$ fits history to cover the updates and enrich the surrogate function(Line 7). In the end, discovered history in $T$ iterations will be returned as algorithm's output. In summary, the loop in the algorithm is a numerical optimization of surrogate function or transformation of the surrogate function, which is cheaper to evaluate in comparison to the actual fitness function [19].

SMBO algorithms are different. Firstly, there are differences in the model that describes the distribution of the loss in terms of hyperparameters values and secondly, in the ways that they select the next set of configurations. In more details, they are different in surrogate function (any regression model or a more complex model) and acquisition function.

$$EI_{y^*}(x) := \int_{-\infty}^{+\infty} max(y^* - y, 0) p_M(y|x) dy \tag{2.1}$$

There are several algorithms for selecting a new configuration, like probability of improvement [20], minimizing the condition entropy of minimizer [21], and bandit-based creation [22]. This thesis, similar to several well-known AutoML systems, focuses on the expected improvement (EI) criterion [20]. EI is intuitive and works well in a variety of settings. As Equation 2.1 shows, the *EI is the expectation under some model $M$ of $f: X \rightarrow R^N$ that $f(x)$ exceeds some threshold $y^*$* [19]. In this Equation, $x$ represents the hyperparameters, $y^*$ is the target performance, and $y$ is the loss corresponding to a given configuration.

Subsection 2.1.3.1 covers the Tree-structured Parzen Estimator (TPE) as the backbone of Hyperopt. TPE is a novel strategy for approximating the expensive function $f$ by considering surrogate function. Other well-known strategies are briefly introduced in Subsection 2.1.3.2.

#### 2.1.3.1 TPE

TPE (Tree-structured Parzen Estimator) [19] is an algorithm that uses the insight from former configurations to provide a new set of parameters. Bayes' theorem [23] could formulate the loss ($y$) in hyperparameters optimization according to a given configuration ($x$) like below.

$$p(y|x) = \frac{p(x|y)p(y)}{p(x)} \tag{2.2}$$

TPE is one of the most popular choices for hyperparameter optimization. This algorithm was developed with a primary focus on the modeling of $p(y)$ and $p(x|y)$ in surrogate function, instead of modeling $p(y|x)$ directly.

TPE algorithm first sorts the observed configurations based on their losses then constructs two different probability functions depending on a threshold loss ($y^*$) to make a new proposal. Equation 2.3 describes this fact in more details.

$$p(x|y) = \begin{cases} l(x), & if\ y < y^* \\ g(x), & if\ y \geq y^* \end{cases} \tag{2.3}$$

Where $l(x)$ is the density formed by the observations, which are known as good configurations. These observations have corresponding loss $f(x^{(i)})$ less than target performance $y^*$ while $g(x)$ is the density which is formed by the rest of observations. TPE algorithm depends on $y^*$ that is larger than the best observed loss so that some points still can use to form $l(x)$. It means TPE algorithm chooses the $y^*$ in the way that $p(y < y^*) = \gamma$ [19].

In other words, TPE builds two probabilistic models. Firstly, for parameters that perform better than $\gamma$-quantile. Secondly, for hyperparameters, which perform worse than $\gamma$-quantile. As the original TPE's paper [19] proved, TPE employs the Equation 2.1 to find points with the highest EI.

$$EI(x) \propto (\gamma + \frac{g(x)}{l(x)}(1-\gamma))^{-1} \tag{2.4}$$

Equation 2.4 finds the point $x$ with high probability under the distribution of $l(x)$ and low probability under the distribution of $g(x)$. Because TPE favors the tree-structured form, it is easy to draw many candidates according to $l$ and $g$ and evaluate them based on $g(x)/l(x)$. However, only the candidate $x^*$ with highest EI is selected.

One clear advantage of TPE is that the dependency between hyperparameters are explicitly defined. Hyperparameters that are not part of the given configuration (inactive hyperparameters) never influence expected improvement or surrogate function's results after training. This feature can be beneficial for highly conditional scenarios such as a deep neural network or automatic machine learning. Moreover, it is more efficient than SMAC (Sequential Model-based Algorithm Configuration, see Section 2.1.3.2) in some points because it does not need to learn the conditional dependencies based on the data. Furthermore, TPE is able to explicitly model numerical and categorical variables without any transformation.

Regardless of TPE capabilities [24], there is a premium cost when working with this algorithm. TPE assumes hyperparameters are independent. This means, TPE is able

to track only dependencies of being active or inactive depending on different hyperparameters. This oversimplification against another algorithm like SMAC could remark as an incapability in modeling joint probabilities. The value proposed for one hyperparameter does not influence any other ones. However, in some cases the value of one hyperparameter can be highly correlated or even dependent on other values from other hyperparameters. This direct influence can not be captured in the TPE algorithm [25].

#### 2.1.3.2 Other Algorithms for SMBO

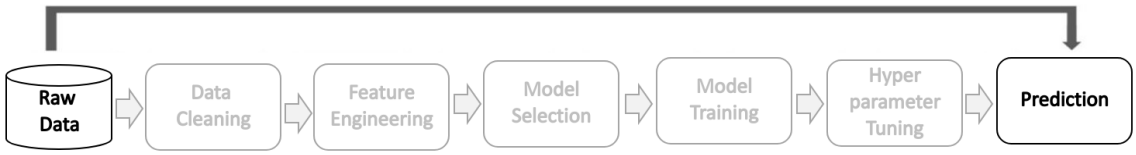
The most successful methods in literature try to model the surrogate function with different strategies. SMAC [26] is one of the widely used algorithms. It is a popular choice for hyperparameter optimization. SMAC tries to find hyperparameters according to a set of related objectives instead of only one task. SMAC employs random forest [27] as the surrogate function. As shown in [28], this algorithm has an excellent performance in a variety of challenges. Random forests are ensemble methods that combine individual outputs by decision trees and have several hyperparameters. SMAC's authors fixed these hyperparameters to some fixed values. SMAC is one of the popular hyperparameter optimization and is not further explored in this thesis. To find a comprehensive description of the mechanism involved in the optimization of SMAC the reader can refer to their original paper [26].

Another technique is Gaussian Processes (GP) [29], and as its name suggests, the surrogate function is based on Gaussian Processes. Essentially, GP is a regression technique which is a generalization of the Gaussian probability distribution in the function space. GPs provide an assessment of prediction uncertainty, which includes data scarcity, making the GPs an excellent candidate for surrogate function. Expected improvement is also used in this technique to balance exploitation and exploration. Exploitation focuses on the part of space that the mean of function is close to target performance. Exploration focuses on under-explored regions where the algorithm is not sure about the space and has a high uncertainty. Moreover, the run time of the GP approach in each iteration scales linearly in the number of the variables being optimized and is cubical in the size of given history. For a more detailed description of Gaussian Processes the reader can refer to their original paper [30].

## 2.2 AutoML Systems

Machine learning has gained a wide range of success in various fields, and this fact brings many non-experts to the area. As a result, we need some methods and processes to make machine learning available to everyone. These phenomena are known as automated machine learning (AutoML) which represents the automatic selection of algorithms, feature preprocessing steps for a given dataset and tuning the hyperparameters for a selected model [4].


(a) Traditional Machine Learning Workflow




(b) AutoML Workflow

Figure 2.3: Traditional Machine Learning versus AutoML Workflow [31]

Although AutoML provides machine learning out-of-the shelf for non-experts, it faces with a vast search space for selecting the machine learning algorithm and corresponding hyperparameters. Most of the open-source machine learning packages, such as WEKA [32], mlr [33], and PyBrain[34] ask the user to identify the learning algorithm as well as customizing the hyperparameters. This autonomy can be challenging since lots of possible choices would be available. Users select the algorithms based on their reputations or intuitive application and allow hyperparameters to remain the same as default values. This approach ends with worse results than the best method and sophisticated way for hyperparameters tuning. This obstacle gave a chance to AutoML for automating these selections and tuning the hyperparameters for each algorithm [5].

As Figure 2.3 illustrates, AutoML automates as many steps as possible in the machine learning pipeline. While some of these steps are easier to automate, some others are extremely difficult, e.g., choosing the correct model and tuning the hyperparameters. AutoML systems could focus on various states of machine learning like data preparation, feature engineering, model selection, selection of evaluation metrics, and hyperparameter optimization [35].

Data cleaning [36] and feature engineering [37] are two steps that are largely considered human tasks. Feature engineering in particular requires an in-depth domain knowledge, and involves much trial and error. In most cases, excellent performance of the machine learning algorithm depends on features sets which are selected. Furthermore, humans are capable of creating new features that are not part of the given features. The aim of generating new features is to get more insight into data. Although feature engineering can be difficult for humans, there is an automated framework for this step. Data Science Machine [38] is a research project at MIT which is an end-to-end software system that automatically develops predictive models from relational data. Feature tools [39] is a

python library that automates the feature engineering process.

Once the features have been prepared, it is time to find a suitable model and to tune the corresponding hyperparameters. There are many algorithms in different categories (e.g., clustering, classification, regression). Choosing an algorithm and tuning its hyperparameters are costly tasks. The following subsections introduce two momentous and well-known AutoML systems. The first subsection starts with Auto-WEKA, followed with Auto-sklearn.

### 2.2.1 Auto-WEKA

Nowadays, scientists and researchers are not the only groups who use machine learning algorithms. The use of ML algorithms by non-experts can be challenging. Most of the ML packages ask users to determine the learning algorithm and respective hyperparameters. This degree of freedom makes the process even more difficult for non-experts, leaving them to select the algorithm by their intuitive appeal, and setting the hyperparameter to default values. Of course, this strategy reflects in the performance, which is far from the best answer, and often unacceptable.

Auto-WEKA [5] focuses on classification and considers the AutoML problem as the CASH (Combined Algorithm Selection and Hyperparameter optimization) problem. CASH refers to the idea of combining the learning space of the algorithm with its hyperparameters. As the search space is high dimensional and involves categorical and continuous choices, as well as hierarchical dependencies, it is very challenging. To make it more precise, the hyperparameter which belongs to one algorithm, only selects whenever the algorithm is selected itself. Moreover, CASH considers choosing the algorithm itself as the hyperparameter. It means the algorithm selection and hyperparameter tuning are a single hierarchical hyperparameter optimization.

Scientists tackle the CASH problem in various ways. One of the efficient ways is Bayesian Optimization and specifically SMBO (sequential model-based optimization). For more details the reader can check Section 2.1.3.

Auto-WEKA covers a broad range of feature selection techniques as well as all classification methods available in WEKA [32]. Additionally, it span two ensemble techniques and ten meta-methods. The system does not need any requirements to start, and in practice users could use Auto-WEKA if they can use WEKA itself [5].

The following remarks explain two critical parts of the Auto-WEKA in more details.

- **Model Selection**

  The goal of this step is to find a model that minimizes the generalization error based on available data and machine learning models. Auto-WEKA splits the

data into two separate parts of training and validation. This splitting allows the system to train the model on the training part and evaluates the evaluation part. Auto-WEKA uses cross-validation [40] for splitting the data. Auto-WEKA defines the loss as the miss-classification rate for each model. Then, it tries to minimize loss as much as possible [5]. In this step, choosing one algorithm does not have any influence on choosing other algorithms.

- **Hyperparameter Optimization**

The concept of hyperparameter optimization is similar to model selection. The difference is that the machine learning model is present, and the system needs to tune the corresponding hyperparameters. Most of the time, hyperparameters are continuous and correlated with each other [5]. In some scenarios, hyperparameters can influence each other. For example, the number of nodes in the fifth layer of the neural network is irrelevant if the number of layers is three. In the case of hyperparameters being conditional to other parameters or hyperparameters the search space can be seen as tree-structure space [19] or a directed acycle graph [41].

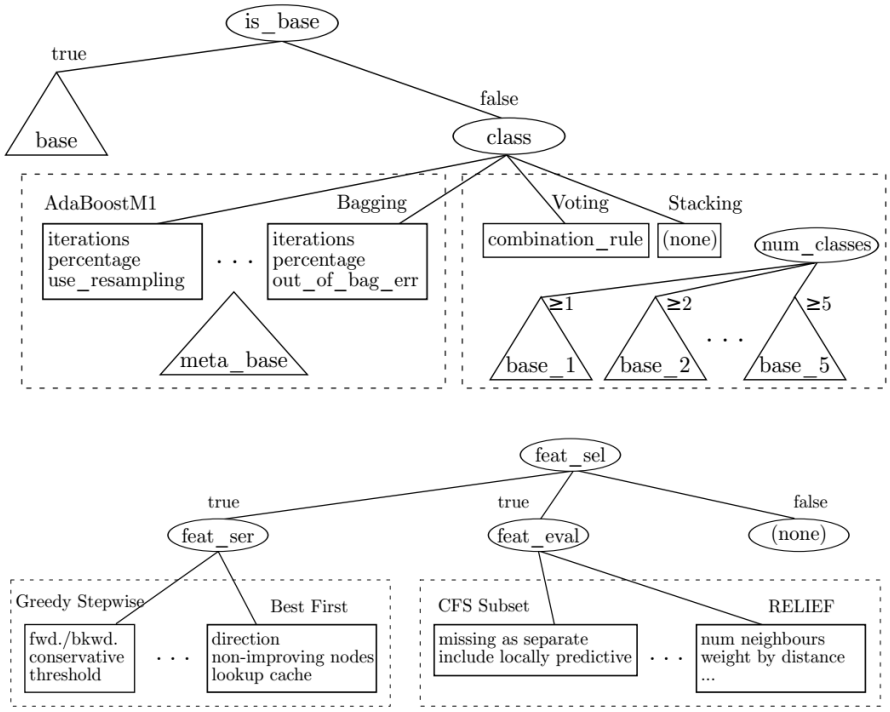



Figure 2.4: Auto-WEKA Parameter Space [5]

In general, as it is shown in Figure 2.4, Auto-WEKA has two top-level boolean parameters which are *is_base* and $feat\_sel$. The first boolean (*is_base*) is responsible for selecting a classifier. It chooses between a classifier and an ensembler or meta-classifier. In other words, if *is_base* set to True, it means the parameter *base* determines which of the 27 classifiers need to be selected. Otherwise (if *is_base* equal to False) then *class* indicates that either an ensembler or meta-classifier should be selected. In case *class* equal to meta-classifier, then a parameter is chosen among 27 classifiers. In ensembler scenario (when *class* equal to ensembler), one value from 1 to 5 assign to the *num_classes*, which is the number of the classifier in the ensembler. The second top boolean is $feat\_sel$, which is responsible for selecting the feature selection method. In case $feat\_sel$ receives False, it means Auto-WEKA does not choose any feature selection method. Otherwise, $feat\_ser$ and $feat\_eval$ choose the feature selection method and feature evaluator, respectively.

Auto-WEKA leverages SMAC and TPE, as two Bayesian Optimization algorithms for handling the hierarchical parameter spaces in different implementations. These implementations are available to the public under their website at British Colombia university. In total Auto-WEKA takes care of 786 parameters in 4 hierarchical layers. For further information the reader can refer to their original paper for more details [5].

### 2.2.2 Auto-sklearn

Auto-sklearn [4] is another AutoML system which takes advantage of Bayesian Optimization. This system improves the existing automated machine learning systems by considering the past performance on a similar dataset, which is known as warm-starting. Auto-sklearn constructs the ensemble from the model evaluated during optimization. This system won the first phase of the ChaLearn AutoML challenge [42], and their experiment over 100 datasets illustrates that Auto-sklearn significantly outperforms other AutoML systems [4].

Auto-sklearn covers 15 classification algorithms, 14 feature preprocessing, and 14 data preprocessing methods. This search space leads to 110 hyperparameters. In some sense, Auto-sklearn is an expansion of Auto-WEKA, extending the approach of using Bayesian Optimization to tackle the problem. Auto-sklearn also treats the AutoML as the CASH problem. It means Auto-sklearn formalizes the problem as combined algorithm selection and hyperparameter optimization to address two common and vital issues in AutoML.

The first issue is that there is not a single machine learning method that performs best on all datasets. Secondly, some ML methods specifically rely more on hyperparameter tuning than others like non-linear SVMs. In Auto-sklearn these two problems are efficiently tackled as a single optimization problem. Auto-sklearn fits a probabilistic model to track the relationship between hyperparameter setting and measured performance. Auto-sklearn then uses this probabilistic model to select the hyperparameters from the promising space. Auto-sklearn uses the mentioned probabilistic model to a trade-off

between exploitation in known suitable regions and exploration in an unknown part of the search space.

Figure 2.5 visualizes the overall workflow of Auto-sklearn system, including two-component Meta-learning and automatic ensemble creator. Auto-sklearn categorizes the machine learning pipeline in three steps (data preprocessing, feature preprocessing, and classifier).

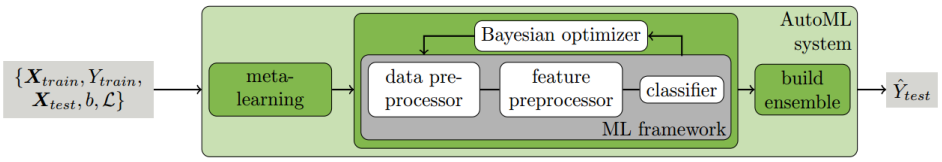



Figure 2.5: Auto-sklearn Workflow [4]

The success of the Auto-sklearn can be categorized in the following directions.

- **Meta-learning as the complementary step for Bayesian Optimization:**

  Domain experts in each area exploit the knowledge regarding the performance of each algorithm based on previous tasks. Meta-learning repeats the same behavior over datasets [43]. This means collecting the characteristics of datasets and monitoring the result of the ML algorithms, leading the system to determine which algorithm to use on a new dataset. Auto-sklearn uses Meta-learning by selecting the machine learning algorithms that are likely to work better on the given dataset. Furthermore, the characteristics of data are considered as a parameter for picking the algorithm for each dataset. In other words, whenever Auto-sklearn receives a new dataset, it firstly computes its Meta-features. It then ranks all the datasets based on their distance (in Meta-feature space) to the given dataset. Finally, it selects ML algorithm based k = 25 nearest datasets, and feeds these algorithms as the seed to Bayesian Optimization.

  More precisely, Auto-sklearn authors collect the number of datasets and calculate Meta-features (e.g., number of data points, features, number of classes, etc.) for each dataset. Then, they run ML algorithms and use Bayesian Optimization to determine promising ML algorithms for the given dataset.

- **Construct the ensembles automatically out of the evaluated models by Bayesian Optimization**

  Scientists combine the results of classifiers to achieve a better result compared to a stand-alone classifier. This technique, which is known as ensemble selection [44], is

also used on Auto-sklearn. In general, ensembles often work better than individual models. The ensemble works specifically well when firstly, each of the algorithms in the ensembler is strong itself and secondly, when each classifier produces an uncorrelated error in comparison with other methods in the ensembler.

Auto-sklearn paper shows that building a uniformly weighted ensemble [45] out of configuration that is suggested by Bayesian Optimization does not work well. As a result, they have employed different optimization approaches for choosing these weights, like *stacking*[46], *gradient-free numerical optimization*, and *method ensemble selection* [44]. It is proven that numerical optimizations are costly and non-efficient due to overfitting. Thus, Auto-sklearn authors' have ended up with an ensemble selection approach, which is suitable for their task. They built an ensemble with a size of 50 out of 50 best configurations found by Bayesian Optimization. In other words, they start from an empty association of ensemble and greedy add ensemble (classifier) that maximizes the validation performance with uniform weight [4].

| AutoML Systems | Language | Optimization | Meta-learning | Licence |
|---|---|---|---|---|
| Auto-WEKA | Java, CLI, GUI | Bayesian Optimization | - | GPLv3 |
| Auto-sklearn | Python | Bayesian Optimization | warm-start | 3-clause BSD |

Table 2.1: Comparison of AutoML Systems

Table 2.1 shows a comparison between Auto-sklearn and Auto-WEKA as popular tools in AutoML. Statistically, Auto-sklearn is significantly better than Auto-WEKA. Although there are cases in which Auto-sklearn loses against Auto-WEKA. The reason for this failure is that Auto-WEKA uses algorithms that the sklearn package does not support (e.g., the tree with a pruning component)[4].

### 2.2.3 Other AutoML Systems

This subsection briefly presents other AutoML frameworks that are capable of building ML pipelines [47].

TPOT [48] is an AutoML framework in python that is specially designed for classification and regression. TPOT is built on top of the scikit-learn package, it uses its regressor and classifier methods. Similar to the grid search, TPOT is not capable of handling continuous hyperparameters, and those hyperparameters need to be discretized beforehand [49]. TPOT favors genetic programming in constructing flexible pipelines as well as a different testing combination. To compensate overfitting, TPOT selects pipelines with low pipeline complexity and high prediction accuracy. TPOT is an open-source project and is under active development by the University of Pennsylvania. The reader can refer

to their documentation for more details [50].

Hyperopt-sklearn [51] is another AutoML framework, and as its name suggests, works on top of the scikit-learn package. Hyperopt-sklearn was designed for pipelines with exactly one preprocessor and one classifier (or regressor). Hyperopt is responsible for selecting and configuring the algorithms. Hyperopt-sklearn is only a wrapper over Hyperopt by defining a fixed pipeline shape and configuration space for each implemented algorithm. Cross-validation helps Hyperopt-sklearn to prevent or at least limit the effect of overfitting. Hyperopt-sklearn terminates the optimization after a fixed number of iterations as well as the time budget per evaluation.

## 2.3 Hyperopt

Model selection, hyperparameter tuning, and in general function optimization, are expensive tasks. Some techniques, like sequential model-based optimization (SMBO), are helpful in these scenarios. They are known for handling the parallel evaluation of expensive functions, leveraging smoothness without analytic gradient, and they can cope with hundreds of variables even with few hundreds of evaluation [52].

Hyperopt is capable of dealing with any SMBO problem. In this thesis, Hyperopt is described with the main focus on selecting the machine learning algorithm and tuning respective hyperparameters automatically in python. Hyperopt is responsible for finding the best value over the set of possible arguments. Hyperopt tries to find the best values in hard search settings, especially when there are a lot of hyperparameters and a small budget for function evaluation. Hyperopt encourages the user to provide more information about the search space and uses that knowledge to search more efficiently.

### 2.3.1 Hyperopt Setup

To begin using Hyperopt [53], a user needs to describe a function to minimize the cost function. Hyperopt, by default, tries to minimize the defined objective function. Hyperopt needs a search space as well as the algorithm to use for optimization. The following steps introduce each of these components.

Step 1: Defining an objective function
Hyperopt provides several levels of flexibility, when we want to define the objective function to minimize. An objective function is a python function that takes a single argument as input and returns a scalar value as output. The input argument stands for one or more objects which Hyperopt needs to receive, and scalar output stands for corresponding loss that needs to be minimized. Objective function calculates the cost based on the received argument.

Step 2: Defining a configuration space

Configuration space represents the domain boundaries that Hyperopt is allowed to search. Hyperopt asks the user to identify the search space for each hyperparameter. In each iteration of the search, Bayesian Optimization then chooses one value for each hyperparameter in the search space. The idea of search space is common between grid search and Bayesian Optimization. The search space in Bayesian Optimization is not discrete values. Instead, this space has a probability distribution for each hyperparameter.

It is essential to notice that *search space consist of nested function expression, including stochastic expression . The stochastic expressions are the hyperparameters and each of them has a label.* [53, 52]. This label is crucial for the time the selected value returned from the search space and for internal execution. The following part covers the important recognized stochastic expression by Hyperopt's optimization. For a full list, the reader can refer to Hyperopt's GitHub page [53].

- $hp.choice(label, options)$
  Hyperopt is capable of the nested stochastic expression with the help of *choice* expression. The element of *options* can be a simple list or a nested stochastic expression. In most of the cases, *choice* is used for a certain parameter or conditional expression. As a result, *choice* returns one of the options, which is list or tuple.

- $hp.randint(label, upper)$
  This stochastic represents the situation in the loss function where there is no correlation between integer values. This stochastic is an appropriate distribution for describing random hyperparameters (e.g., random seed). In the end *randint* returns integer in range $[0, upper)$.

- $hp.uniform(label, low, high)$
  This stochastic returns a float value uniformly between two boundaries of $low$ and $high$. This variable is constrained to a two-sided interval.

- $hp.quniform(label, low, high, q)$
  This stochastic returns a value similar to $round(uniform(low, high)/q) * q$ and is used whenever a discrete uniform value is needed. It is smooth but still bounded to $low$ and $high$.

- $hp.loguniform(label, low, high)$
  This stochastic returns a value drawn according to $exp(uniform(low, high))$ with the emphasis on the returned logarithm value being uniformly distributed.

- $hp.normal(label, mu, sigma)$
  This is an unconstrained variable and returns a real value which is normally-distributed with mean mu and standard deviation sigma.

In general, with defining the search space for Hyperopt, firstly, hard boundaries for each hyperparameter are determined and secondly, we give an idea to Hyperopt about

sampling strategies.

Step 3: Choosing a search algorithm
Currently, there are very few algorithms supported by Hyperopt. Tree Parzen Estimator and random search are the officially endorsed options, although the GitHub page [53] reports that a new method is being developed. There are also some external implications from other optimization strategies (e.g., Gaussian process-based [54]). However, Hyperopt currently does not support them as an algorithm.

It is essential to mention that hyperopt is capable of accepting trials. A trial in Hyperopt, by definition, is a point (configuration) that the algorithm uses before starting the search process. In this thesis, we used trials and history interchangeably.

## 2.4 OpenML

OpenML is an online Machine learning experimental database that allows the researchers to share their datasets as well as their efforts in machine learning experiments by details containing their code, results, models, predictions, evaluations, etc. The core concept of the OpenML relies on a single repository of dataset and results of ML experiments. It enables the scientists to work more efficiently and to learn from each other through collaboration [8].

As time goes by, ML is becoming increasingly popular among the people and practitioners who need to learn more about ML. Many of these people will walk the same path and will mostly encounter the same challenges. To prevent people from continuously reinventing the wheel, we need a strong community to help practitioners benefit from this collaboration and to speed them up on the overlapping parts of the road. As new folks in the machine learning community we can observe generous community members who share their experiments and results on the OpenML website. This collaboration enables people to take advantage of others' codes and to start from the place others have left off. Ultimately, OpenML empowers the community to engage in and hopefully to solve bigger and harder problems [55].

### 2.4.1 OpenML Components

OpenML consists of four components that will be covered in detail in the following subsections. Figure 2.6 illustrates the relation between these components.

#### 2.4.1.1 Dataset

Dataset is a collection of related data which is mostly in the form of a table and consists of instances which are the number of rows in the table. OpenML has more than 2000 active datasets for regression, classification, and stream processing. Each user is capable of uploading a new dataset. Whenever a dataset is uploaded, OpenML computes some Meta-features on the dataset (e.g., number of classes, number of missing values, etc.).

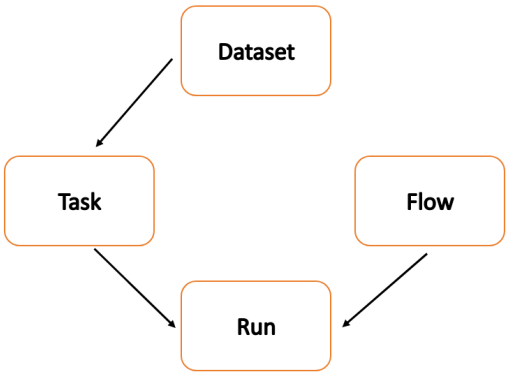



Figure 2.6: OpenML Components [55]

#### 2.4.1.2 Task

The task is a combination of a dataset and any applicable machine learning method, such as classification or clustering. Because the task can be supervised or unsupervised, in the data there exists a target column which specifies when the task is supervised. In more details, the task is always linked to a specific dataset (Figure 2.6) as well as the target variable (Run). Tasks also determine evaluation measures such as accuracy, precision, the area under the curve, etc.

#### 2.4.1.3 Flow

Flow stands for a kind of modeling that needs to be performed. It involves a series of steps, similar to the scikit-learn pipelines. It could represent an algorithm that solves a task, including the source code.

#### 2.4.1.4 Run

Run represents the application of flow on a specific task. In other words, pairs of flow and tasks, together result in a run. The run has the final prediction with specified hyperparameters, which are effectively captured by execution.

## 2.5 Summary

At the beginning of this chapter, we explained hyperparameter optimization and corresponding algorithms. Challenges in the grid and random search lead us to use Bayesian Optimization for HPO. Then, we discussed TPE as a promising algorithm in hyperparameter optimization and also as a backbone for Hyperopt. In the following, we reviewed the structure of the most recent and well-known AutoML tools. This chapter came to

an end by a review about hyperopt as a hyperparameter optimization tool and OpenML as an experimental database. Chapter 3 discuss different strategy for warm-starting Bayesian optimization and enables the AutoML to use an experimental database for designing the search space and guiding the search process.

# 3 HyperOpenML: Proposed Approach

In this chapter, we introduce HyperOpenML as a tool that enables AutoML to warm-start Bayesian Optimization with an experimental database to design the search space and guide the searching process for achieving a better result. This chapter starts with an overview (Section 3.1) and warm-start Bayesian Optimization (Section 3.2). It follows with an investigation on exploited pipelines form OpenML (Section 3.3) and different strategies to select history (Section 3.4). This chapter concludes with a summary (Section 3.5) at the end.

In this thesis, human or prior knowledge refers to exploited pipelines from OpenML that the Bayesian Optimization algorithm (TPE) uses before starting the search process (we consider these points as the starting point). These pipelines, also known as history or trials in the text.

## 3.1 Overview

HyperOpenML is the recommended tool that is empowered by Hyperopt and OpenML and it is the achievement of this thesis. The proposed tool after receiving a task, automatically exploits the correspond pipelines from OpenML, and with the help of Hyperopt, tries to solve the given task. The most challenging part in this path is choosing the algorithm and tuning respective hyperparameters. Warm-starting the optimizer with experimental database is studied to overcome the mentioned challenges. Moreover, to better reflect human knowledge, some hyperparameters from Hyperopt are modified ( e.g., linear forgetting and gamma). Furthermore, this thesis studies the characteristics of exploited pipelines from OpenML to better perform hyperparameter tuning. In addition, it cleans up the proposed history, to give the optimization only informative pipelines (flows) gained from prior knowledge, saving time and resources.

## 3.2 Warm-start Bayesian Optimization

Feeding human knowledge to Bayesian Optimization for achieving a better result is considered as warm-starting the Bayesian Optimization. Hyperopt is used for optimizing the process of selecting classifiers and tuning respective hyperparameters based on TPE. However, some modifications are needed to reflect better the characteristics of human knowledge for solving a given task. These modifications are also known as Hyperopt's

hyperparameter tuning or hyper-hyperparameter tuning. The first two subsections introduce linear forgetting (Section 3.2.1) and gamma (Section 3.2.2) as two hyperparameters for hyperopt. This section covers the history size (Section 3.2.3) as another significant character in warm-staring.

### 3.2.1 Linear Forgetting

According to Hyperopt's paper [56], *Hyperopt gave the full weight to the most recent 25 trials and applied a linear ramp from 0 to 1.0 to older trials* which authors call this strategy, linear forgetting. When we are feeding the history with human knowledge, we need to ask Hyperopt to give all trials a full weight and consider all of them equally for proposing a new set of hyperparameters. In other words, Hyperopt gives more attention to the last 25 trials that are available in the history. A significant number is assigned to linear forgetting in order to compensate against discrimination between points in history.

### 3.2.2 Gamma

As mentioned in Section 2.3, Hyperopt, after sorting the parameters based on their loss and applying the kernel density estimator as well as stochastic expression, names the top-performing distribution $l(x)$ and the rest $g(x)$. Generally, Hyperopt out of $T$ observations of any given variable, considers $\sqrt{T}/4$ top-performing trials to estimate the density of $l$, and this fraction is determined by gamma. To better reflect human knowledge in the trials, this fraction is tried with different and diverse numbers [56].

### 3.2.3 History Size

Regardless of the trial's effect on the final result, the size of the proposed history has a direct influence on the required time for TPE to offer a new set of hyperparameters. It means the run time for each iteration of the TPE's algorithm linearly scale with the size of history.

In the same way, TPE required time in each iteration also increases linearly with the number of the dimensions (hyperparameters) being optimized [19]. However, in this thesis, the number of hyperparameters is considered the same for all experiments.

## 3.3 Human Knowledge Analysis

This section explains the characteristics of human knowledge that have been exploited by HyperOpenML. In the first subsection, available features for most of the runs in OpenML are covered (Section 3.3.1), then pipelines (flows) are investigated in more detail (Section 3.3.2).

### 3.3.1 Runs Features

In the following, all features that have been exploited for each pipeline (run id in OpenML) are categorized as follows:

- **Identity Features**: For each given pipeline in OpenML some specific indexing is available.
  - **Run id**: Specify run corresponding to run_id in OpenML
  - **Flow id**: Specify flow corresponding to flow_id in OpenML
- **Metric Features**: Evaluation metrics are as follows:
  - **Accuracy**: Accuracy classification score.
  - **F1-score**: The harmonic mean of precision and recall.
  - **Area under ROC curve**: *Area under a graphical plot that illustrates the diagnostic ability of a binary classifier system as its discrimination threshold is varied* [57]
  - **Kappa**: *Considered an overly conservative measure of agreement*[58].
  - **Kb relative information score**: *Information-based evaluation criterion for the performance of the classifier* [59]
  - **Mean absolute error**: *Measures how close the predictions of the model are to the actual target values* [60]
  - **Mean prior absolute error (MPAE)**: *The mean absolute error of the prior (e.g., default class prediction)* [61]
  - **Prior entropy**: *Entropy, in bits, of the prior class distribution. Calculated by taking the sum of -log2(priorProb) over all instances, where priorProb is the prior probability of the actual class for that instance* [62]
  - **Relative absolute error (RAE)**: *Mean absolute error (MAE) divided by the mean prior absolute error (MPAE)* [63]
  - **Root mean prior squared error (RMPSE)**: *The Root Mean Squared Error (RMSE) of the prior (e.g., the default class prediction)* [64]
  - **Root mean squared error (RMSE)**: *Measures how close the predictions of the models are to the actual target values*
  - **Root relative squared error (RRSE)**: *The Root Mean Squared Error (RMSE) divided by the Root Mean Prior Squared Error (RMPSE)* [65].
- **Data preprocessing**: There are six options for this step.
  - **Normalizer**
  - **Column Transformer**

  - **Simple Imputer**
  - **Standard Scaler**
  - **Minmax Scaler**
  - **Do nothing**

- **Feature preprocessing**: There are four options for this step. In the following each option is presented by its hyperparameters.
  - **PCA**: Iterated power, Number of components, Svd solver,Tol, Whiten
  - **Kernel PCA**: Kernel, Number of components
  - **Variance Threshold**: Threshold
  - **Do nothing**

- **Classifiers**: There are ten options for this step. In the following, each option is presented by its hyperparameters.
  - **Randomforest classifier**: Criterion, Max depth, Min samples leaf, Min samples split, Min weight fraction leaf, Max features, Number of estimators, Oob score
  - **Decisiontree classifier**: Criterion, Max depth, Min samples leaf, Min samples split
  - **Gradientboosting classifier**: Criterion, Learning rate, Max depth, Max features, Min impurity decrease, Min samples leaf, Min samples split, Min weight fraction leaf, Number of estimators, Number of iteration no change, Subsample, Tol, Validation fraction
  - **Kneighbors classifier**: Number of neighbors, Algorithm
  - **Extratrees classifier**: Bootstrap, Criterion, Max features, Min samples leaf, Min samples split,
  - **MLP classifier**: Activation, Alpha, Batch size, Beta 1, Beta 2, Early stopping, Hidden layer sizes, Learning rate, Learning rate init, Max iteration, Momentum, Number of iteration no change, Nesterovs momentum, Power t, Shuffle, Solver, Tol
  - **SGD classifier**: Loss, Penalty, Alpha, Max iteration, Tol
  - **Bernoullinb**: Fit prior, Alpha
  - **Fkceigenpro**: Degree, Gamma, Kernel, Number components
  - **SVC**: C, Coef0, Degree, Gamma, Kernel, Shrinking, Tol

### 3.3.2 Dataset Characteristics

This subsection represents different types of flows with various ranges of accuracy. It also reports the availability of each flow for each dataset. Understandably, the difference

in accuracy for multiple runs in the same flow is the effect of hyperparameter tuning. This fact shows the importance of hyperparameter tuning. This information gives us insight into the search space as well as the best and worst candidate flow for each dataset. Moreover, exploring this information enables TPE to choose the configuration wisely. We have shown this information for each dataset by two graphs. First shows the effect of hyperparameter tuning in each flow, and the second one illustrates the available number of runs for each flow.

We used four favorite datasets in OpenML for our experiments. These datasets have a considerable number of runs in OpenML, so we chose these datasets for our experiments. We show exploited pipelines for each dataset in three steps of Data preprocessing, Feature preprocessing, and Classifier (DFC). It is clear that the first two steps are optional, and 'do_nothing' represents the absence of each of these steps in the ML pipeline.

- **Dataset 3**
  This dataset is about chess games and is titled as *Chess End Game*. It was initially generated and described by Alan Shapiro at UCI repository in 1989 [66]. This dataset has 37 features in total that are board-descriptions for the chess game. The first 36 attributes describe the board, and the last one indicates the winner of the game. Moreover, this dataset has 3196 instances, and the classes' distribution is 52% (White can win) against 48% (White cannot win).

  Figure 3.1 displays the exploited pipelines for Dataset 3. As shown, most of the pipelines could come up with high accuracy (e.g., more than 0.8). At the same time, some pipelines provide diverse and different results with a different configuration. In the exploited history for this dataset, nine pipelines provide the best available accuracy. Although these nine pipelines use the same classifier (SVC) with different hyperparameters, there are differences in data preprocessing and feature preprocessing steps. The fact that different pipelines with different hyperparameters provide the same final accuracy lets us know more about complex search space. Such information might be a facilitator for warm-starting the Bayesian Optimization. Choosing gradient boosting as a classifier for this dataset provides a different range of accuracy. Figure 3.2 represents the number of available sample for each pipeline.

- **Dataset 31**

  This dataset is about German credit data, is titled as *Credit-g* at UCI repository and was published in 1994 [66]. In more details, this dataset classifies people based on their characteristics to good and bad in terms of credit risk. This dataset has 21 features (including the target), and the classes' distribution is imbalanced, 70% (good) against 30% (bad). Moreover, this dataset has 1000 instances.

  Figure 3.3 depicts that each flow provides a range of accuracy with a different set

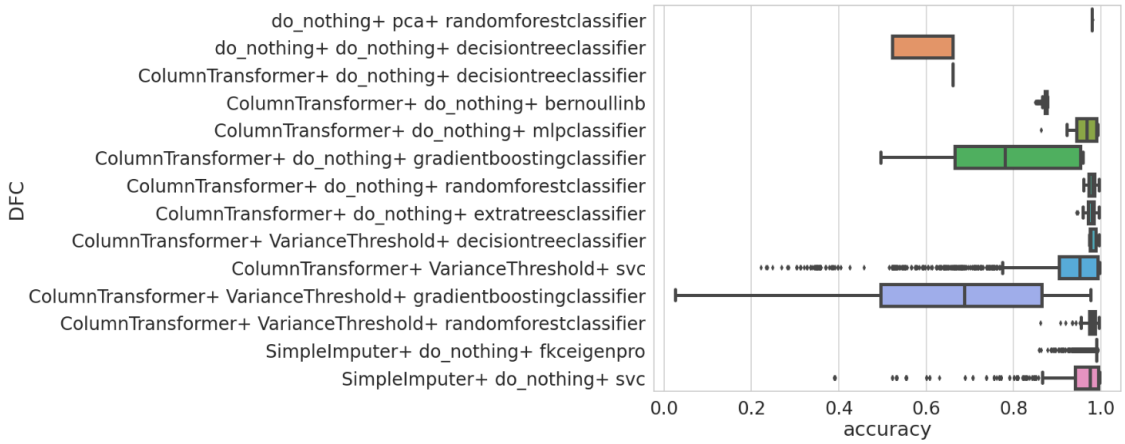



Figure 3.1: The variation range of accuracy for exploited pipelines from OpenML - Dataset 3

of hyperparameters for each run. For example, in the given Figure 3.3, column transformer, variance threshold, and on top gradient-boosting as data preprocessor, feature preprocessor, and classifier, are respectively shown. This pipeline is capable of providing accuracy from under 0.3 to more than 0.7, due to hyperparameter tuning. Moreover, Figure 3.4 shows the availability of pipelines for this dataset is imbalanced. While the combination of PCA and randomforest have worked well for this dataset, there are few available pipelines from this type in our collected history. For some specific flows, the available samples are limited (e.g., one sample), and this is one of the challenges regarding collected data from OpenML. As a result, this imbalanced knowledge about search space can mislead the Bayesian Optimization at some points.

- **Dataset 32**
  This dataset is about digits and is titled *Pen-Based Recognition of Handwritten Digits* [66] at UCI repository, published in 1998. This dataset was collected by 250 samples from 44 writers. There are 17 features (including the target), and 10992 instances in this dataset. The class feature indicates one of the numbers between 0 to 9, based on 16 available features.

  Figure 3.5 shows the exploited pipelines for Dataset 32. As shown, the number of tried flows is less that than datasets 31 and 3. Furthermore, there is a limited number of the run for the pipeline (PCA - Randomforest). In the exploited history, a diverse range of accuracy is available for some flows. For example, simple-imputer as a data preprocessor and SVC as classifier could provide accuracy less than 0.1 to more than 0.9, so it would be complicated to find the right set of hyperparameters. Moreover, as shown, a pipeline with a single component (only classifier, e.g., decision-tree) is too simple and would not lead us to any good performing

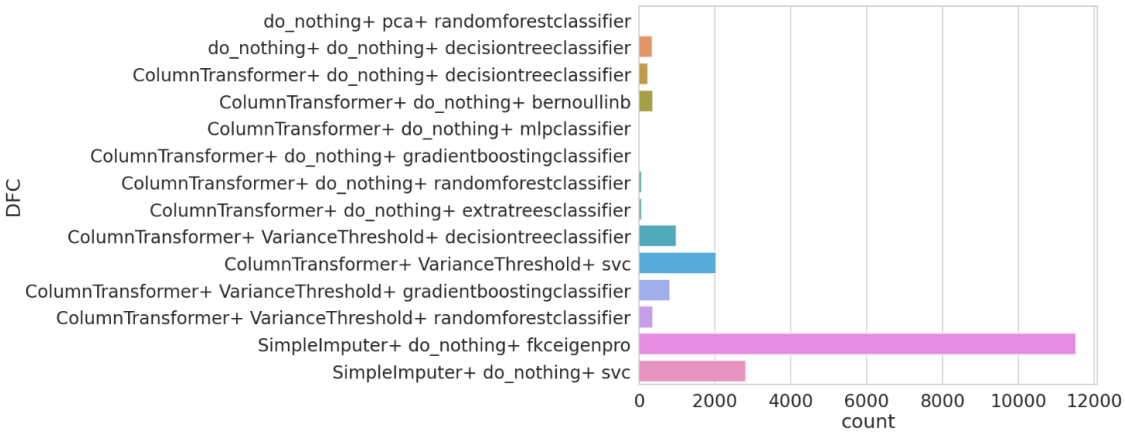



Figure 3.2: Count Plot out of exploited pipelines from OpenML - Dataset 3

results. There are nine flows with the same classifier (SVC, simple imputer or column transformer) that end-up to the best available result on OpenML. As shown in Figure 3.6, in contrast to other datasets, the best performing pipeline for this dataset has the most significant available size on OpenML.

- **Dataset 40509**
  This dataset is about the Australian credit card and is titled *Australian Credit Approval* [66] at UCI repository, published in 1987. It originates from the StatLog project, and concerns credit card applications with 15 features (including the target features), and it has 690 instances. Features cover the applicant's characteristics, and the class target includes the decision yes (55.5%) or no (44.5%).

  Figure 3.7 shows the exploited pipelines for dataset 40509, similar to Dataset 32; the available number of flows is less than other datasets. As Figure 3.8 shows, the available runs for specific pipelines are limited (PCA - Random-forest and column transformer - gradient-boosting). Moreover, for this dataset, three flows end up to the best available accuracy. While all best pipelines have the same data preprocessing and feature prepossessing (Column transformer and Variance threshold), they have different classifiers (random forest and gradient boosting classifier). In contrast to other datasets, SVC for this dataset is hard to tune because it can provide accuracy less than 0.2, to more than 0.8.

The knowledge which has been exploited for each dataset is an imbalance, and it means we have more insight into some parts of the search space, while some parts are still undiscovered. Even in discovered parts, the number of samples is not equal, and this lack of balance could cause complexity in the space. Furthermore, too many pipelines for one flow could motivate Bayesian Optimization to search that part of the space deeply. As a result, and due to the limited number of iterations, it could head to local optima.

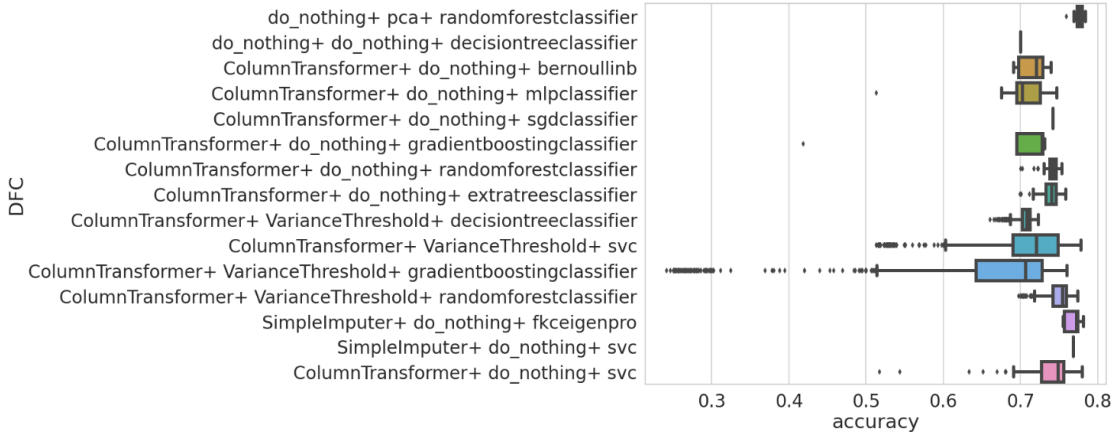



Figure 3.3: The variation range of accuracy for exploited pipelines from OpenML - Dataset 31

Moreover, if we investigate the final accuracy, there are many pipelines that end up with the same result. Table 3.1 shows the unique accuracies for each dataset. In more details, this table reports how many pipelines after hyperparameter tuning reach to the same accuracy. For example, in Dataset 3, there are only 1118 unique accuracies and the rest of configurations (till 19666, total history size) after execution end up to one of those 1118 accuracies.

| Data set | History size | unique accuracy pipelines | total history size |
|---|---|---|---|
| 3 | 19666 | 1118 | 19666 |
| 31 | 6660 | 294 | 6660 |
| 32 | 8052 | 1965 | 8052 |
| 40509 | 4857 | 275 | 4857 |

Table 3.1: Accuracy Investigation - Determine the unique number of pipelines in terms of accuracy for each dataset

In this section we showed the diversity of pipelines as well as the direct influence of hyperparameter tuning on the final result. For some pipelines, tuning the hyperparameters is challenging, and different configurations could result in different results. For example, the Gradient Boosting classifier in Dataset 3 and 31 is hard to tune. Since it produces the final accuracy of less than 0.2 until more than 0.7, it is very the same for SVC in Dataset 32 and 40509. Our exploration showed SVC pipelines are more prevalent between users in OpenML, so users have used this classifier more than others. As we have seen, the availability of some pipelines is limited, and this imbalanced information about the search space could (might) mislead Bayesian Optimization after warm-starting.

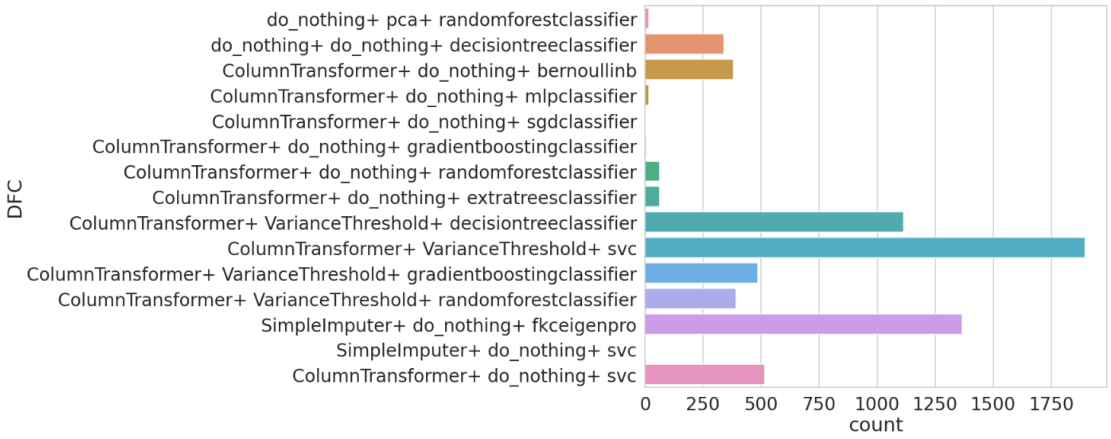



Figure 3.4: Count Plot out of exploited pipelines from OpenML - Dataset 31

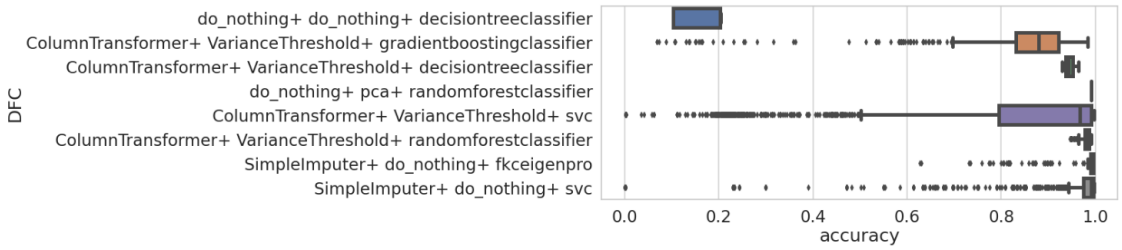



Figure 3.5: The variation range of accuracy for exploited pipelines from OpenML - Dataset 32

## 3.4 History Selection Strategies

As mentioned earlier (Section 3.2.3), the size of the given trials is one factor that directly impacts on the TPE run time. Moreover, a different subset of history can provide different results. Then a selection criterion is needed. This section investigates various procedures for selecting a subset of available trials. In more details, it starts with full history (Section 3.4.1) and accuracy threshold (Section 3.4.2). Then we sort the exploited pipelines and select largest and smallest accuracies pipelines for warm-starting (Section 3.4.3). Histogram-based (Section 3.4.4) and cluster-based (3.4.5) are other approaches for subset selection. Unique accuracy (Section 3.4.6) is the last approach that we cover in this section.

### 3.4.1 Full History

All trials that have been exploited for each dataset feed as prior knowledge.

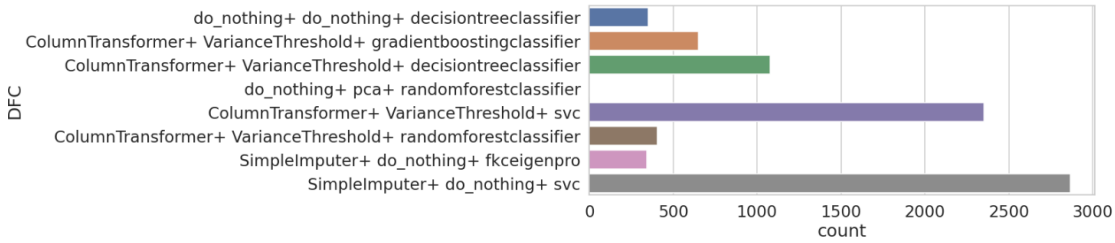



Figure 3.6: Count Plot out of exploited pipelines from OpenML - Dataset 32

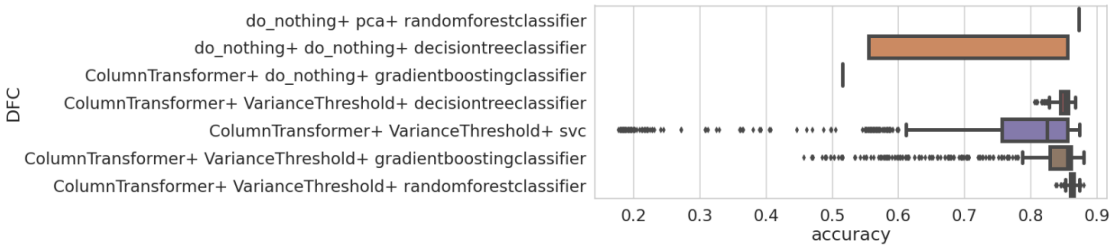



Figure 3.7: The variation range of accuracy for exploited pipelines from OpenML - Dataset 40509

### 3.4.2 Accuracy Threshold

This section focuses more on the quality of the exploited pipelines (trials). Trials are filtered based on their final accuracy. This approach only allows pipelines that meet 50 percent or higher accuracy to be fed to Hyperopt.

### 3.4.3 Largest and Smallest Accuracies

In this approach, exploited pipelines are sorted based on their final accuracy. In the first scenario, only N best pipelines are fed into Hyperopt as prior knowledge. In the second scenario N worst pipelines are selected from the sorted pipelines list.

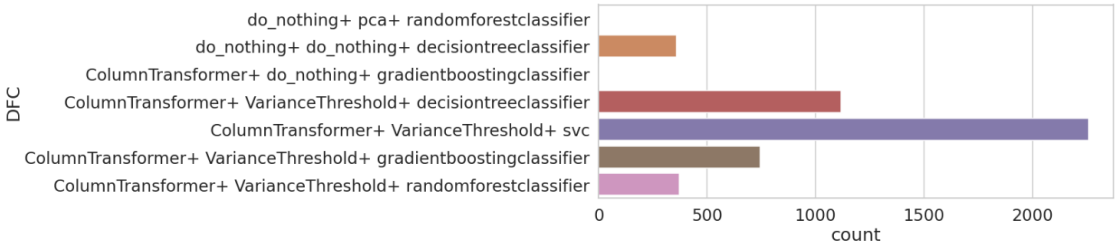



Figure 3.8: Count Plot out of exploited pipelines from OpenML for Dataset 40509

### 3.4.4 Histogram-based History

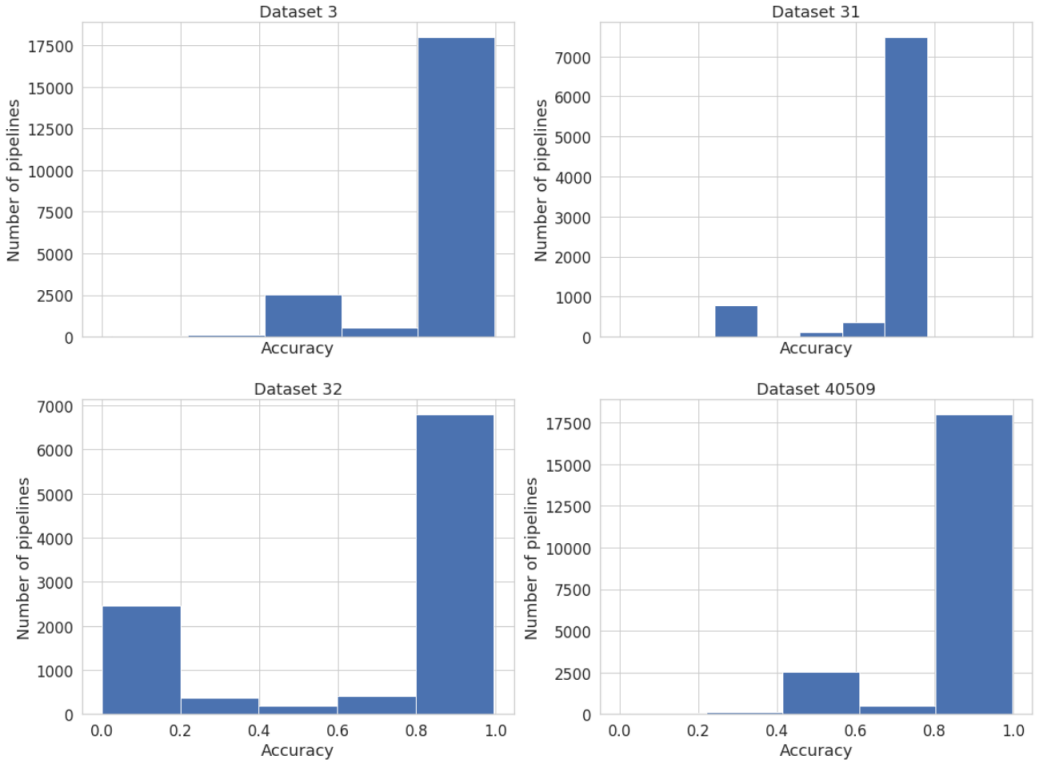



Figure 3.9: Histogram plot for exploited pipelines based on their final accuracy

Figure 3.9 illustrates the distribution of the final accuracy of our four datasets. In this approach, because accuracy distribution is imbalanced, we fix the number of bins in a histogram to five. To give all bins the same chance, we select an equal percentage from all bins for warm-starting.

### 3.4.5 Cluster-based History

In this approach, each pipeline is turned into a vector, and with the help of a clustering algorithm (K-Means [67]), these vectors are grouped into different clusters. Similar to the histogram approach and to give all clusters the same chance, N points are then selected, accumulated, and fed to Hyperopt as prior knowledge. The first subsection covers feature engineering (3.4.5.1). The second subsection discusses different strategies for finding the number of clusters for the K-Means algorithm.

#### 3.4.5.1 Clustering Input Representation

For each exploited pipeline, there are different types of features. For encoding the categorical features, one hot encoding is used, as well as mean imputation for the numerical

features. Scenarios for calculating the best number of K in K-Means are provided here:

- All features: All features are considered for finding the best K in K-Means (including the metric features, data preprocessing, feature preprocessing).
- Only classifiers' features: Features which are related to classifier's hyperparameters are considered for finding the required K.
- Only evaluation features: Metric feature are considered for finding the best classifier.
- Dimensional reduction: Features are reduced by PCA and then the number of the K is determined.
- Selective features: Accuracy, F1-score, data preprocessing, and feature prepossessing are only considered for finding the number of clusters.

#### 3.4.5.2 Finding the Best K

Determining the K in the K-Means algorithm can be considered as the most important hyperparameter in this algorithm, and changing this parameter can change the whole result. To determine the best number of clusters in K-Means, the following approaches are used:

- Elbow Method [68]:
  This method relies on the sum of squared error (SSE). This value defines a sum of the squared distance between the centroid and each member in the cluster. Plotting K against SSE gives us the Figure which known as elbow graph. In this graph, increasing the K case the decrease within-cluster SSE (distortion). This is because the samples will be closer to their assigned centroid [69].
- Silhouette coefficient [70]:
  This method is a tool to assert the cluster validity and help users to find the optimal number of cluster. Silhouette is a measure of how similar an object is to its cluster rather than the neighbor's clusters.

#### 3.4.5.3 Feeding Clusters

In this part, different approaches of feeding clustering results are investigated.

- Feed the biggest cluster: Biggest cluster is fed as a trial to Hyperopt.
- Feed the most distinct cluster: Compute the sum of squared error for each cluster, and feed the cluster with the biggest SSE.
- Sample from clusters: Sample from each cluster with an equal sample size (sample size is equal to number of points in smallest cluster) for all clusters. In the selection process, the nearest points to centroid would be selected for each cluster.

### 3.4.6 Unique Accuracy

In this approach, exploited flows are sorted based on their final accuracy. As mentioned in Table 3.1, many pipelines reach the same final accuracy at the end. In this approach, pipelines with unique accuracies are fed as prior knowledge.

## 3.5 Summary

In this chapter, we introduced HyperOpenML as a tool that enables AutoML to use an experimental database to design the search space and guide the search process. We discussed the characteristic of exploited pipelines for each dataset and different strategies for selecting history in warm-starting Bayesian Optimization. Chapter 4 covers the implementation details of HyperOpenML.

# 4 Implementation Details

This chapter describes the implementation of the components in HyperOpenML (Hyperopt + OpenML) that uses experimental databases to help TPE to suggest better pipelines for automatic machine learning. This chapter explains each step after HyperOpenML interface, shown in Figure 4.1 in more detail, it begins with three steps for building history from OpenML and follows with Hyperopt adoptions.

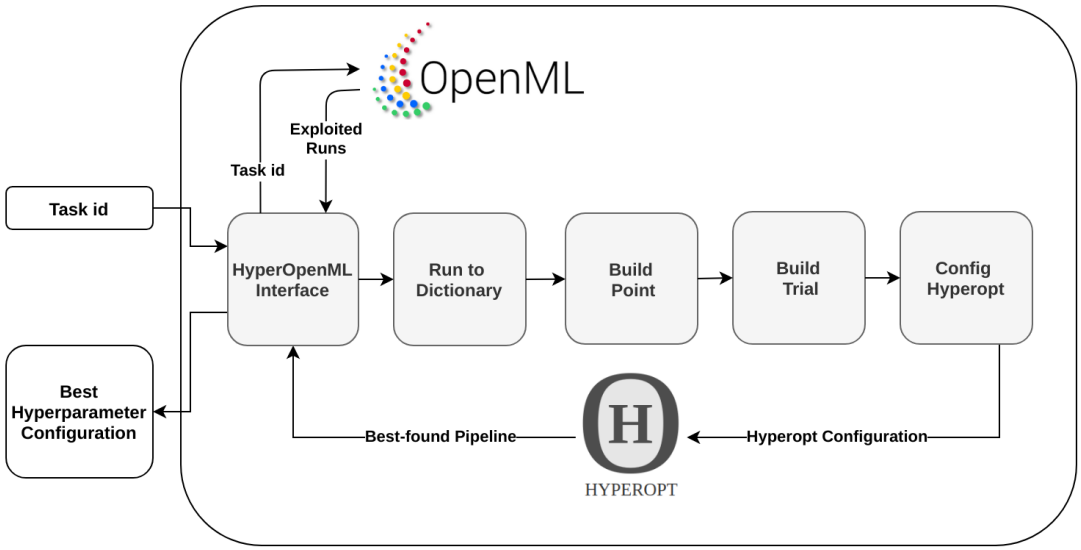



Figure 4.1: HyperOpenML Work Flow

## 4.1 History Building

We consider all flows with more than 100 runs on OpenML as a candidate to use its run as prior knowledge. This approach allows us to focus on more important flows, and respectively, we have downloaded the corresponding runs for each flow. In the following subsections, three components of this module are explained in more details.

### 4.1.1 Run to Dictionary

OpenML API's enable us to collect runs based on specific task id and flows. This API helps to collect as many runs that exist under a specified task id. As we followed three steps (data preprocessor, feature preprocessor, and classifier) for machine learning pipelines such as auto-sklearn [4], we have filtered the complex pipelines with more than three steps. As the last part of this step, the list of dictionaries will be passed to the next step (each dictionary represents one run in the OpenML).

### 4.1.2 Point Building

In this step, for each key in the transmitted dictionary (Hyperparameter in each run in OpenML), the range of values are checked by Hyperopt's search space. This checking filters the runs which are out of search space boundaries and allows us to bring runs that are part of Hyperopt's search space.

In this step, the evaluation parameters (e.g., accuracy), available on OpenML, are added to the prepared points. Bringing this information empowers Hyperopt to save a lot of time because it prevents Hyperopt from examining those configurations under acquisition function.

### 4.1.3 Trial Building

In this step, the trials are built based on the structure of Hyperopt. It means that the prepared points in the last step save in the way that Hyperopt can read them as history, preventing TPE from trying those configurations and generating better pipelines.

### 4.1.4 Trial Preparation

After the aforementioned steps, we managed to bring different runs for each task. Table 4.1 represents a summary of the number of runs that are exploited for the rest of the experiments.

Since all pipelines in the OpenML are designed and uploaded by humans, they are incomplete in some terms. For example, some pipelines are missing some evaluation scores like F1-score. Since F1-score is one of the critical evaluation metric [71, 72], we have to exploit pipelines in two scenarios without and with F1.

| Task number | Number of history with F1 | Number of history without F1 |
|---|---|---|
| 3 | 19666 | 21249 |
| 31 | 6660 | 8743 |
| 32 | 8052 | 10203 |
| 125923 | 4857 | 6803 |

Table 4.1: Number of exploited pipelines for each dataset based on OpenML

For simplicity, in the rest of the experiment history without and with F1 are called *history_without_F1* and *history_with_F1*, respectively.

## 4.2 Config Hyperopt

As Section 2.3 also mentioned, Hyperopt needs objective function and search space as a requirement. Therefore, the subsections cover those requirements.

### 4.2.1 Search Space

This step consists of three steps: data preprocessing, feature preprocessing, and classifier.

This search space is built based on flows that are available on OpenML, and a range of hyperparameters is defined based on 50763 runs, which are considered to build this space. There are 6 options for data preprocessing, 4 alternatives for feature preprocessing, and 10 candidates for classifier's step. The range of each hyperparameter belongs to one of these three-step sets, with diversity of that hyperparameter in the 50763 downloaded runs. For more details please check Table 4.2.

For example, the minimum and maximum number of an estimator (as one hyperparameter) for randomforest (as one classifier) in search space is defined based on downloaded flows from OpenML. However, due to human knowledge, some range is extended for more exploration.

In general, HyperOpenML tries to tune 79 hyperparameters. They include a combination of choosing an item for each step, and tuning the hyperparameters for a selected algorithm.

### 4.2.2 Objective Function

In our scenario, the objective function that Hyperopt tries to minimize is negative (minus) accuracy (in general, Hyperopt is designed for minimization task, because we need to maximize the accuracy we have used -accuracy). Strictly speaking, Hyperopt tries to improve the average accuracy in each iteration after ten-fold cross-validation. In more details, Hyperopt searches for pipelines (data preprocessing, feature preprocessing, and classifier) based on TPE that provides higher average accuracy after ten-fold cross-validation.

## 4.3 Scenario and Assumptions

Figure 4.1 represents that HyperOpenML starts its job by receiving a task id in its interface. HyperOpenML then connects to OpenML and downloads corresponding runs for

| Step | Options | Number of hyperparameter(s) |
|---|---|---|
| Data Preprocessing | Normalizer | |
| | Simple Imputer | |
| | Column Transformer | 1 |
| | Standard Scaler | |
| | Minmax Scaler | |
| | Do nothing | |
| Feature Preprocessing | PCA | 5 |
| | Kernel PCA | 2 |
| | Variance Threshold | 1 |
| | Do nothing | |
| Classifier | Randomforest classifier | 8 |
| | Decisiontree classifier | 4 |
| | Gradientboosting classifier | 13 |
| | Kneighbors classifier | 2 |
| | Extratrees classifier | 5 |
| | MLP classifier | 17 |
| | SGD classifier | 5 |
| | Bernoullinb | 2 |
| | Fkceigenpro | 4 |
| | SVC | 7 |

Table 4.2: Number of hyperparameters for every step in the pipeline

the given task id, then transfers and adopts downloaded-runs as a trial (history) to Hyperopt. In this step, TPE in Hyperopt starts to find a better pipeline based on pipelines that have been exploited from OpenML. In the worst case-scenario with a defined budget, if HyperOpenML can not find a better pipeline, it will retrieve from OpenML the best pipeline that exists for a given task. In the best-case scenario, HyperOpenML can outperform practitioners on OpenML and can find a better pipeline that reports to the user in the given budget.

The main focus of this thesis is on pipelines that support the scikitlearn's package and consist of three steps. However, as the designed search space is high dimensional and complex, HyperOpenML needs a huge iteration budget for achieving a better result, which was not applicable in this thesis due to time and resource limitation. For this reason the average accuracy is also reported, as well as the best-found item in the generated pipelines at the evaluation section.

## 4.4 Summary

This chapter reported the implementation detail about HyperOpenML. It started with steps to exploit the experimental database, followed by the required step for Hyperopt adoption. Chapter 5 illustrates the experiment results that is empowered from implementation in this chapter.

# 5 Evaluation

In this chapter, we evaluate the effectiveness of automatically using an experimental database to improve Bayesian Optimization capabilities. To show the efficiency, we measure improvement in the quality (accuracy) of the best-found pipeline and the average quality of proposed pipelines by TPE. Additionally, history selection in HyperOpenML tries to minimize the TPE run time. It is essential to mention, due to complexity in tasks and search space [73] in ML problems, even small improvements are considered in production or ML competitions [74, 75]. Because in reality, for large companies, even minor improvements in the ML models means millions of dollars [76].

This chapter starts with showing the influence of history size on TPE's run time (Section 5.1). The second subsection follows with different approaches for improving the quality of proposed pipelines. In more details, the HyperOpenML (considering the full history - all pipelines downloaded from OpenML) compares with vanilla Hyperopt to check the influence of the experimental database on the quality of proposed pipelines (Section 5.2). In the third subsection, the importance of all parts of the proposed history is shown by removing the low-quality pipelines (Section 5.3). This experiment is designed to show the importance of low-quality pipelines in history. Furthermore, to confirm the required diversity in the proposed history, top k best and worst pipelines are selected from the history and fed to the TPE algorithm (Section 5.4). In the following sections, to give an equal chance to different accuracy range to be part of the history, uniform sampling from each bin in the histogram is studied (Section 5.5). Besides, the clustering approach is used for fair sampling from the experimental database (Section 5.6). This chapter concludes with hyper-hyperparameter tuning to better reflect the human knowledge characteristic for the best approach for each dataset (Section 5.8).

It is important to note that all of the experiments reported in this thesis are the average of three times execution of experiments (100 iterations in each experiment). It is essential to mention that due to the complexity of search space, a higher number of iterations is required. We limit this number to 100 due to time and resource limitation.

## 5.1 TPE Run-time

TPE runtime is directly influenced by increasing the size of the historical trials. In more details, in each iteration, increasing the size of the proposed history linearly increases the TPE algorithm's runtime. Figure 5.1 illustrates the TPE overhead increases when the number of trials increases. In this figure, we removed the worm up iteration (first it-

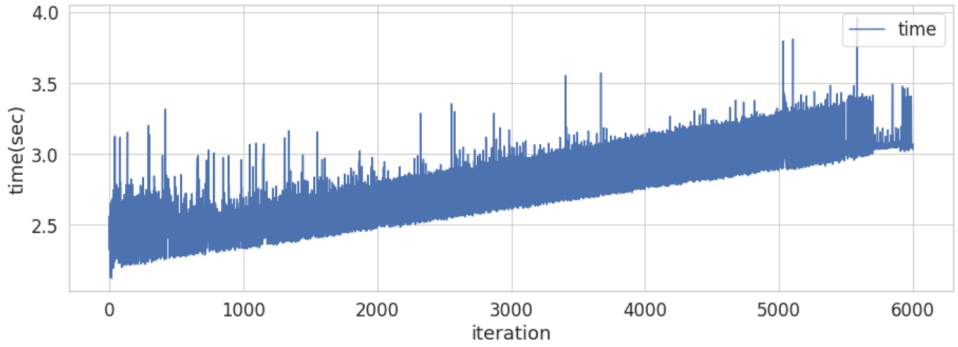



Figure 5.1: TPE Overhead in Each Iteration

eration) and runtime for each configuration to show precisely the overhead time on TPE when the algorithm wants to propose a new configuration. In this experiment, we start from a null history and set the iteration size to 6000. For offering a new configuration in iteration T, the TPE algorithm needs to consider T-1 configurations in history. As a result, with increasing the size of history, the TPE overhead for configuration suggestion rises linearly. This linear increment motivates us to study the strategies to limit the number of trials (due to the runtime) and, meanwhile, control the quality improvement.

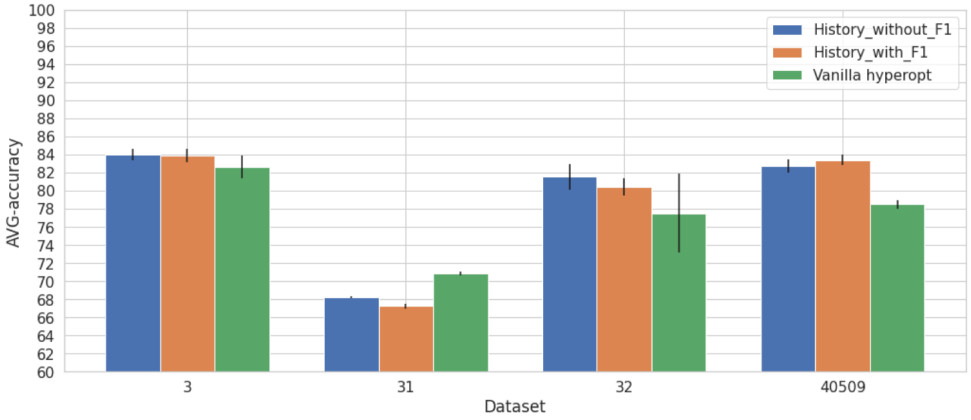
The following sections cover the impact of the mentioned strategies (Section 3.4) on the quality of the proposed pipelines.

## 5.2 Full History

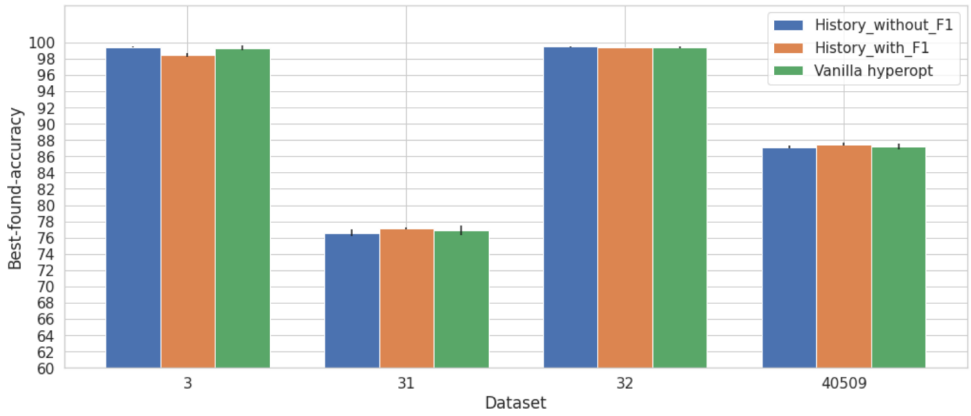
In this approach, we fed the experimental database (presented in Chapter 3), as prior knowledge (Trials) to Hyperopt. Due to importance of F1-score [71, 72] we have exploited the pipelines in two scenarios. As we noted in Table 4.1, *history_without_F1* represents the pipeline without F1 score, and *history_with_F1* represents pipelines with F1-score.

Figure 5.2 represents the comparison between the vanilla Hyperopt (without any historical trials) and HyperOpenML (*history_without_F1*, *history_with_F1*). It is essential to mention we reported the results after only 100 iterations, due to time and resource criteria in this thesis. Thus, in most datasets (three out of four), prior knowledge can help the HyperOpenML to reach a better accuracy in the average of one hundred iterations. Although the competition between the vanilla Hyperopt and HyperOpenML in finding the best-found pipeline for each task is extremely close, HyperOpenML defeats the vanilla version in all four datasets and gets the (slightly) better results.

Table 5.1 shows the results in more detail.




(a) Average Accuracy




(b) Best Found

Figure 5.2: Comparison between Vanilla Hyperopt and HyperOpenML (History_with_F1, History_without_F1) - This figure compares the average accuracy and best-found pipeline after 100 iterations for each dataset. The error bar in each chart indicates the level of variation in the results.

## 5.3 Accuracy Threshold

Table 5.2 shows the result after removing the low-quality pipelines from experimental database for each task. We consider the pipelines with the final accuracy under 50%, as low-quality pipelines due to binary classification in some of our tasks. The number of removed pipelines is shown in the third column of this table for each dataset. In most cases (3 out of 4, after 100 iterations), removing low-quality pipelines caused the best-found and average pipelines to drop. This experiment illustrates the potential helpfulness of low-quality pipelines. We see that including low-quality pipelines - not just the best, or high-quality pipelines - in most cases, amounts to a better result. Low-

| Data set | Approach name | Number of trials | Average accuracy | Best Accuracy |
|---|---|---|---|---|
| 3 | Vanilla Hyperopt | 0 | 82.66±1.24 | 99.31±0.28 |
| 3 | History_with_F1 | 19666 | 83.88±0.70 | 98.46±0.27 |
| 3 | History_without_F1 | 21249 | 84±0.68 | **99.44±0.44** |
| 31 | Vanilla Hyperopt | 0 | 70.88±0.20 | 76.90±0.57 |
| 31 | History_with_F1 | 6660 | 67.24±0.23 | **77.13±0.12** |
| 31 | History_without_F1 | 8743 | 68.25±0.09 | 76.60±0.37 |
| 32 | Vanilla Hyperopt | 0 | 77.52±4.37 | 99.41±0.10 |
| 32 | History_with_F1 | 8052 | 80.42±0.98 | 99.35±0.01 |
| 32 | History_without_F1 | 10203 | 81.54±1.38 | **99.48±0.07** |
| 40509 | Vanilla Hyperopt | 0 | 78.52±0.47 | 87.20±0.36 |
| 40509 | History_with_F1 | 4857 | 83.40±0.60 | **87.48±0.18** |
| 40509 | History_without_F1 | 6803 | 82.72±0.73 | 87.15±0.18 |

Table 5.1: Comparison between Vanilla Hyperopt and HyperOpenML (History_with_F1, History_without_F1) - This figure compares different trial size for each dataset.

quality pipelines in trial prevent Hyperopt from exploring worthless parts of the search space. This kind of prior knowledge can save time and computation resources and offer better results.

| Data set | New input | Removed trials | AVG-accuracy | Best found |
|---|---|---|---|---|
| 3 | 19367 | 299 | 79.80±1.35 | 95.85±0.26 |
| 31 | 6573 | 87 | 69.08±2.07 | 77.33±0.69 |
| 32 | 7267 | 758 | 72.99±5.28 | 99.15±0.20 |
| 40509 | 4721 | 136 | 83.20±0.40 | 87.15±0.07 |

Table 5.2: Remove Low Quality Pipelines - For each dataset number of removed low quality pipelines (under 50% accuracy) is shown. This table also shows the results with new input (only pipelines with more than 50% accuracy are fed)

## 5.4 Largest and Smallest Accuracies

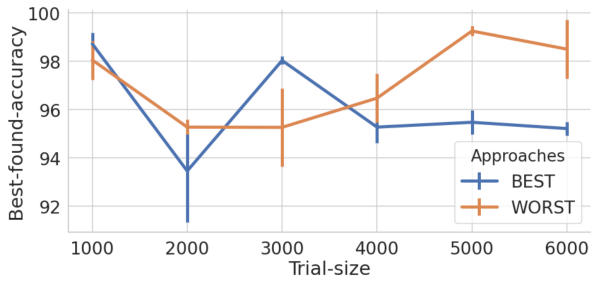
The need to have low-quality pipelines in history (see: Section 5.3), raises the idea of selecting largest and smallest accuracies from the available trials. In this strategy, N best and N worst pipelines are selected based on their final accuracy. Figure 5.3 represents the comparison between these two scenarios. As shown, neither only best-performing history or only worst-performing history are suitable subsets. In most cases, when a mix of good-performing and bad-performing pipelines reaches a balance (in terms of available pipelines in the history), TPE algorithm achieves better results, compared to the extreme situation of only including the best or worst pipelines.

Moreover, as mentioned, TPE divides the received prior knowledge into two parts ($l(x)$ and $g(x)$). One part belongs to pipelines that have the best-performing distribution ($l(x)$), and the other part covers the rest of the pipelines ($g(x)$). When those two parts feed with only best-performing or worst-performing pipelines, and not with a mixture of both distributions, it seems this misleads Bayesian Optimization.

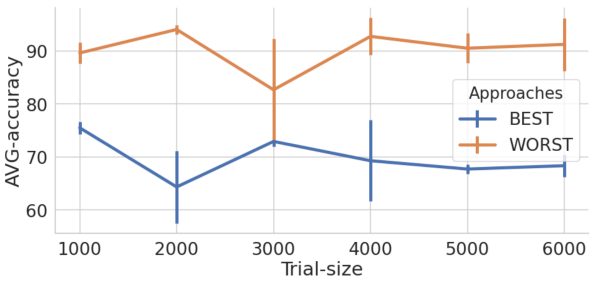
For example, if the trial is fed with the best available pipelines, TPE receives a proper distribution about the part of space that is worthy of exploring, while at the same time, TPE is misleading about the worthless part of space. This is why in most datasets, whenever there is a balance between the two best and worst distributions, the final results are better. Moreover, according to the distribution of accuracies of given pipelines (see Figure 3.9), most of the available trials have good performing accuracy (more than 50 percent accuracy). That means whenever N number of worst pipelines are selected, TPE reaches a better balance in comparison to only N best pipelines. As a result, TPE achieves better accuracy in average and best-found section (first and the second column of Figure 5.3, respectively).

## 5.5 Histogram-based History

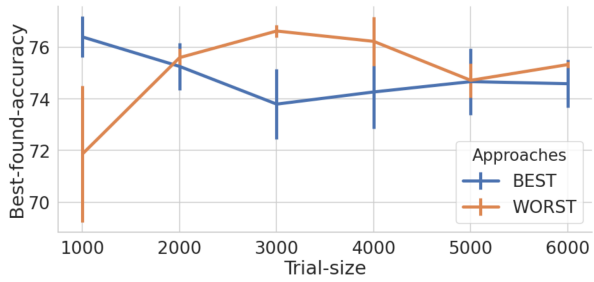
In this section, given pipelines are categorized based on their distributions in terms of accuracy into five bins (see Figure 3.9). Then, for each execution, an equal portion is selected from each bin and fed as prior knowledge to the TPE algorithm. With this approach, a fair chance is given to each of the bins to provide pipelines. Although an equal portion from each bin is selected, the selected pipeline from each bin has a different size due to different bin sizes. Figure 5.4 illustrates the result of this approach. Moreover, it shows in most of the datasets, that increasing the selected portion from each bin, also increases the best-found pipeline in terms of accuracy in most of the datasets. In other words, for most datasets, selecting all members of a bin instead of just a portion of members leads to better final results. Because of the complexity of search space, when more insight about search space is provided to TPE, it can perform better at the end. As an example, Figure 5.4 Section b, shows that TPE achieves a better maximum when we provide more information from each bin. A comparison of the results of selecting each portion, compared to selecting all members of each container in full 100% (Figure 5.2) confirms this idea as well.



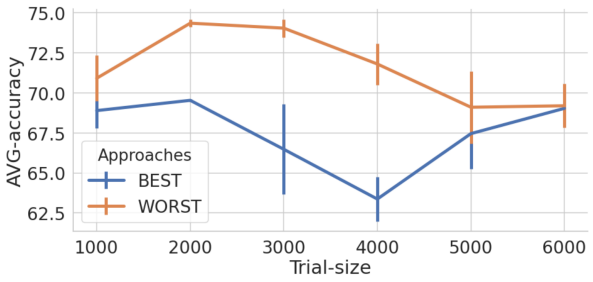

(a) Best Accuracy - Dataset 3



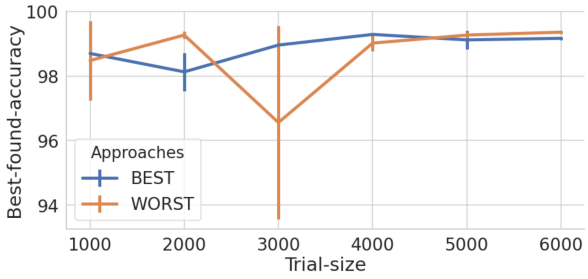

(b) Average Accuracy- Dataset 3



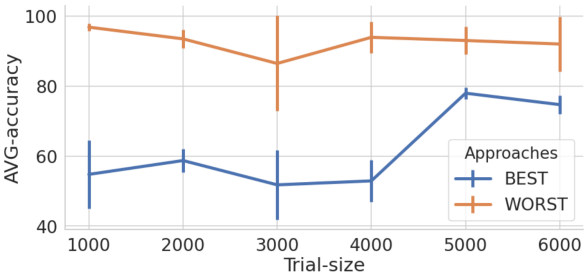

(c) Best Accuracy - Dataset 31



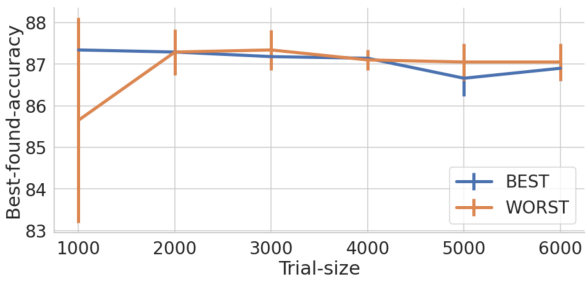

(d) Average Accuracy - Dataset 31



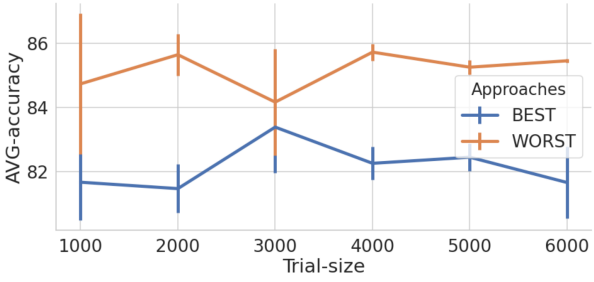

(e) Best Accuracy - Dataset 32




(f) Average Accuracy - Dataset 32

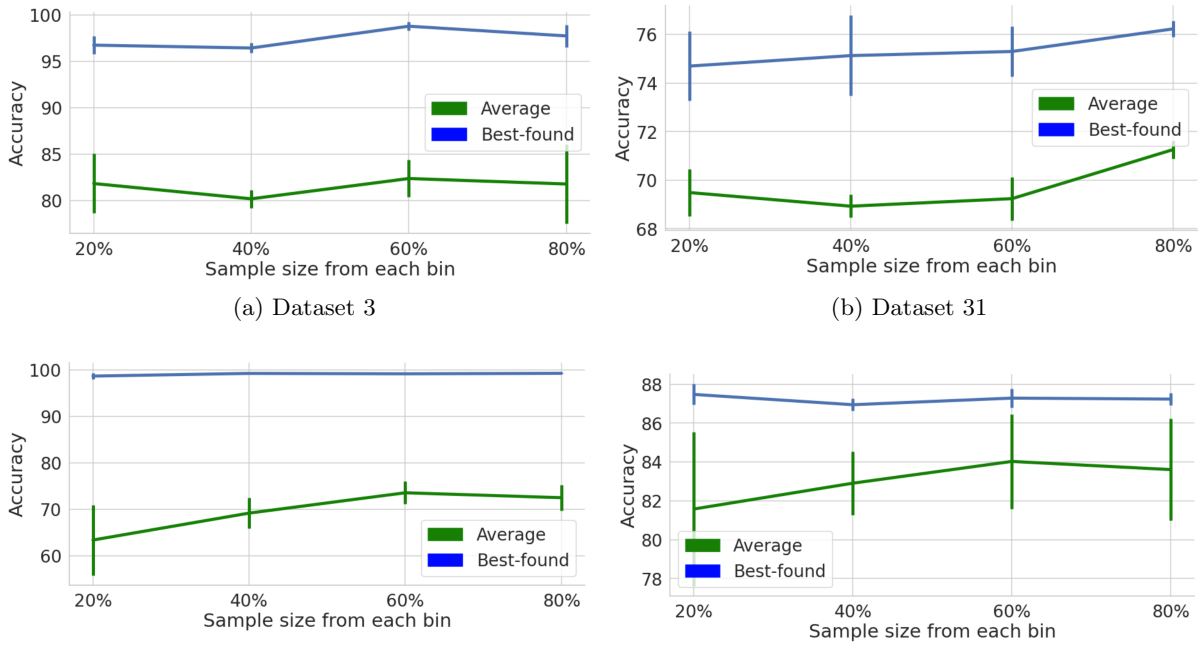



(g) Best Accuracy - Dataset 40509




(h) Average Accuracy- Dataset 40509

Figure 5.3: Largest and Smallest Accuracies - Two warm-starting strategies are compared for each dataset. In best scenario TPE is only fed with N best available pipelines (sort the available pipelines based on their final accuracy and only N best pipelines are selected) in the history while in worst scenario trial is filled with N worst pipelines.




(a) Dataset 3

(b) Dataset 31

(c) Dataset 32

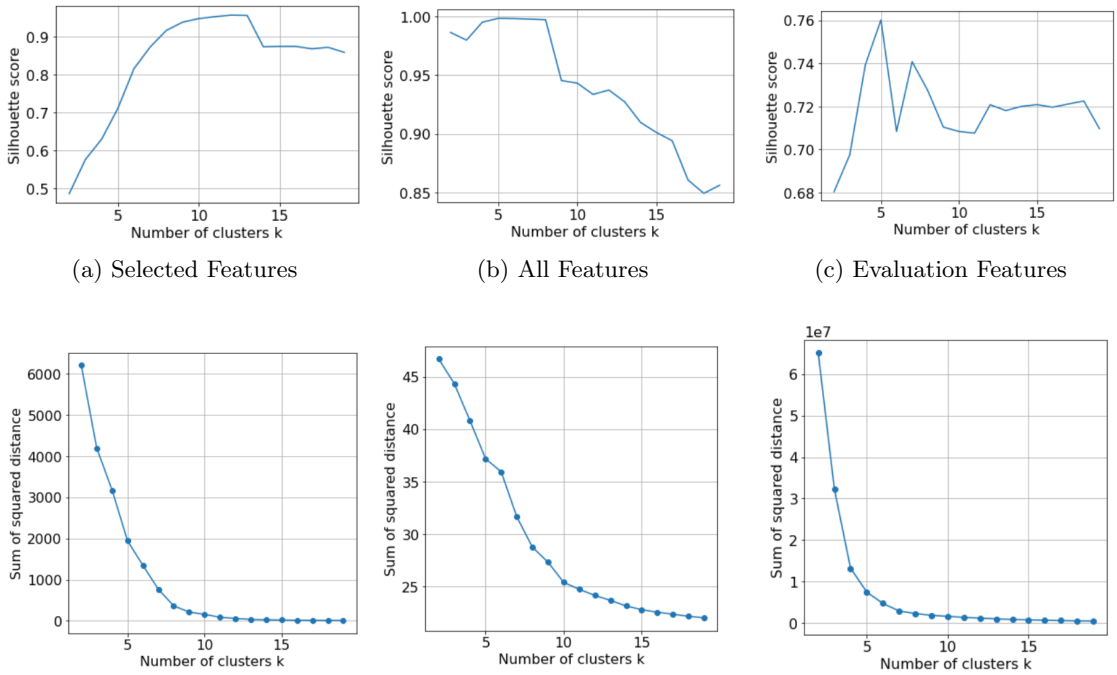
(d) Dataset 40509

Figure 5.4: Histogram Approach - Exploited pipelines from OpenML are categorized into five bins, and in each TPE execution, an equal portion from each bin is fed to warm-start the TPE algorithm. In this graph, the horizontal axis indicates the equal sample percentage from each bin. In the selection process in each bin, the points that are closer to the mean of the bin got selected.

## 5.6 Cluster-based History

In this section, for each dataset, we transform all exploited pipelines into vectors and, with the help of K-Means, group them to separate clusters. We employ different strategies to sample from each cluster and feed the trial. We use SSE and Silhouette to determine the number of clusters. Furthermore, we include details about each cluster's accuracy as well as the minimum and maximum.

One of the essential criteria in clustering with K-Means is defining the number of clusters. In this regard, we have defined the feature space in several scenarios. Figure 5.5 shows the different feature sets that can result in the different number of clusters (K) suggested by Silhouette and SSE. Figure 5.5 represents the three different feature set for finding the number of clusters (K) by Silhouette (firs row) and SSE (second row) approach. Sub-figure *a* and *d* show the number of clusters when we only consider particular features (Accuracy, F1-score, data preprocessing, and feature prepossessing)




(a) Selected Features (b) All Features (c) Evaluation Features

(d) Selected Features (e) All Features (f) Evaluation Features

Figure 5.5: Determine the K for Different Feature Set in Dataset 31

for determining the K. Sub-figure *a* shows 13 different clusters while SSE points at 10 clusters. Sub-figure *b* and *e* suggest 5 and 10 clusters, respectively; when we consider all available features for determining the K. When we only consider the evaluation (Metric) features for determining the number of clusters two last Sub-figure *c* and *f* denote 5 and 9 clusters, respectively.

In our experiments, as initial results suggest, we have considered all features to determine the number of clusters. In more details, we have considered all hyperparameters to build the corresponding vector for each run. Then, we use these vectors to determine the suitable number of clusters. For more information about the number of clusters for each dataset the reader can refer to the appendix of this thesis.

In the following, Figures (5.6, 5.7, 5.8, 5.9) show the detailed of clustering for each dataset in more details. In these graphs, each circle represents a cluster. Each circle reports name of the cluster and available options for data preprocessing, feature preprocessing, and classifiers. It also announces the range of available accuracy in the cluster as well as cluster's size. Furthermore, Table 5.3 represents the results of three strategies for feeding the trial, including the biggest cluster, biggest SSE, and all cluster samples. Firstly, the biggest cluster stands for a time we feed the trial only with the biggest clus-

ter in terms of the number of points. Secondly, the biggest SSE represents, feeding the trial with cluster that has the biggest SSE value. (The cluster has the biggest sum of the square distance between the centroid and each member in the cluster). Lastly, all cluster sample represents the sampling strategy from all available clusters. The sample size for each dataset is equal to size of the smallest cluster in that dataset.

**Dataset 3**
While Silhouette indicates 2 different clusters, SSE has a choice of 12 different clusters for exploited pipelines for dataset 3. We have employed the SSE idea because 2 clusters it not enough for our approaches. Figure 5.6 shows 12 clusters suggested by SSE in more details. Cluster 3, with 8798 members, is the biggest in terms of members, and it has a diverse number of classifiers. There are 7 different types of classifiers accommodated once by Cluster 3 and then by Cluster 6, while tolerance of accuracy in Cluster 6 is less than Cluster 3. It is also interesting to mention that 9 clusters in Figure 5.6 end up only to SVC as a classifier, and it shows different hyperparameters of SVC changing their clusters.

The first two rows of Table 5.3 show the result of warm-starting the TPE with the biggest and most diverse cluster, as well as the sample from all clusters. Moreover, in all cluster samples, the prior knowledge was fed with the sample from all clusters.

**Dataset 31**
Although Silhouette hints at 5 different clusters, SSE chooses 10 different clusters for exploited pipelines for dataset 3. Figure 5.7 shows 5 different clusters suggested by Silhouette. Cluster 0, with 5295 members, is the biggest cluster and simultaneously, is the most diverse cluster in terms of classifier's number, as well as feature and data processing. Due to a diverse range of its hyperparameters, the Fkceigenpro classifier (Fast Kernel Classifier) turns into 4 clusters, while the accuracy is not diverse in any of its clusters. Table 5.3 shows the result of warm-starting the TPE with the most significant and diverse cluster, as well as the sample from all clusters for this dataset. The reader can refer to the appendix to check the diversity of clusters suggested by the SSE approach.

**Dataset 32**
Silhouette suggests 4 different clusters, whereas SSE finds 12 different clusters for exploited pipelines for dataset 32. Figure 5.8 shows the distribution of each cluster suggested by Silhouette for dataset 32. Cluster 0 with 7707 members is the most significant cluster, while Cluster 2 with 321 members is the most diverse in terms of the sum of square error. Furthermore, Cluster 0 has the most diverse range of accuracy in comparison to other clusters. Table 5.3 shows the result of warm-starting the TPE with the most significant and diverse cluster, as well as the sample from all clusters for this dataset. Dataset 32 is the only dataset where the most significant cluster is not the cluster with the biggest sum of square error; this is why it has 3 rows in that table. The reader can

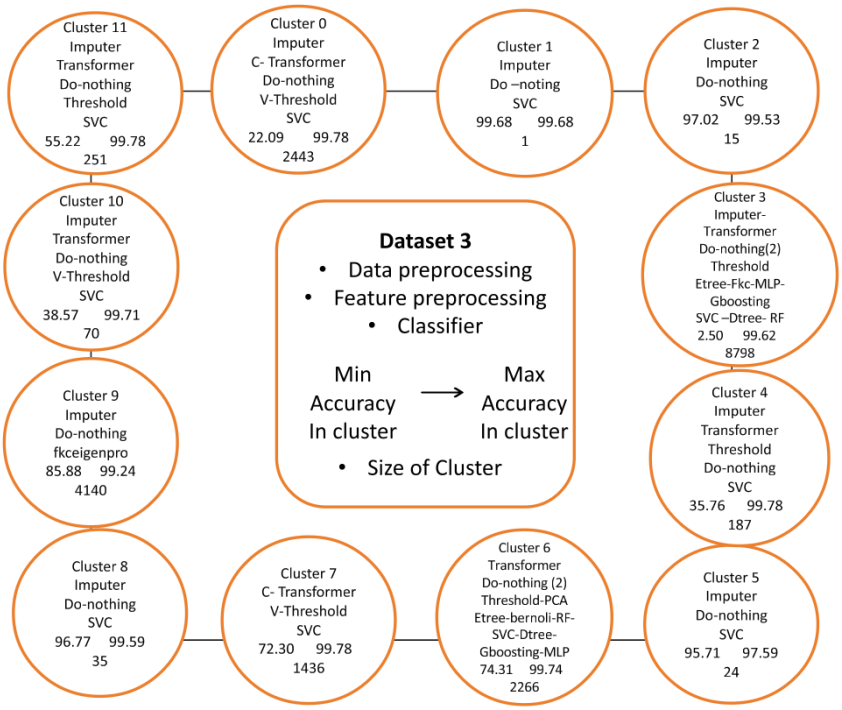



Figure 5.6: Cluster Distributions, Dataset 3 - K-Means categorizes the exploited pipelines for this dataset into 12 different clusters. Cluster 1 with only one member and Cluster 3 with 8798 members are the smallest and biggest Clusters for this dataset, respectively. Furthermore, Cluster 3 is the most diverse cluster in the range of accuracy.

refer to the appendix to check the diversity of clusters suggested by the SSE approach.

**Dataset 40509**

Although Silhouette suggests 2 different clusters, SSE finds 8 different clusters for exploited pipelines for dataset 40509. We have employed the SSE idea because 2 clusters it not enough for our approaches. Figure 5.9 illustrates 8 clusters suggested by SSE. The flow that consists of the SVC as a classifier is the most popular pipeline in history, which is why 7 clusters end up to SVC as the classifier. These flows are only different in the hyperparameters. Cluster 2, with 2800 members, is the biggest cluster, and includes the 4 different classifiers. Table 5.3 shows the result of warm-starting the TPE with the most significant and diverse cluster, as well as the sample from all clusters for this dataset.

Table 5.3, illustrates a comparison between three different strategies (Biggest cluster, Biggest SSE, and Cluster sample) in more details. In most datasets (three out of four), the average accuracy for the cluster sample strategy is better than the biggest SSE. While the best-found pipeline for most of the datasets (three out of four) is better in the biggest SSE strategy. It seems the biggest SSE strategy achieves a better result for

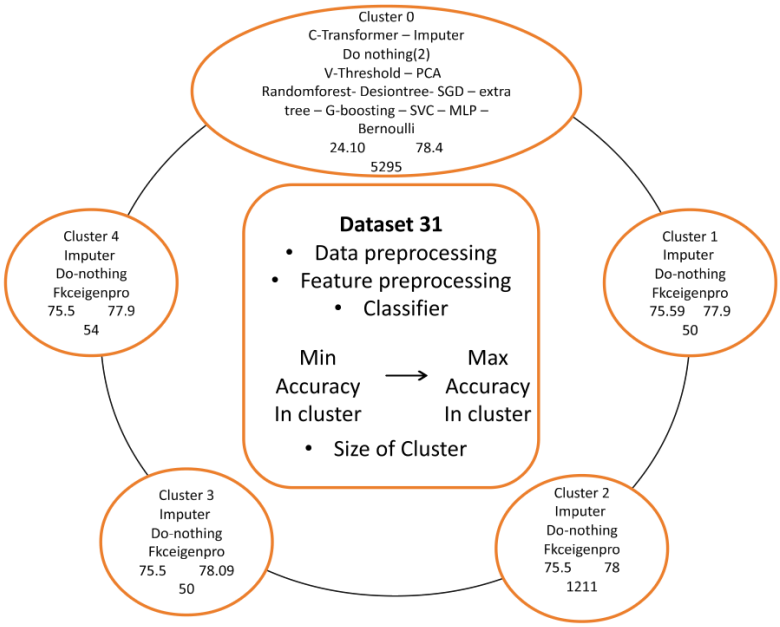



Figure 5.7: Cluster Distributions, Dataset 31 - K-Means categorizes the exploited pipelines for this dataset into 5 different clusters. Cluster 1,3 with 50 members and Cluster 0 with 5295 members are the smallest and biggest Clusters for this dataset, respectively. Furthermore, Cluster 0 is the most diverse cluster in the range of accuracy.

each dataset because it feeds the trial with the most diverse cluster (the sum of the square distance between the centroid and each member in the cluster is more than other clusters). This level of diversity in the trial could be beneficial for TPE and gives the algorithm the wise insight about the search space.

## 5.7 Unique Accuracy

This approach studies the effect of removing duplicate pipelines in terms of final accuracy. In this method, exploited runs that end up to the same accuracy are considered duplicates. From each duplicate group, only one pipeline which is selected randomly, is kept in history, and the rest are removed. This experiment shows the importance of duplicate pipelines in terms of accuracy. Table 5.4, shows the results for unique pipelines whenever they were fed as prior knowledge. In most cases, by removing duplicate pipelines, TPE cannot reach the same results (before removing the duplicate pipelines). It seems removing duplicate pipelines limits the insight of TPE over search space and cause the drop in best-found and average accuracy. Moreover, this removal in history could mislead the TPE algorithm, so that vanilla Hyperopt outperforms this approach in best-found pipeline.

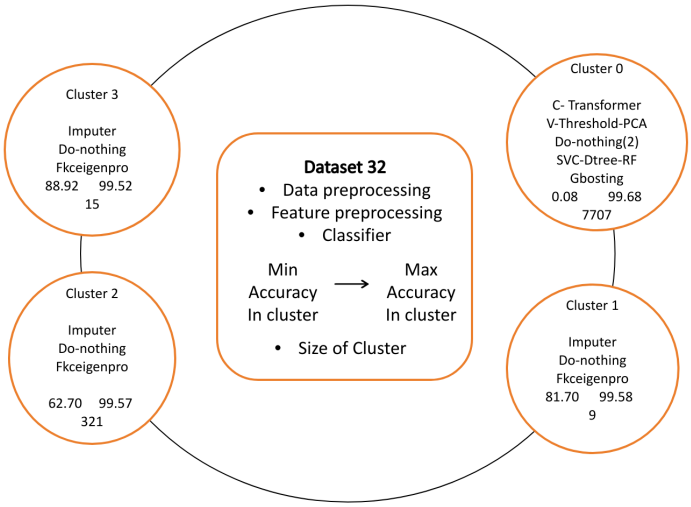



Figure 5.8: Cluster Distributions, Dataset 32 - K-Means categorizes the exploited pipelines for this dataset into four different clusters. Cluster 1 with only nine members and Cluster 0 with 7707 members are the smallest and biggest Clusters for this dataset, respectively. Furthermore, Cluster 0 is the most diverse cluster in the range of accuracy, while Cluster 2 has the biggest SSE distance between the centroid and each member in the cluster.

| Data set | Approach name | AVG-accuracy | Best found |
|---|---|---|---|
| 3 | Biggest cluster Biggest SSE | 82.26±3.78 | 97.67±1.09 |
| 3 | All cluster sample | 74.07±12.65 | 95.59±1.73 |
| 31 | Biggest cluster Biggest SSE | 67.32±0.40 | 76.80±0.33 |
| 31 | All cluster sample | 68.74±0.92 | 74.90±0.88 |
| 40509 | Biggest cluster Biggest SSE | 82.66±0.61 | 87.24±0.54 |
| 40509 | All cluster sample | 85.77±0.48 | 87.24±0.24 |
| 32 | Biggest cluster | 78.07±1.58 | 99.37±0.03 |
| 32 | Biggest SSE | 67.02±2.16 | 99.44±0.08 |
| 32 | All cluster sample | 73.35± 10.68 | 95.23±4.42 |

Table 5.3: Different Approach Clustering - Each row of this table represents feeding the trial with a different strategy. As shown, for three datasets, two approaches (biggest cluster and biggest SSE) select the same cluster, and the only reason is that only Dataset 32 has three rows

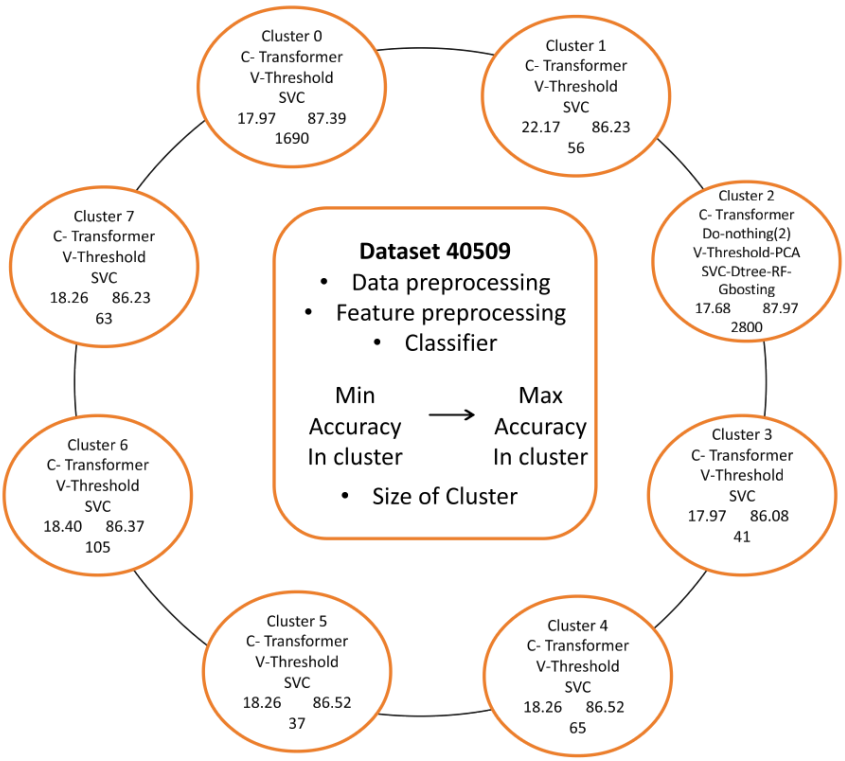



Figure 5.9: Cluster Distributions, Dataset 40509 - K-Means categorizes the exploited pipelines for this dataset into eight different clusters. Cluster 5 with 37 members and Cluster 2 with 2800 members are the smallest and biggest Clusters for this dataset, respectively. Furthermore, Cluster 2 is the most diverse cluster in the range of accuracy.

| Data set | New input | AVG-accuracy | Best found |
|---|---|---|---|
| 3 | 1118 | 83.24±7.03 | 97.90±0.67 |
| 31 | 294 | 69.70±0.24 | 77.19±0.22 |
| 32 | 1965 | 70.13±6.56 | 98.69±0.65 |
| 40509 | 275 | 80.37±2.76 | 87.15±0.07 |

Table 5.4: Unique Accuracy Pipelines - For each dataset, exploited pipelines that end up into the same final accuracy are considered as duplicate pipelines. This approach selects only one pipeline from each duplicate group. Column New input in this table represents the new number of trials after removing duplicates from each dataset.

## 5.8 Hyper-hyperparameter Tuning

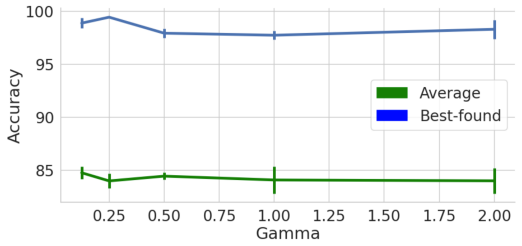
This section is about the results of tuning the hyperparameter of Hyperopt on top of the best performing approach for each dataset. The direct influence of changing gamma and linear forgetting is investigated in more details. Figure 5.10 illustrates the result

when gamma increases at the same time (left column). In addition, it shows the effect of removing linear forgetting while gamma is increasing (right column). Figure 5.10 shows that increasing the gamma, in most cases, enhances the average quality of pipelines (left column). This influence is stronger when we remove the linear forgetting (right column). Although the best-found pipelines are very close in each situation (with and without linear forgetting), most of the time, linear forgetting is helpful in finding a better pipeline.

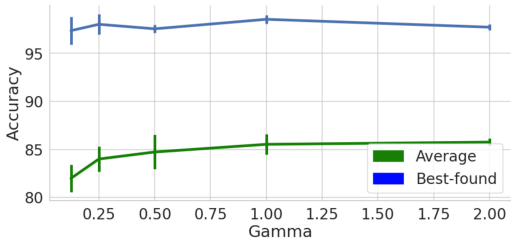
## 5.9 Summary

In Chapter 5, we have shown the direct influence of trial size on TPE run time. This chapter also covered a comparison between vanilla Hyperopt and HyperOpenML. Then it followed with different strategies (e.g., Higher than Random Quality, Special Points, Histogram, Clustering, Unique Accuracy ) for HyperOpenML to select the pipelines from the experimental database. This chapter concluded with Hyper-hyperparameter tuning for best-found pipelines for each dataset. Chapter 6 is the final chapter of this thesis and covers the conclusion and future work.

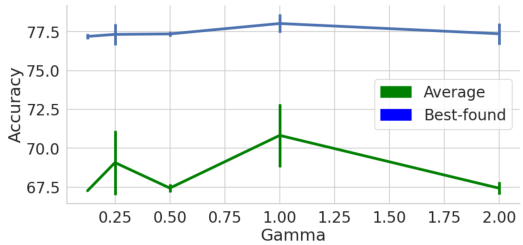
It seems warm-starting the TPE algorithm is efficient when we prepare a balanced history with good and bad performing pipelines. Worst-performing pipelines prevent TPE from exploring part of the search space, which is worthless, and good-performing pipelines lead the TPE to exploit more. Thus, it is essential to balance the number of good and bad-performing pipelines in history. The duplicated pipelines in terms of final accuracy are also necessary because they give a better insight to TPE about the search space. Keep the balance in the number of each flow in history is also another factor. Lack of information about the specific flow can mislead the TPE algorithm. It is better to present a decent number of pipelines for each flow. Our experiments show it is possible to achieve better average accuracy (for proposed pipelines) if we remove the effect of linear forgetting and increase the gamma.



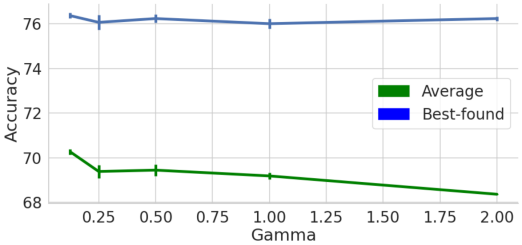

(a) Change Gamma - Dataset3



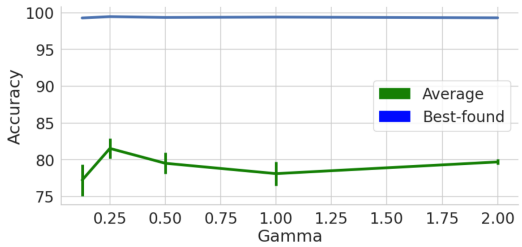

(b) Change Gamma and Remove LF - Dataset3



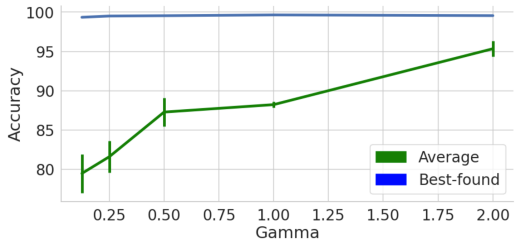

(c) Change Gamma - Dataset 31



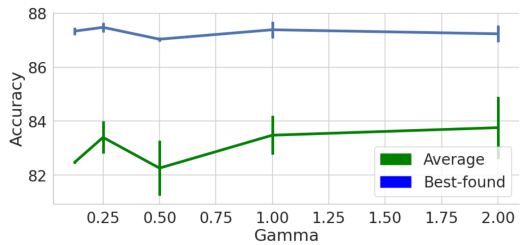

(d) Change Gamma and Remove LF - Dataset 31



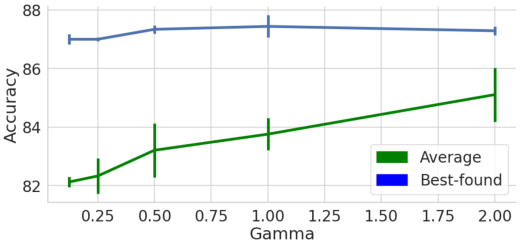

(e) Change Gamma - Dataset 32




(f) Change Gamma and Remove LF - Dataset 32




(g) Change Gamma - Dataset 40509




(h) Change Gamma and Remove LF - Dataset 40509

Figure 5.10: Change Gamma and Linear Forgetting - Hyper-hyperparameter tuning for each dataset. This figure in the left column shows the impact of changing gamma on average and best-found pipeline accuracy. The right column of this figure illustrates the direct influence of changing gamma when liner-forgetting is removed.

# 6 Conclusion

This thesis has studied automatic machine learning using the experimental database. We enable AutoML to utilize experiment database (OpenML) to design the search space and guide the search process. We have considered how prior knowledge, or warm-starting, can help Bayesian Optimization and especially TPE to achieve better results. We also discussed the characteristics of the experimental database, and different strategies for warm-starting Bayesian Optimization, in order to save required time and computation.

Based on the previous observations and the characteristics of algorithms and tools we analyzed in Chapter 2, this thesis proposed a mechanism for efficient hyperparameter optimization, which is called HyperOpenML. As its name suggests, it combines the Hyperopt as one tool for optimization with OpenML as an online experimental database, to make the hyperparameter tuning process even more efficient.

Section 5 described the results of different strategies for warm-starting the Bayesian Optimization for hyperparameter optimization, on a variety of problems with different datasets. The vanilla Hyperopt was our baseline method. The results showed that HyperOpenML, in comparison to baseline, improved both the quality for the average of the proposed pipelines, as well as the best-found pipeline. Remarkably, this improvement happened even within a limited number of iterations (100 iterations, limited by resource and time constraints). Moreover, we have seen that the existence of both low-quality and duplicate (in terms of accuracy) pipelines, in the given history can prevent TPE from exploring worthless parts of the search space. This information seems to be additionally helpful.

## 6.1 Future Work

Although we enabled AutoML to use experimental database, it could be interesting to work on fully automatic adoptive search space based on experimental databases. It means not only AutoML can use the experimental databases but also prune the search space adaptively and automatically. To do so, AutoML needs to have a feedback loop for narrowing down the search space based on the experimental databases.

Besides TPE, SMBO has other strategies, like SMAC and Gaussian Process, for approximating the expensive function (i.e., hyperparameter optimization). To extend the warm-starting Bayesian Optimization with experimental database, it could be interesting to study the influence of warm-starting other SMBO strategies like SMAC and GP,

and compare them after warm-starting in terms of quality and required time.

# List of Acronyms

| | |
|---|---|
| SMBO | Sequential Model-Based Global Optimization |
| AutoML | Automatic Machine learning |
| HPO | Hyperparameter Optimization |
| CASH | Combined Algorithm Selection and Hyperparameter optimization problem |
| TPE | Tree-structured Parzen Estimator |
| SMAC | Sequential Model-based Algorithm Configuration |
| GP | Gaussian Process |
| DFC | Data preprocessing, Feature preprocessing, Classifier |
| EI | Expected Improvement |
| BO | Bayesian Optimization |
| SSE | Sum of Square Error |

# Bibliography


[1] Medium. *Difference between Traditional programming versus Machine Learning from a PM perspective*, 2020. `https://productcoalition.com/difference-between-traditional-programming-versus-machine-learning-from-a-pm-pers`

[2] Matthias Feurer and Frank Hutter. Hyperparameter optimization. In *Automated Machine Learning*, pages 3–33. Springer, 2019.

[3] Matthias Feurer and Frank Hutter. Towards further automation in automl. In *ICML AutoML workshop*, page 13, 2018.

[4] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Advances in neural information processing systems*, pages 2962–2970, 2015.

[5] 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*, pages 847–855. ACM, 2013.

[6] 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. *arXiv preprint arXiv:1012.2599*, 2010.

[7] James Max Kanter and Kalyan Veeramachaneni. Deep feature synthesis: Towards automating data science endeavors. In *2015 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, pages 1–10. IEEE, 2015.

[8] Joaquin Vanschoren, Jan N Van Rijn, Bernd Bischl, and Luis Torgo. Openml: networked science in machine learning. *ACM SIGKDD Explorations Newsletter*, 15(2):49–60, 2014.

[9] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[10] Gavin C Cawley and Nicola LC Talbot. On over-fitting in model selection and subsequent selection bias in performance evaluation. *Journal of Machine Learning Research*, 11(Jul):2079–2107, 2010.

[11] Antonio Valerio Miceli Barone, Barry Haddow, Ulrich Germann, and Rico Sennrich. Regularization techniques for fine-tuning in neural machine translation. *arXiv preprint arXiv:1707.09920*, 2017.

[12] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(Feb):281–305, 2012.

[13] Bert L Hartnell, Douglas F Rall, and Kirsti Wash. On well-covered cartesian products. *Graphs and Combinatorics*, 34(6):1259–1268, 2018.

[14] Mael Fabien. *A Guide to Hyperparameter Optimization (HPO)*, 2020. `https://maelfabien.github.io/machinelearning/Explorium4/#randomized-search`.

[15] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration (extended version). *Technical Report TR-2010–10, University of British Columbia, Computer Science, Tech. Rep.*, 2010.

[16] Peter I Frazier. A tutorial on bayesian optimization. *arXiv preprint arXiv:1807.02811*, 2018.

[17] Bobak Shahriari, Ziyu Wang, Matthew W Hoffman, Alexandre Bouchard-Côté, and Nando de Freitas. An entropy search portfolio for bayesian optimization. *arXiv preprint arXiv:1406.4625*, 2014.

[18] Matthew D Hoffman, Eric Brochu, and Nando de Freitas. Portfolio allocation for bayesian optimization. In *UAI*, pages 327–336. Citeseer, 2011.

[19] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554, 2011.

[20] Donald R Jones. A taxonomy of global optimization methods based on response surfaces. *Journal of global optimization*, 21(4):345–383, 2001.

[21] Julien Villemonteix, Emmanuel Vazquez, and Eric Walter. An informational approach to the global optimization of expensive-to-evaluate functions. *Journal of Global Optimization*, 44(4):509, 2009.

[22] Niranjan Srinivas, Andreas Krause, Sham M Kakade, and Matthias Seeger. Gaussian process optimization in the bandit setting: No regret and experimental design. *arXiv preprint arXiv:0912.3995*, 2009.

[23] James Joyce. Bayes' theorem. *Stanford Encyclopedia of Philosophy Archive*, 2003.

[24] Fereshteh Ghanbari-Adivi and Mohammad Mosleh. Text emotion detection in social networks using a novel ensemble classifier based on parzen tree estimator (tpe). *Neural Computing and Applications*, 31(12):8971–8983, 2019.

[25] Aleksei Mashlakov, Ville Tikka, Lasse Lensu, Aleksei Romanenko, and Samuli Honkapuro. Hyper-parameter optimization of multi-attention recurrent neural network for battery state-of-charge forecasting. In *EPIA Conference on Artificial Intelligence*, pages 482–494. Springer, 2019.

[26] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization*, pages 507–523. Springer, 2011.

[27] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[28] Jamie Shotton, Andrew Fitzgibbon, Mat Cook, Toby Sharp, Mark Finocchio, Richard Moore, Alex Kipman, and Andrew Blake. Real-time human pose recognition in parts from single depth images. In *CVPR 2011*, pages 1297–1304. Ieee, 2011.

[29] Carl Edward Rasmussen. Gaussian processes in machine learning. In *Summer School on Machine Learning*, pages 63–71. Springer, 2003.

[30] CE Rusmassen and CKI Williams. Gaussian process for machine learning, 2005.

[31] Medium. *A brief overview of AutoML*, 2020. `https://medium.com/yogesh-khuranas-blogs/a-brief-overview-of-automl-8a847c3b5f6e`.

[32] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[33] Bernd Bischl, Michel Lang, Lars Kotthoff, Julia Schiffner, Jakob Richter, Erich Studerus, Giuseppe Casalicchio, and Zachary M Jones. mlr: Machine learning in r. *The Journal of Machine Learning Research*, 17(1):5938–5942, 2016.

[34] Tom Schaul, Justin Bayer, Daan Wierstra, Yi Sun, Martin Felder, Frank Sehnke, Thomas RĆ1/4ckstieĆ, and JĆ1/4rgen Schmidhuber. Pybrain. *Journal of Machine Learning Research*, 11(Feb):743–746, 2010.

[35] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. Automated machine learning-methods, systems, challenges, 2019.

[36] Ziawasch Abedjan, Xu Chu, Dong Deng, Raul Castro Fernandez, Ihab F Ilyas, Mourad Ouzzani, Paolo Papotti, Michael Stonebraker, and Nan Tang. Detecting data errors: Where are we and what needs to be done? *Proceedings of the VLDB Endowment*, 9(12):993–1004, 2016.

[37] Nazrul Hoque, Dhruba K Bhattacharyya, and Jugal K Kalita. Mifs-nd: A mutual information-based feature selection method. *Expert Systems with Applications*, 41(14):6371–6385, 2014.

[38] MIT. *The Data Science Machine*, 2020. `https://people.csail.mit.edu/kalyan/dsm/`.

[39] Feature labs. *Feature tools,automate more and more of the feature engineering process*, 2020. `https://github.com/FeatureLabs/featuretools`.

[40] Ron Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, volume 14, pages 1137–1145. Montreal, Canada, 1995.

[41] Frank Hutter, Holger H Hoos, Kevin Leyton-Brown, and Thomas Stützle. Paramils: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306, 2009.

[42] Isabelle Guyon, Kristin Bennett, Gavin Cawley, Hugo Jair Escalante, Sergio Escalera, Tin Kam Ho, Núria Macia, Bisakha Ray, Mehreen Saeed, Alexander Statnikov, et al. Design of the 2015 chalearn automl challenge. In *2015 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2015.

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

[44] Rich Caruana, Alexandru Niculescu-Mizil, Geoff Crew, and Alex Ksikes. Ensemble selection from libraries of models. In *Proceedings of the twenty-first international conference on Machine learning*, page 18. ACM, 2004.

[45] Rich Caruana, Art Munson, and Alexandru Niculescu-Mizil. Getting the most out of ensemble selection. In *Sixth International Conference on Data Mining (ICDM'06)*, pages 828–833. IEEE, 2006.

[46] David H Wolpert. Stacked generalization. *Neural networks*, 5(2):241–259, 1992.

[47] Marc-André Zöller and Marco F Huber. Survey on automated machine learning. *arXiv preprint arXiv:1904.12054*, 9, 2019.

[48] Randal S Olson and Jason H Moore. Tpot: A tree-based pipeline optimization tool for automating machine learning. In *Automated Machine Learning*, pages 151–160. Springer, 2019.

[49] Marc-Andre Zoller and Marco F Huber. Benchmark and survey of automated machine learning frameworks. *arXiv*, 2020.

[50] University of Pennsylvania. *TPOT documentation and codes*, 2020. `http://epistasislab.github.io/tpot/using/`.

[51] Brent Komer, James Bergstra, and Chris Eliasmith. Hyperopt-sklearn: automatic hyperparameter configuration for scikit-learn. In *ICML workshop on AutoML*, volume 9. Citeseer, 2014.

[52] 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.

[53] GitHub. *Hyperopt GitHub documentation*, 2020. `https://github.com/hyperopt/hyperopt/wiki/FMin`.

[54] GitHub. *Hyperopt-gpsmbo GitHub documentation*, 2020. `https://github.com/hyperopt/hyperopt-gpsmbo`.

[55] Medium. *OpenML: Machine Learning as a community*, 2019. `https://towardsdatascience.com/openml-machine-learning-as-a-community-d678306e1a7e`.

[56] James Bergstra, Daniel Yamins, and David Daniel Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. *Rowland Institute of Harvard*, 2013.

[57] Wikipedia. *Area under curve*, 2020. `https://en.wikipedia.org/wiki/Receiver_operating_characteristic`.

[58] openml. *Kappa*, 2020. `https://www.openml.org/a/evaluation-measures/kappa`.

[59] openml. *kb_relative_information_score*, 2020. `https://www.openml.org/a/evaluation-measures/kb-relative-information-score`.

[60] openml. *mean_absolute_error*, 2020. `https://www.openml.org/a/evaluation-measures/mean-absolute-error`.

[61] openml. *mean-prior-absolute-error*, 2020. `https://www.openml.org/a/evaluation-measures/mean-prior-absolute-error`.

[62] openml. *prior-entropy*, 2020. `https://www.openml.org/a/evaluation-measures/prior-entropy`.

[63] openml. *relative-absolute-error*, 2020. `https://www.openml.org/a/evaluation-measures/relative-absolute-error`.

[64] openml. *root-mean-prior-squared-error*, 2020. `https://www.openml.org/a/evaluation-measures/root-mean-prior-squared-error`.

[65] openml. *root-mean-squared-error*, 2020. `https://www.openml.org/a/evaluation-measures/root-mean-squared-error`.

[66] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

[67] K Krishna and M Narasimha Murty. Genetic k-means algorithm. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 29(3):433–439, 1999.

[68] MA Syakur, BK Khotimah, EMS Rochman, and BD Satoto. Integration k-means clustering method and elbow method for identification of the best customer profile cluster. In *IOP Conference Series: Materials Science and Engineering*, volume 336, page 012017. IOP Publishing, 2018.

[69] Tippaya Thinsungnoena, Nuntawut Kaoungkub, Pongsakorn Durongdumronchaib, Kittisak Kerdprasopb, and Nittaya Kerdprasopb. The clustering validity with silhouette and sum of squared errors. *learning*, 3(7), 2015.

[70] Hong Bo Zhou and Jun Tao Gao. Automatic method for determining cluster number based on silhouette coefficient. In *Advanced Materials Research*, volume 951, pages 227–230. Trans Tech Publ, 2014.

[71] Zachary C Lipton, Charles Elkan, and Balakrishnan Naryanaswamy. Optimal thresholding of classifiers to maximize f1 measure. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 225–239. Springer, 2014.

[72] Ye Nan, Kian Ming Chai, Wee Sun Lee, and Hai Leong Chieu. Optimizing f-measure: A tale of two approaches. *arXiv preprint arXiv:1206.4625*, 2012.

[73] Evgeny Antipov and Elena Pokryshevskaya. Applying chaid for logistic regression diagnostics and classification accuracy improvement. *Journal of Targeting, Measurement and Analysis for Marketing*, 18(2):109–117, 2010.

[74] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. Large-scale parallel collaborative filtering for the netflix prize. In *International conference on algorithmic applications in management*, pages 337–348. Springer, 2008.

[75] Paper Swith Code. *Image Classification on ImageNet*, 2020. `https://paperswithcode.com/sota/image-classification-on-imagenet`.

[76] Netflix. *Netflix Prize*, 2020. `https://www.netflixprize.com/assets/rules.pdf`.

# Appendix

This appendix starts with cluster distribution for Dataset 31, 32 (Section A.1), and it follows with a comparison between SSE and Silhouette for determining the number of clusters for the K-Means algorithm (Section A.2). It concludes with feeding the TPE with best and worst points (Section A.3).

## A.1 Cluster-based Approach

Figure A.1 illustrates, 10 different clusters suggested by SSE for Dataset 31. Cluster 8, with 2973 is the most diverse and biggest cluster for this dataset. It is interesting that seven different clusters only contain the Fkceigenpro classifier (Fast Kernel Classifier) as a classifier.

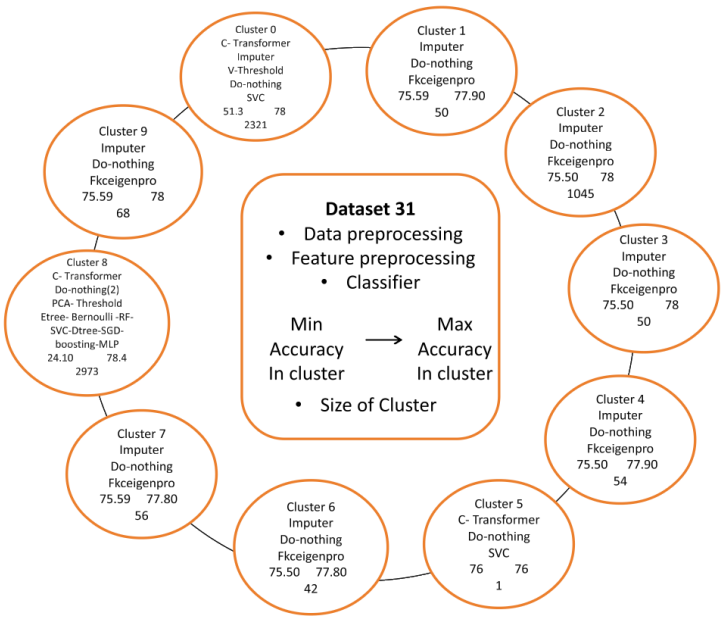



Figure A.1: Cluster Distributions, Dataset 31 - K-Means based on SSE categorizes the exploited pipelines for this dataset into 10 different clusters. Cluster 5 with only one member and Cluster 8 with 2973 members are the smallest and biggest Clusters for this dataset, respectively. Furthermore, Cluster 8 is the most diverse cluster in the range of accuracy.

Figure A.2 shows the 11 clusters, suggested by SSE for Dataset 32. Difference in hyperparameters *Gamma* and *C* in SVC causes five different clusters to have SVC as a classifier.

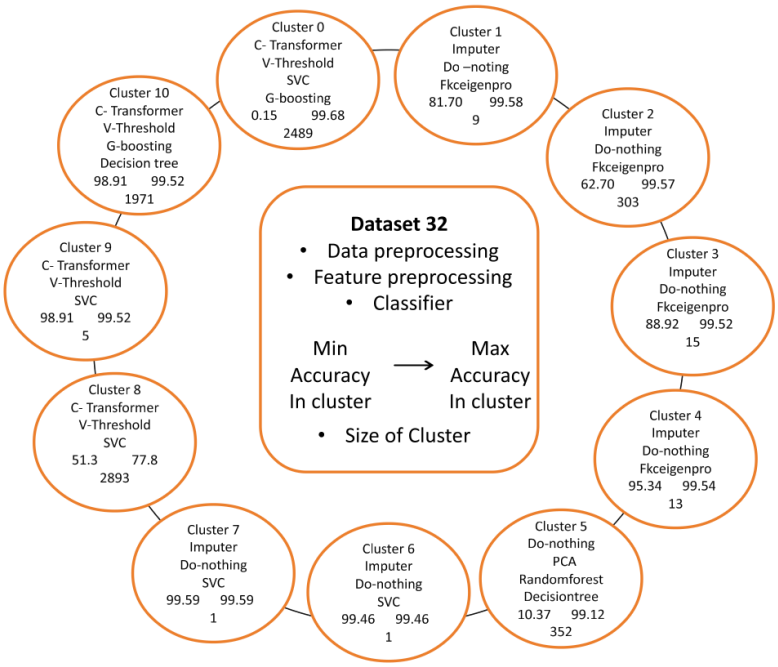
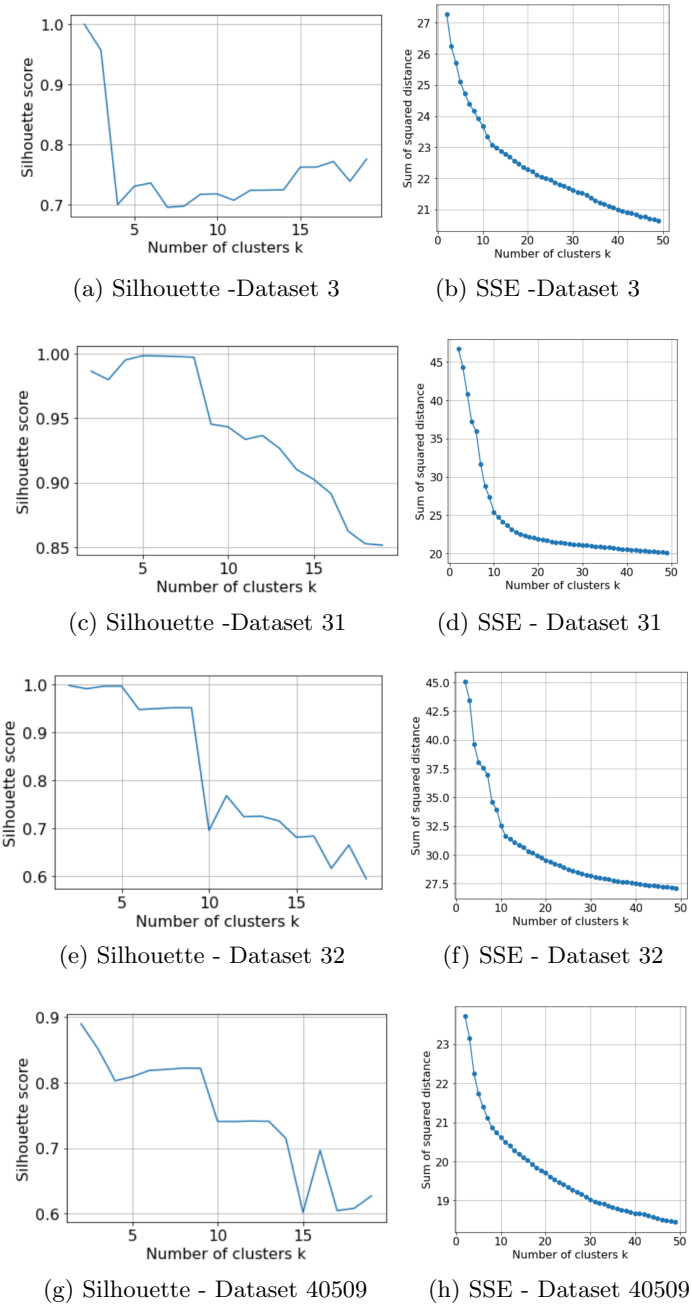



Figure A.2: Cluster Distributions, Dataset 32 - K-Means based on SSE categorizes the exploited pipelines for this dataset into 11 different clusters. Cluster 6, 7 with only one member and Cluster 8 with 2893 members are the smallest and biggest Clusters for this dataset, respectively. Furthermore, Cluster 0 is the most diverse cluster in the range of accuracy.

## A.2 Silhouette versus SSE

Figure A.3 compares the number of clusters suggested by Silhouette and SSE for each dataset. Silhouette gets maximum when each object has great cohesion (similarity to its cluster) and good separation (far from other clusters). For each data point, calculate the average Euclidean distance between that point and all points belonging to the same cluster (Distance A). Then calculate the average distance between that data point and closest cluster (this cluster is like the second-best choice for considering data point) (Distance B). If that data point is well-grounded, distance B needs to be large, and Distance A needs to be small. In the same way (B - A) should be maximum as possible. While SSE defines a sum of the square distance between the centroid and each member in the cluster. Plotting K against SSE gives us the Figure which known as elbow graph. In this graph, increasing the k case the decrease within-cluster SSE (distortion). This is




(a) Silhouette -Dataset 3 (b) SSE -Dataset 3

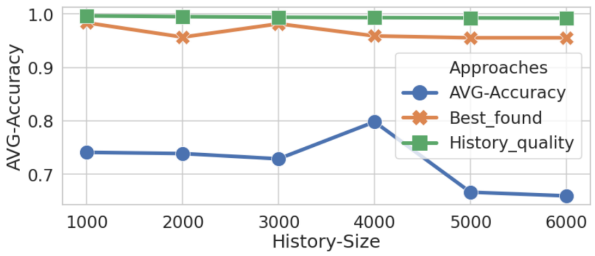
(c) Silhouette -Dataset 31 (d) SSE - Dataset 31

(e) Silhouette - Dataset 32 (f) SSE - Dataset 32

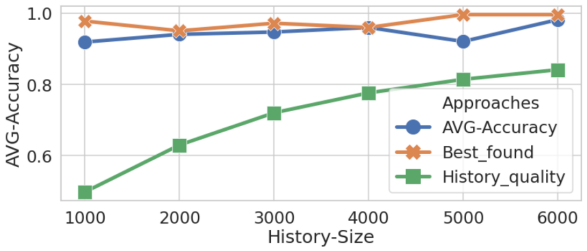
(g) Silhouette - Dataset 40509 (h) SSE - Dataset 40509

Figure A.3: Comparison between Silhouette and SSE for Find the number of clusters for each dataset.

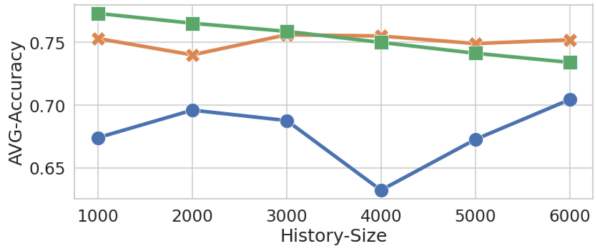
because the samples will be closer to their assigned centroid [69].



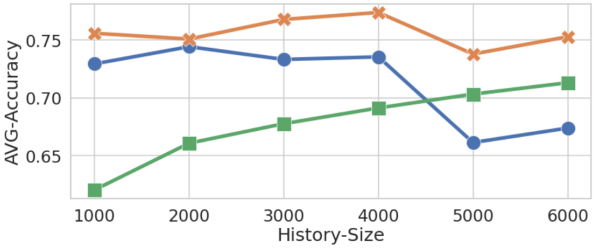

(a) Best points- Dataset 3



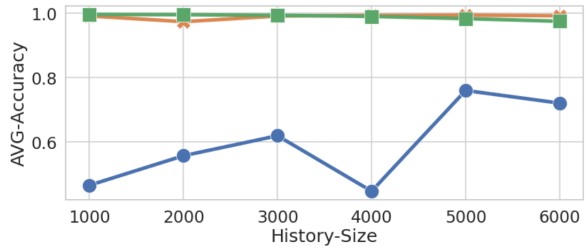

(b) Worst points- Dataset 3



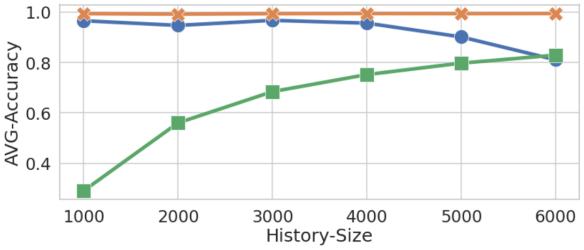

(c) Best points-Dataset 31



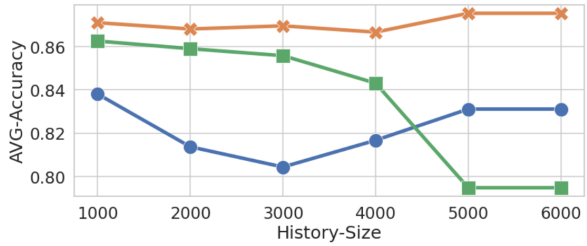

(d) Worst points- Dataset 31



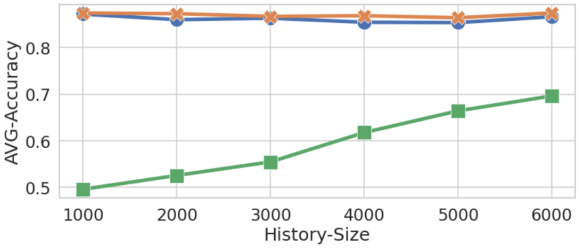

(e) Best points-Dataset 32




(f) Worst points- Dataset 32




(g) Best points-Dataset 40509




(h) Worst points- Dataset 40509

Figure A.4: Comparison between feeding TPE with best and worst available pipelines in the history

## A.3 Largest and Smallest Accuracies

Figure A.4 represents the warm-starting the TPE algorithm with best-performing pipelines (left column) and worst-performing pipelines (right column). This graph shows the result

for one-time exaction and reports the TPE algorithm's behavior with different trials.