



Ca' Foscari  
University  
of Venice

Department of Environmental Sciences,  
Informatics and Statistics

**Bachelor's Degree in Computer Science**

Final Thesis

# **Dash AutoML Benchmark**

**Supervisor**

Prof. Claudio Lucchese

**Graduand:** Riccardo Zuliani

**Matriculation Number:** 875532

**Academic Year:** 2020/2021



# Dash AutoML Benchmark

Riccardo Zuliani 875532

## Abstract

Nowadays, machine learning algorithms are used in every application domain like economics or even agriculture. This technology plays a crucial role to manage the massive amount of data collected, which increases exponentially day by day. In general, the process to create complete and effective machine learning algorithms requires a high level of domain knowledge and experience.

Since this is not always feasible, we need a way to automate the creation of machine learning algorithms or part of them. This type of technique is called Automated Machine Learning, also known as AutoML.

In this thesis we present an overview for the state-of-the-art of AutoML with a focus on the component that creates the pipeline for an algorithm, e.g., data pre-processing, feature engineering, model selection and hyperparameter optimization.

Furthermore, we provide a comparison between five of the most famous AutoML tools: AutoSklearn, TPOT, H2O, MLJAR-Supervised and AutoGluon. We analyze the structure and what are the differences between them, focusing on the way they approach the problem. Note that these five frameworks are all open source thus they are supported by the community of data scientists.

Moreover, we discuss the Dash application used to benchmark the five tools mentioned above, with a description of the key concept choices and why they were the best between the options, such as why I decided to use the Dash framework and what was the initial idea of the application.

And finally, we discuss the benchmark results from running the application on 8 Kaggle dataset and 24 OpenML dataset. The summary analyses the boxplots of each algorithm by comparing the prediction score and the amount of run time given to the algorithm to create the best possible pipeline for its problem. As a result, we indicate the best overall tool for classification problems and regression problems.



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>AutoML</b>	<b>3</b>
2.1	Problem Formulation . . . . .	4
2.2	Automatic Data Cleaning . . . . .	6
2.2.1	AlphaClean . . . . .	7
2.2.2	HoloClean . . . . .	7
2.3	Automated Feature Engineering . . . . .	8
2.3.1	Feature Tools . . . . .	10
2.3.2	Cognito . . . . .	10
2.4	CASH Optimization . . . . .	12
2.4.1	Grid Search . . . . .	13
2.4.2	Random Search . . . . .	13
2.4.3	Bayesian Optimization . . . . .	13
2.4.4	Genetic Algorithm . . . . .	14
2.4.5	Multi-armed Bandit Learning . . . . .	15
2.4.6	Gradient Descent . . . . .	15
2.5	Performance Improvements . . . . .	16
2.5.1	Multi-fidelity Approximations . . . . .	16
2.5.2	Early Stopping . . . . .	16
2.5.3	Scalability . . . . .	16
2.5.4	Ensemble Learning . . . . .	17
2.5.5	Meta-Learning . . . . .	18
<b>3</b>	<b>SOTA AutoML Algorithms</b>	<b>21</b>
3.1	AutoSklearn . . . . .	21
3.2	TPOT . . . . .	23
3.3	H2O . . . . .	24
3.4	MLJAR-Supervised . . . . .	25
3.5	AutoGluon . . . . .	26
<b>4</b>	<b>Dash AutoML Benchmark</b>	<b>29</b>
4.1	Application Overview . . . . .	29
4.2	Goals & Choices . . . . .	30
<b>5</b>	<b>Benchmarks and Results</b>	<b>33</b>
5.1	Datasets . . . . .	34
5.2	OpenML Benchmarks . . . . .	35
5.2.1	Smaller Datasets . . . . .	35

---

5.2.2	Medium-sized Datasets . . . . .	36
5.2.3	Large Datasets . . . . .	37
5.3	Kaggle Benchmarks . . . . .	38
5.3.1	Classification Problems . . . . .	38
5.3.2	Regression Problems . . . . .	39
<b>6</b>	<b>Conclusion</b>	<b>41</b>
<b>Appendix A</b>	<b>Application Screenshots</b>	<b>49</b>

# List of Figures

2.1	ML Pipeline . . . . .	3
2.2	AutoML Pipeline . . . . .	3
2.3	Typical data cleaning pipeline . . . . .	6
2.4	Probabilistics Problem Formulation . . . . .	7
2.5	Iterative feature generation procedure . . . . .	8
2.6	A Taxonomy of Meta-Learning Techniques . . . . .	18
3.1	AutoSklearn Pipeline . . . . .	21
3.2	Configuration Space of AutoSklearn. Squared box denote parent hyperparameters, rounded box represent leaf hyperparameters and grey colored box mark active hyperparameters . . . . .	22
3.3	TPOT Pipeline . . . . .	23
3.4	AutoGluon Multi-Layer Stacked Ensemble . . . . .	26
5.1	Results for smaller datasets . . . . .	35
5.2	Results for medium datasets . . . . .	36
5.3	Results for bigger datasets . . . . .	37
5.4	Kaggle Classification datasets results . . . . .	38
5.5	Boxplots of algorithms execution times for Kaggle Classification datasets . . . . .	38
5.6	Kaggle Regression datasets results . . . . .	39
5.7	Boxplots of algorithms execution times for Kaggle Regression datasets . . . . .	39
A.1	Screenshot of the OpenML Benchmark Interface . . . . .	49
A.2	Screenshot of the Kaggle Benchmark Interface . . . . .	50
A.3	Screenshot of the Test Benchmark Interface . . . . .	50
A.4	Screenshot of the Analysis of Past Benchmark . . . . .	51
A.5	Screenshot of the pipeline view . . . . .	51





# List of Tables

3.1	Summary of Algorithms Characteristic . . . . .	28
5.1	OpenML Datasets . . . . .	34
5.2	Kaggle Datasets . . . . .	34



# Chapter 1

## Introduction

Right the beginning of the millennium the machine learning field has been a hot topic involving lots of researchers. This is why a huge amount of sophisticated systems came to live. Such as Tesla Autopilot<sup>1</sup> and Google Waymo<sup>2</sup>, two self driving system which use Neural Architecture Search (NAS)[2], a type of automated machine learning that we is not discussed in this thesis. Or even the system that provide maintenance in the era of Industry 4.0. Or even AlphaGo, a computer program that plays Go, the oldest game board ever. It combines advanced search trees with deep neural networks that allow it to defeat a world champion Go player[7].

To build such advanced ML pipelines, the ML research team must be very comprehensive. Most of the time is established by a group of data scientists who have statistics and ML knowledge background and a group of domain experts, which have the experience needed to redirect the wrong data scientists choices made due to their lack of experience. Together these two groups have all that is needed to create a very specialized and effective machine learning pipeline.

The problem is that this process is often complex by running it iteratively with several trials and errors, which sometimes involves the reboot from scratch. As a consequence building powerful ML pipelines is a long and expensive effort that involves lots of time and human resources.

The idea of AutoML came out when many data scientists, after solving lots of different kinds of problems, noticed a sort of similarity between some processes of pipeline creation. Thus they try to generalize as much as possible to avoid spending time on boring and constant tasks, like Hyperparameter Optimization, Feature selection, data cleaning and so on. The AutoML tools in brief allow to automate some process of building a ML pipeline or, even the full process by leaving the machine learning expert to focus on other tasks that require more time and concentration to make better solutions. But it also allows non-experts, who have recently become interested in this field, to provide a good solution to a problem without having the same experience as a senior data scientist.

It is important to note that AutoML is not a new trend. Already in the early of the 90s Unica Software released its suite of marketing automation software which was based on neural networks. In the mid 90s it introduced a Pattern Recognition Workbench, a software which uses an automated grid search to optimize model tuning again for neural networks. Three years later Unica with Group 1 Software

---

<sup>1</sup><https://www.tesla.com/AI>

<sup>2</sup><https://waymo.com>

released a tool for automating model selection over four different types of predictive models. Two other commercial attempts were made in the late 90s about automated predicting modeling. The first MarketSwitch<sup>3</sup> consisted of a solution for marketing offer optimization, which included an embedded “automated” predictive modeling capability. The second was KXEN<sup>4</sup>, a company founded in France in 1998, which built its analytics engine around an automated modeling technique called structural risk minimization. Late on in the early of 2010 the actual software analytics vendors, like SAS and IBM, added automated modeling features to their high-end product. SAS introduced SAS Rapid Modeler<sup>5</sup>, a set of macros implementing heuristics that handle tasks such as outlier identification, missing value treatment, feature selection and model selection, like a complete AutoML algorithm. Although these tools were not open-source, the first one that we have to mention is Auto-WEKA<sup>6</sup>. This software selects a learning algorithm from 39 available algorithms, including 2 ensemble methods, 10 meta-methods and 27 base classifiers. Since each classifier has many possible parameter settings, the search space is very large; the developers use Bayesian optimization to solve this problem [9]. Moreover other tools came to live like Auto-Sklearn, TPOT and H2O.ai which are discussed in depth in chapter 3.

In the next chapter we discuss the structure and the all components that form an actual machine learning algorithms with a focus on the mathematical aspects. Chapter 3 gives a summary on the pros and cons of the five algorithms chosen, while in chapter 4 we discuss the key concept of the application used to benchmark the five tools and finally in chapter 5 we present the result obtained by running the benchmarks application on different datasets with a different time life for each algorithm.

---

<sup>3</sup><https://www.experian.com/decision-analytics/marketswitch-optimization.html>

<sup>4</sup><https://www.kxen.com/>

<sup>5</sup><https://support.sas.com/resources/papers/proceedings10/113-2010.pdf>

<sup>6</sup><https://www.cs.ubc.ca/labs/beta/Projects/autoweika>

# Chapter 2

## AutoML

Before we jump into the AutoML world we have to overview the pipeline used by all data scientists to create a complete ML algorithm. As we can see from the image 2.1 the first step is cleaning the input dataset with multiple methods like, filling missing values, removing inconsistent data and applying one-hot encoding for the categorical features. Then the most useful features are selected and finally the model is trained returning the prediction.

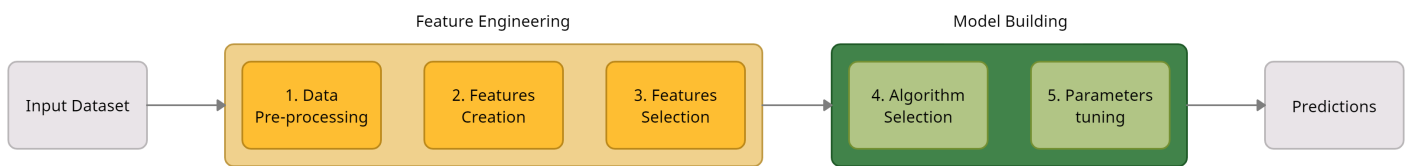


Figure 2.1: ML Pipeline

Instead, as shown in figure 2.2 all the AutoML frameworks aim to automate the process of creation without any human input, or at least, the least possible. Take for example the process of algorithm selection and parameters tuning, as the data scientist knows a bunch of algorithms how they work and their parameters, also the AutoML frameworks have knowledge on multiple of them, with the ability of shorter learning time.

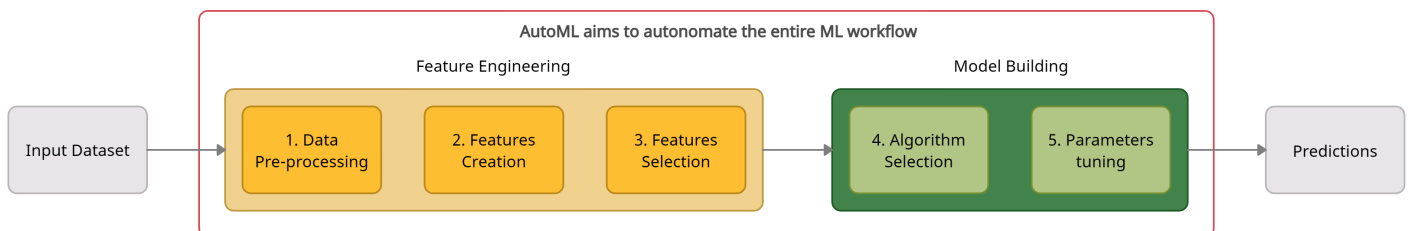


Figure 2.2: AutoML Pipeline

This let us introduce what are the aspects for this chapter, but before we start, we must first analyse the mathematical aspect of the composition of a machine learning pipeline.

## 2.1 Problem Formulation

An ML pipeline  $h : \mathbb{X} \rightarrow \mathbb{Y}$  is a sequential of numerous algorithms that transform a feature vector  $\vec{x} \in \mathbb{X}$  into a target value  $y \in \mathbb{Y}$ . Let a fixed set of basic algorithms be given as  $\mathcal{A} = \{\mathcal{A}^{(1)}, \mathcal{A}^{(2)}, \dots, \mathcal{A}^{(n)}\}$ . Each algorithm  $\mathcal{A}^{(i)}$  is configured as a vector of hyperparameters  $\vec{\lambda}^{(i)}$  from its domain  $\Lambda_{\mathcal{A}^{(i)}}$ . Without loss of generality, let a pipeline structure be modeled as a directed acyclic graph. Where each node form a basic algorithm and each edges rapresent the flow of data through different algorithms. Then let  $G$  denote the set of valid pipeline structures and  $|g|$  the length of the pipeline, where the number of nodes in  $g \in G$ .

All of this introduce the first definition of the chapter

**Definition 1 - Machine Learning Pipeline:** Let a triplet  $g, \vec{A}, \vec{\lambda}$  define an ML pipeline with  $g \in G$  a valid pipeline structure,  $\vec{A} \in \mathcal{A}^{|g|}$  a vector consisting of the selected algorithm for each node, and  $\vec{\lambda}$  a vector comprising the hyperparameters of the all selected algorithms. The pipeline is denoted as  $\mathcal{P}_{g, \vec{A}, \vec{\lambda}}$  [42]

Following the notation of the empirical risk minimization, let  $P(\mathbb{X}, \mathbb{Y})$  be a joint probability distribution of the feature space  $\mathbb{X}$  and target  $\mathbb{Y}$  known as *generative model*. So we denote a pipeline trained on the generative model  $P$  as  $\mathcal{P}_{g, \vec{A}, \vec{\lambda}, P}$

This bring us to the second definition:

**Definition 2 - True Pipeline Performance:** Let a pipeline  $\mathcal{P}_{g, \vec{A}, \vec{\lambda}}$  be given. Given a loss of function as  $\mathcal{L}(\cdot, \cdot)$  and a generative model  $P$  as  $\mathcal{P}_{g, \vec{A}, \vec{\lambda}, P}$ , the performance of  $P$  as  $\mathcal{P}_{g, \vec{A}, \vec{\lambda}, P}$  is calculated as

$$R(\mathcal{P}_{g, \vec{A}, \vec{\lambda}, P}, P) = \mathbb{E}(\mathcal{L}(h(\mathbb{X}), \mathbb{Y})) = \int \mathcal{L}(h(\mathbb{X}), \mathbb{Y}) dP(\mathbb{X}, \mathbb{Y}) \quad (2.1)$$

With  $h(\mathbb{X})$  being the prediction of  $P$  as  $\mathcal{P}_{g, \vec{A}, \vec{\lambda}, P}$  [42]

Now let a Machine Learning Task be defined by: a generative model, a loss of function and a ML problem type, like regression or classification. Generating an ML pipeline from a given ML task can be split into three tasks:

- Determining the pipeline structure, like how many preprocessing and feature engineering are needed, and how many models have to be trained
- Selecting an algorithm for each steps
- Selecting the corresponding hyperparameters for each algorithm selected

**Definition 3 - Pipeline Creation Problem** Let a set of algorithms  $\mathbb{A}$  with an according domain of hyperparameters  $\Lambda_{(\cdot)}$ , a set of valid pipeline structures  $\mathbb{G}$  and a generative model  $P(\mathbb{X}, \mathbb{Y})$  be given. The pipeline creation problem consist of finding a pipeline structure in combination with a joint algorithm and hyperparameters selection that minimize the following loss:

$$(g, \vec{A}, \vec{\lambda})^* \in \arg \min_{g \in \vec{\mathbb{G}}, \vec{A} \in \mathbb{A}^{|\mathbb{G}|}, \vec{\lambda} \in \Lambda} R(\mathcal{P}_{g, \vec{A}, \vec{\lambda}, P}, P) \quad (2.2)$$

[42]

In general equation 2.2 cannot be computed directly as the distribution  $P(\mathbb{X}, \mathbb{Y})$  is unknown. So we can define a set of observations  $D = (\vec{x}_1, y_1), \dots, (\vec{x}_m, y_m)$  of i.i.d samples draw from  $P(\mathbb{X}, \mathbb{Y})$  be given. Equation 2.1 can be adopted to  $D$  to calculate an empirical pipeline performance as

$$\hat{R}(\mathcal{P}_{g, \vec{A}, \vec{\lambda}, D}, D) = \frac{1}{m} \sum_{i=1}^m \mathcal{L}(h(x_i), y_i) \quad (2.3)$$

To limit the overfitting of last equation is often suggest to do cross-validation. So let the dataset  $D$  be split into  $k$  folds like  $D_{valid}^{(1)}, \dots, D_{valid}^{(k)}$  and  $D_{train}^{(1)}, \dots, D_{train}^{(k)}$  such that  $D_{train}^{(i)} = D \setminus D_{valid}^{(i)}$ . As result the final objective function is defined as:

$$(g, \vec{A}, \vec{\lambda})^* \in \arg \min_{g \in \vec{\mathbb{G}}, \vec{A} \in \mathbb{A}^{|\mathbb{G}|}, \vec{\lambda} \in \Lambda} \frac{1}{k} \sum_{i=1}^k \hat{R}(\mathcal{P}_{g, \vec{A}, \vec{\lambda}, D_{train}^{(i)}}, D_{valid}^{(i)}) \quad (2.4)$$

Using the equation 2.2 the pipeline creation problem is formulated as a black box optimization problem. Many algorithm aim on solve this problem, but is not possible predict any propriety of the loss of function or even formulate it as closed-form expression as it depends on the generative model.

On the other hand human data scientist experts usually solve the pipeline creation problem in a iterative way: first of all a simple pipeline structure with standard algorithms and default hyperparameters is selected and trained. Then the pipeline structure is adapted to the problem with the selection of new algorithms and hyperparameters to get better result.

## 2.2 Automatic Data Cleaning

Data cleaning is the first step to get into the machine learning pipeline and it is also one of the most annoying tasks. Almost 60% of the work of data scientists regards the cleaning and repairing of dataset records [35].

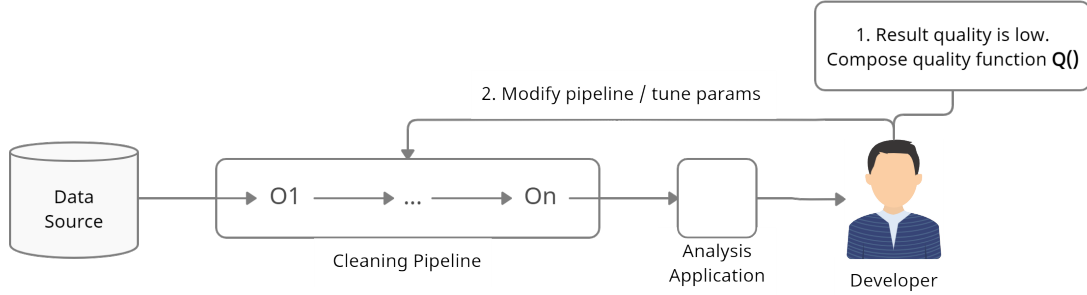


Figure 2.3: Typical data cleaning pipeline

Usually the cleaning process is represented as a pipeline (Figure 2.3). In the first step the user analyse the result and concluded it is not good as it must be (*error detection*), so he/she starts to compose a quality function to better analyse the quality issues and secondly modify the data cleaning pipeline to fix the errors (*data repairing*) [23].

The purpose of data cleaning is to improve the quality of the actual features by removing inconsistent records from the dataset. This process includes: removing redundant entries, filling missing values by some sort of function like median or mean, and removing outliers or feature columns that contain a single value or very few of them. When a predictor contains a single value, we call this a zero-variance predictor because there truly is no variation displayed, thus we simply remove it [29]. If it should happen to reach near-zero variance predictors or have the potential to have near zero variance during the resampling process, removal of these corresponding features is brought forth. These predictors have few unique values, (such as two values for binary dummy variables) and occur seldomly in the data, so they likely contain little valuable predictive information that we may not desire to filter out [29].

Data cleaning was one of the aspects that allowed the initial formation of the AutoML algorithms by generalizing the aspect cited before. Automating this process ensure many advantages, like free up availability for other task and process, increase data scientists confidence who can rest easy without the fear of human error, or even in the business aspect, since cleaning up data takes a long time is also expensive and therefore automating this process save a lot of money [36]. Yet most current approaches still aim to assist a human data scientist instead of fully automated data cleaning [42]. In the next pages we introduce two common semi-automated data cleaning tools.



### 2.2.1 AlphaClean

Sanjay Krshnan from the University of Chicago and Eugene We from the University of Columbia designed a framework called AlphaClean. Its main insight is to create a common intermediate reparation representation. It can facilitate more efficient data cleaning pipeline optimization, rather than treating the entire pipeline as a single parameterized black-box. Their goal is to develop a system to automatically generate and tune data cleaning pipelines based on user-specified quality characteristics. In brief AlphaClean improves the human-in-the-loop process shown in figure 2.3 by providing a quality function and automatically search for cleaning pipelines [23].

### 2.2.2 HoloClean

Another highly recommended tool for the semi-automated data cleaning is HoloClean founded by Theo Rekatsinas, Ihab Ilyas and Chris Ré. HoloClean is a data repairing framework that relies on statistical learning and inference to repair errors in structured data. The team build upon the paradigm of weak supervision and demonstrated how to leverage diverse signals, including user-defined heuristic rules (such as generalized data integrity constraints) and external dictionaries, to repair erroneous data [38].

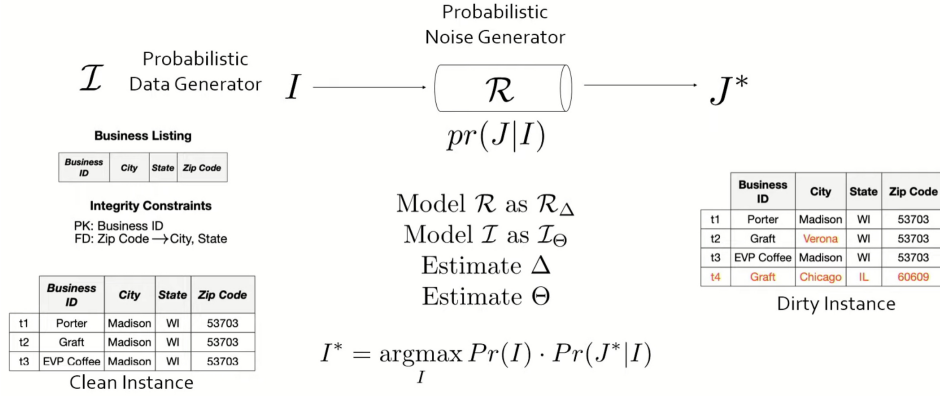


Figure 2.4: Probabilistics Problem Formulation

To answer the question "Why data cleaning is a probabilistic problem", we first assume there is a generative process that generates clean data and a noisy channel that can introduce error. What we observe is the work of the noisy channel. The Generative process  $\mathcal{I}$  generate a clean instance  $I$  which go through a bunch of devilish work  $\mathcal{R}$  and form the polluted version of  $I$ , called  $J^*$ , which is what we see. As such, if we model  $I$  and  $\mathcal{R}$  by parametrizing these two models and then try to estimate these two parameters, we can learn how the noisy channel works. For example we can repair the data by looking at the most probable instance that maximizes our observation, or try to find the probability that our pollution model attacks a specific cell (*error detection*) [16].

## 2.3 Automated Feature Engineering

Feature engineering, also known as feature creation, is the process of generation and selection of features from a given dataset. This step can be more crucial than the other, like the algorithm selection, because the machine learning algorithm learns only from the given data. So creating relevant features is absolutely crucial and doing it in the best way helps the model to perform better and learn as much as possible.

Like the data pre-processing in most ML applications the feature engineering task is taken by hand and it is relying on domain knowledge, intuition and experience. The result is almost depending both on the human subjectivity and time. Automated feature engineering aims to help the data scientist by automatically creating many candidate features out of a dataset, where the best can be selected and used for training [20].

Before jumping into some examples of automated feature engineering let's review the actual manual feature engineering process.

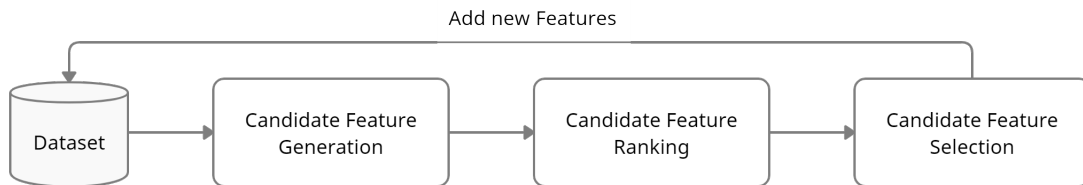


Figure 2.5: Iterative feature generation procedure

Data scientists and automated tools implement the generation and selection of features in an iterative way like Figure 2.5. Based on the initial features the process step into a generation and ranking of feature candidates, then only highly ranked features are added in the original dataset expanding the knowledge of the problem. This process is repeated until the human or in our case the automated algorithm do not find other relevant features.

As mentioned before, feature engineering is divided into feature generation and selection. The first phase is composed by the creation of multiple feature primitives which are splitted into two other categories: transformation and aggregation. Transformation acts on a single table by creating new features out of one or more existing columns. This category is divided also in other two sub category:

- **Unary transformation** operate in a single feature, by discretizing or normalising, like implementing MinMax Scaler or Standard Scaler, or applying logarithms or square root.
- **Binary transformation** combine two features in a new one via basic arithmetic operation. Usually these types of transformation consist in generation of correlation features.

Secondly feature selection is the ability to remove and choose the best relevant features. This step is fundamental for removing redundant and misleading features that could limit the goodness of fit. In all ML pipelines we want to have many instances and the least possible high ranked features. If this does not happen the

algorithm is going to be confused with the amount of dummy features to process, which affect the final prediction. In a brief the advantages of doing a good feature selection are:

- It enables the machine learning algorithm to train faster.
- It reduces the complexity of a model and makes it easier to interpret.
- It improves the accuracy of a model if the right subset is chosen.
- It reduces overfitting.

Algorithms like univariate selection, variance threshold, feature importance, correlation matrices or stability selection are already integrated in modern AutoML frameworks and selected via standard CASH methods (Section 2.4). In general the feature selection is implemented via three ways:

- **Filter methods** are generally used as a preprocessing step. Features are selected in basis on their score in a various statistical test for their correlation, e.g. [18]:
  - Pearson Correlation: is used as a measure for quantifying linear dependence between two continuous variables X and Y.
  - LDA: Linear Discriminant Analysis is used to find a linear combination of features that characterises or separates two or more classes of a categorical variable.
  - ANOVA: stands for Analysis of variance. It is similar to LDA except for the fact that it is operated using one or more categorical independent features and one continuous dependent feature.
  - Chi-Square: It is a statistical test applied to the groups of categorical features to evaluate the likelihood of correlation or association between them, using their frequency distribution.
- **Wrapper Methods** where we use a subset of features to train our model. Based on the inferences that we draw from the previous model, we decide to add or remove features from our subset. These methods search for the best feature subset by testing its performance on a specific ML algorithm [18].
  - Forward Selection: is an iterative method in which we start from having no feature and in each iteration we continue to add features trying to improve our model, till the new added decreases the goodness.
  - Backward elimination: in which we start from having all the features, and at each iteration we remove one by checking if the goodness of the model is improved until the elimination improves.
  - Recursive Feature Elimination: is a fusion between the Forward and the Backward Selection. We start from having all features or no one and at each step we check if from adding or removing features the model makes improvement or not.

- **Embedded Methods:** combine the qualities of filter and wrapper methods, they incorporate feature selection directly into the training process of an ML model. Many ML models provide some sort of feature ranking that can be utilized, such as SVM or random forests. [18]

Now we discuss two automated feature engineering tools that make the data scientists life easier.

### 2.3.1 Feature Tools

Featuretools<sup>1</sup> is an open source library designed to fast-forward the feature generation process, thereby it gives more time to focus on other aspects of machine learning model building [17]. This tool pays attention to the application of automated feature engineering in real world cases. In other words, usually when we have to solve a machine learning problem, the customers give us not a single table but a set of them linked via 1:1, or 1:n, or even n:n relationship. This situation causes confusion in every data scientist by the amount of data and all the possible connections between tables. This is why there is a need for tools that can join all tables in a single one and create relevant features to better describe the problem. Featuretools is one of these tools and it is built from three major components:

- Entity: which it can be considered as a Pandas DataFrame.
- Deep Features Synthesis or DFS: which is a feature engineering method that enables the creation of new features from single or multiple datasets.
- Feature Primitives: which are simple operations applied in a feature column.

The key concept of Featuretools is the DFS, in this method the user has to specify the tables with its primary key, the relationship between them and the table where all generated new features are inserted. With these three parameters Featuretools is able to create advanced custom features using the primitive once. It handles also the relationship between different type of features. [3]

### 2.3.2 Cognito

Cognito is an automated feature engineering tool that explores various feature construction choices in a hierarchical and non exhaustive manner, while progressively maximizing the accuracy of the model through a greedy exploration strategy. Additionally, the system allows users to specify domain or data specific choices to prioritize the exploration. Cognito is capable of handling large datasets through sampling and built-in parallelism, well integrated with a state-of-the-art model selection strategy. [19]

Cognito algorithm can be described as a greedy incremental search approach, where underlying this procedure is a transformation graph. Given a random DAG where the route node represents the input dataset and the weight represents the accuracy measure, each node represents a version of the dataset and each edge represents a data transform performed on the dataset. The transformation can be either unary or binary and affects all the same type of columns. Cognito for each

---

<sup>1</sup><https://www.featuretools.com/>

problem builds a transformation graph by performing transformations on the data. It also evaluates these proposed new features to see if they improved the performance of the model, while it performs the feature selection. The pro of this aspect is the performance of feature selection during generation of the transformation graph, as result this limits the explosive growth of columns. But the con is that it could discard features where additional transformation might have yielded useful features. [28]

But how can we find the best accuracy node? We have multiple strategies like death-first search or breadth-first search, but none of them are efficient. A solution is a combination of these two, such as Reinforcement Learning which outperforms the two previous approaches by not exploring as many nodes. [27]

## 2.4 CASH Optimization

The CASH problem is the abbreviation of *Algorithm Selection and Hyperparameter Optimization*, a notion introduced by Chris Thornton, one of the four researchers to create Auto-WEKA<sup>2</sup>. The CASH method instead of selecting an algorithm first and optimizing its hypermatrameters later, it selects both of them simultaneously. There are two model selection and hyperparameter optimization short-cut:

- Use a popular algorithm, but it can be challenging to make the right choice when faced with these degrees of freedom, like leaving many users to select algorithms based on reputation or intuitive appeal. As result, this approach yields performance far worse [39].
- Sequentially transform data, select models and model hyperparameters. Then test a suite of algorithms with default hyperparameters, select one or a few of them that perform well, and tune the hyperparameters of those top-performing models [5].

Anyway there are two big downside in the AutoML field [15]:

1. No single machine learning method performs best on all datasets.
2. Some machine learning methods (e.g., non-linear SVMs) crucially rely on hyperparameter optimization.

Selecting the best machine learning tool with its hypermaramenters is a search problem, in fact it can be viewed as a single hierarchical hyperparameter optimization problem. Where even the choice of algorithm itself is considered a hyperparameter [15]. At this point we introduce the mathematical aspect of the CASH problem.

Let a structure of valid pipeline  $g \in G$ , a loss of function  $\mathcal{L}$  and a dataset  $D$  be given. For each node in  $g$  an algorithm has to be selected and configured via hyperparameters. This problem is formulated as a black box optimization problem, leading to a minimization problem similar to the pipeline creation problem of equation 2.2. Firstly we assume the length of the pipeline is constant  $|h| = 1$  [42]. The CASH problem is defined like so:

$$(\vec{A}, \vec{\lambda})^* \in \arg \min_{\vec{A} \in \mathcal{A}, \vec{\lambda} \in \Lambda} R(\mathcal{P}_{g, \vec{A}, \vec{\lambda}, D}, D)$$

As said before, the additional choice of the algorithm is translated as an additional categorical meta-hyperparameter  $\lambda_r$ . So the total hyperparameter space for an algorithm is:

$$\Lambda = \Lambda_{A^{(1)}} \times \Lambda_{A^{(n)}} \times \Lambda_r$$

This define the final CASH minimization problem as:

$$\vec{\lambda}^* \in \arg \min_{\vec{\lambda} \in \Lambda} \frac{1}{m} \sum_{i=1}^m \mathcal{L}(h(x_i), y_i) \quad (2.5)$$

<sup>2</sup><https://www.cs.ubc.ca/labs/beta/Projects/autoweka/>

For example if we take the *it* algorithm, only its own hyperparameters  $\Lambda_{A(i)}$  are relevant as all the others do not influence the result. Therefore,  $\Lambda_{A(i)}$  depends on  $\lambda_r = i$ . Moreover the hyperparameters  $\vec{\lambda} \in \Lambda_{A(i)}$  can be divided into mandatory and conditional hyperparameters, where the last one depends on the selected value of another hyperparameter [42].

In these other subsections we discuss the most popular methods for the CASH optimization problem to solve Equation 2.5.

### 2.4.1 Grid Search

The peculiarity of this method is exploring all possible configurations. As the name says, the grid search creates a grid of configuration and evaluates all of them to choose the best one. It has two major drawbacks: it is a heavy CPU process from brute force searching and it does not consider the hierarchical hyperparameter structure. But the advantage is we surely get the best hyperparameters setting for the chosen algorithm.

### 2.4.2 Random Search

However, as the values to test increase, the number of models to be tested increase exponentially and therefore also the time to obtain the best result. The alternative choice is to use Random Search. In this method we replace the exhaustive search with a given number of searches. With this random number RS make  $n$  sample for each selected hyperparameter. The pro of this strategy is the shorter time to obtain the setting rather than grid search but the con is that maybe the given number of samples is insufficient to find the best set-up.

### 2.4.3 Bayesian Optimization

In one sentence, Bayesian Optimization builds a probability model of the objective function and uses it to select the most promising hyperparameters to evaluate the true objective function [21].

As we now know the hyperparameter optimization problem is the process of finding the best hyperparameter to return the top prediction. This problem can be formalized more easily rather than equation 2.5 as follow:

$$x^* = \arg \min_{x \in \mathcal{X}} f(x) \quad (2.6)$$

Here the  $f(x)$  represents an objective score to minimize evaluated on the validation set,  $x^*$  is the set of hyperparameters that yields the lowest value of score, and  $x$  can take any value from  $\mathcal{X}$ . In simple terms, we want to find the model hyperparameters that yield the best score on the validation set metric [21].

We can apply Grid Search or even Random Search with pros and cons, but both of them are completely *uninformed* about the past evaluation, and as result they spend lots of time evaluating insignificant and bad hyperparameters, only because they have to complete the computation.

As cited at the start of the section, Bayesian strategy is based on a probabilistic model, mapping hyperparameters to a probability of score on the objective function:

$$P(\text{score} | \text{hyperparameters}) \quad (2.7)$$

This equation is called *surrogate* for the objective function and it is much easier to optimize rather than the objective function  $f(x)$ .

Bayesian methods work by finding the next set of hyperparameters to evaluate on the actual objective function by selecting hyperparameters that perform best on the surrogate function. The steps for applying Bayesian Optimization are the follow [21]:

1. Build a surrogate probability model of the objective function.
2. Find the hyperparameters that perform best on the surrogate.
3. Apply these hyperparameters to the true objective function.
4. Update the surrogate model incorporating the new results.
5. Repeat steps 2–4 until max iterations or time is reached.

What we have discussed now is the general idea of Bayesian Optimization. One of the implementations and formalizations of this theorem is the Sequential model-based optimization method or SMBO. The sequential word refers to running trials one after another, each time trying better hyperparameters by applying Bayesian reasoning and updating the surrogate. This method has three main types of surrogate model:

1. **Gaussian Process:** the traditional surrogate models for Bayesian Optimization. The key idea is that any objective function  $f$  can be modeled using an infinite dimensional Gaussian distribution. But the drawback is its runtime complexity of  $\mathcal{O}(n^3)$  [42].
2. **Random Forest Regression:** use recursive splitting of the training data to create groups of similar observations. Random forests are fast to train and even faster at evaluating new data obtaining a good predictive power [42].
3. **Tree Parzen Estimators:** replaces the generative process of choosing parameters from the search space in a tree like fashion with a set of non parametric distributions [26]. This method builds the surrogate model in a different way, it apply the Bayes rule and instead of representing the surrogate model like 2.7, it uses:

$$p(x|y) = \frac{p(x|y) \times p(y)}{p(x)} \quad (2.8)$$

where  $p(x|y)$  is the probability of the hyperparameters given the score on the objective function [21].

#### 2.4.4 Genetic Algorithm

Genetic Algorithms or GA are a type of Evolutionary Algorithm inspired by the process of natural selection of Darwin [14]. The general idea of GA is to transform a population (set) of individuals objects, each with an associated fitness value, into a new generation of population using the Darwin process. Each individual in the population represents a possible solution to a given problem. The genetic algorithm attempts to find the best possible solution by genetically breeding the population of individuals over a series of generations [22].



### 2.4.5 Multi-armed Bandit Learning

Multi-Armed Bandit (MAB) is a Machine Learning framework where an agent has to select actions (arms) in order to maximize its cumulative reward in the long term. In each round, the agent receives some information about the current state (context), then it chooses an action based on this information and the experience gathered in previous rounds. At the end of each round, the agent receives the reward associated with the chosen action [37].

The classic example is using one-armed bandits like  $k$  slot-machines. Here we need to find the best possible machine to maximise our income while not losing lots of money. In general, trying each machine once and then choosing the most profitable is not a winning strategy, because the chosen machine could be in general unlucky. Instead the agent should choose an unperformed machine in order to more understand the goodness of it. This is a classical application of the Multi-armed Bandit.

### 2.4.6 Gradient Descent

Gradient Descent is an optimization algorithm for finding a local minimum of a differentiable function. Gradient descent is simply used in machine learning to find the values of a function's parameters (coefficients) that minimize a cost function as much as possible [10]. Given a  $f(x)$ , we select a random starting point called  $x_0$ , then we calculate the gradient/derivative of the function, and we set the *learning\_rate*, which is a constant to determine how far we move on on each iteration of the process. Learning rate is a delicate parameter, because high values not allow obtaining the local/global minimum and too small values have the effect of getting the minimum at a more time consumption rate. *Learning\_rate* setting must be done at just the right point, not too high and not too low, like the common *0.01*. The equation to apply at each step to get into the local/global minimum is the following:

$$x_{i+1} = x_i - \text{learningrate} \times \frac{d}{dx}[f(x)]$$

## 2.5 Performance Improvements

In the previous section we presented the components and techniques to create a machine learning pipeline. Moreover we discuss different performance improvements to help the optimization procedure and the quality of the resulting prediction.

### 2.5.1 Multi-fidelity Approximations

The problem with the AutoML CASH Optimization is the amount of hours or even days to complete the computation. A common method to solve this problem is the usage of multi-fidelity approximations. Data scientists usually do not use all dataset and/or features but only a subset of them. This training step eliminates bad and nonperforming configurations, making sure only the good once is tested in the entire dataset [42]. An example of this strategy is the Successive Halving. It improves Random Search by dividing and selecting randomly generated hyperparameter configurations more efficiently without making more assumptions about the nature of the configurations space. More efficient allocation of resources means that given the same set of configurations, Successive Halving finds the optimal configuration more quickly than in Random Search [8].

### 2.5.2 Early Stopping

Every tools that we discuss later on in chapter 3 implements k-cross-validation to stop the iteration of unpromising configuration, this technique avoids the overfitting of the problem. One of the most common strategies about k-fold-cross validation is to abort the iteration if the newest configuration is worse than the actual.

### 2.5.3 Scalability

As we now know the creation of a common machine learning pipeline is a high CPU process that could take a while. The modern AutoML frameworks allow to parallelize the process using the all amount of available cores from the machine. This is possible by adding the options `n_jobs=-1` on the tools that allow it. Or use a cluster like H2O.ai which creates a local cluster inside our machine which runs separately from the application, but later we shall go deeper within this field. The initialization of the cluster takes into account the creation of cloud based AutoML tools. They offer a web application where their customers can update the dataset and create a performing pipeline for problem solving. In this type of cloud based algorithms we can mention Google Cloud AutoML<sup>3</sup>, Amazon SageMaker<sup>4</sup> and Microsoft Azure Machine Learning<sup>5</sup>. All these tools are not discussed in this thesis.

---

<sup>3</sup><https://cloud.google.com/automl>

<sup>4</sup><https://aws.amazon.com/it/sagemaker/>

<sup>5</sup><https://azure.microsoft.com/en-us/services/machine-learning/>

### 2.5.4 Ensemble Learning

A popular concept in the ML field is ensemble learning, a strategy combination of multiple ML models for better predictions. Actually the classic ML and AutoML frameworks during their computation time create lots of well performing configurations. And instead of taking the best possible one, we take a set of best configurations to create an ensemble [42]. There are three major ensemble techniques: Bagging Ensemble, Boosting Ensemble and Stacking ensemble.

#### Bagging Ensemble

It is an ensemble method based on multiple sampling of the dataset. In other words different samples of the same data are trained to each model, and then the results are combined into one set using classical statistics methods. The key point is the bootstrap sample with curly replacement, the replacement word means if a row is selected, is returned for a potential re-selection in the same training dataset. This ensures the single row can be selected multiple times or not even once.

#### Boosting Ensemble

It is an ensemble method which pays more attention to the incorrect fitted model. The key property of boosting is to try to make better predictions where previous models have failed. The three major properties are:

- Bias training data toward those examples that are hard to predict.
- Iteratively add ensemble members to correct predictions of prior models.
- Combine predictions using a weighted average of models.

The idea of combining many weak learners into strong learners was first proposed theoretically and many algorithms were proposed with little success [4]. In the long run, it does not leave much time to develop the Adaptive Boosting of AdaBoost, since many other Boosting methods came to life like the Stochastic Gradient Boosting with XGBoost.

#### Stacked Ensemble

It involves combining the predictions from multiple machine learning models on the same dataset. But unlike bagging the models trained are different from each other, and unlike boosting a single model is used to learn how best to combine the predictions from the contributing models. The architecture of a stacking ensemble involved multiple models/layers [6]:

- Base-Models: where the model fit on the training data.
- Meta-Model: where the model learns how to best combine the predictions of the base models.

The predicted output of each model/layer is appended as a new feature to the training data of the next layer. So the newest layer can perform better than the previous [42].

### 2.5.5 Meta-Learning

Meta-Learning can be described as the process of learning from previous experience gained during applications of various learning algorithms on different kinds of data [11].

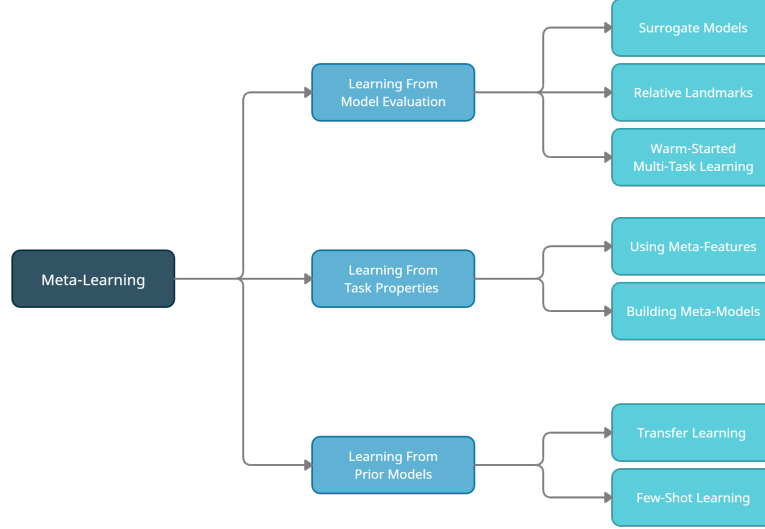


Figure 2.6: A Taxonomy of Meta-Learning Techniques

As Figure 2.6 indicates, these techniques can be categorized into three main groups: *learning based on task properties*, *learning from previous model evaluations* and *learning from already pretrained models*.

Starting from *Task Properties* we can use the *meta-features* to characterize a particular dataset, where each task is referred to a feature vector. In brief information from a previous task can be injected to a new task based on their similarity. Where the similarity can be calculated on the distance of their feature vectors, like the Euclidean Distance. Some meta features commonly used to describe the dataset have simple functionality based on the number of instances, like classes, missing values or outliers, or more complex such as PCA, correlation, covariance or ANOVA. [11, 41] The next strategy is *Building our Meta-Models*, this process aims to build a meta model to learn the complex relationship between meta features of prior tasks [11]. An example is *Ranking*, where we build the top-K most promising configuration, using K-nearest neighbors to predict what tasks are similar and rank them [41]. In the same group we can also mention another strategy called *Performance Prediction* where the meta-model can predict how good the configuration would be on a given task, using its meta-features. This provides a kind of shortcut for determining whether or not to evaluate a specific configuration [41].

The next group is *Learning from Model Evaluation*, where we get the hyperparameter recommendations for a new task and analyse similar prior tasks. We have three similarity approaches: *Relative Landmarks*, *Surrogate Models* and *Warm-Started Multi-Task Learning*. *Relative Landmarks* measures the performance between two models in the same task, where  $t_{new}$  and  $t_i$  are considered similar if only if their relative landmarks are similar. Learning from model evaluation can be done using a *Surrogate Model*, where models get trained on all prior evaluations of all prior tasks. In brief, if a surrogate model has good performance in both  $t_{new}$  and  $t_i$ , the

two tasks are considered similar [11]. The last for model evaluation is *Warm-Started Multi-Task Learning* where basically all the previous methods get an initialization phase where random configurations are selected [42].

The final learning method uses the *Previous Models*. In this group we can find *Transfer Learning* a process that makes use of pretrained models on a previous task  $t_j$  to be adapted on a new one  $t_{new}$ , where both of them are similar or not. It is preferable to use this method when the new task to be learned is similar to the prior tasks [11]. The other method belonging to this group is *Few-Shot Learning* which aims to build accurate machine learning models with less training data and prior experience, gained from already trained models on similar tasks [32, 11].

## Summary of Chapter 2

In this chapter we have discuss:

- In Section 2.1 we discussed the mathematical aspect of the creation of the machine learning pipeline, and found by using the equation 2.5 the pipeline creation problem is formalized as a black-box optimization problem.
- In Section 2.2 we discussed the process of data cleaning with its benefits if automated, in regards to time and cash saving. We then analysed two modern Auto Data Cleaning tools, HoloClean which formalized the cleaning process as a probabilistic problem and AlphaClean which used user-specified quality characteristics.
- In Section 2.3 we discovered the iterative feature generation procedure shown in figure 2.5, then we discussed the unary and binary category which divided the generation of features. Then we described three implementations of feature engineering, filter methods, wrapper methods and embedded methods. Ultimately we introduced two automated feature engineering tools. The first, Feature Tools focus on real-world problems with its ability to join multiple tables with corresponding identifiers and generate new features based on feature primitives. And the second, Cognito translates the problem of creating the best possible feature into a DAG search problem, where we have to find the most accurate node representing the top transformation of the dataset.
- In Section 2.4 we discussed the algorithm selection and hyperparameter optimization problem on the CASH problem. We discovered the basics of how model selection works, using a popular algorithm and its default parameters, or sequentially transforming data, select models and its hyperparameters. We later learned that the problem of finding the best possible combination of algorithm and hyperparameters can similarly be referred to as the pipeline creation problem. And we introduced the commons CASH optimization strategies like: Grid and Random Search, Bayesian Optimization, Genetic Algorithm, Multi-armed Bandit Learning and Gradient Descent.
- In Section 2.5 we discovered how we can improve our model by applying some common strategy. Multi-fidelity Approximations uses only a subset of training data and/or features to eliminate bad and unperforming configurations, an

example of this method is Successive Handling. Early Stopping aborts the iteration of the k-fold-cross validation if the newest configuration is worse than the actual. Scalability runs the evaluation model in parallel in the most possible CUP cores, or scales it using clusters. Furthermore we introduced three Ensemble Learning methods: Bagging, Boosting and Stacked Ensemble. Bagging Ensemble makes better predictions by training each model in different samples of the same dataset, and then combines the results. Boosting Ensemble pays more attention to bad models to obtain better solutions where previous models have failed. Stacked Ensemble is based on multiple layers/models where each of them appends the result to the next one to ensure it performs better.

The last introduced strategy to increase model performance is Meta-Learning. It avoids most of the trial of a task by learning based on observation of various previous ML tasks. As shown in Figure 2.6, this technique is divided into three main category: Learning from Model Evaluation where we can find Relative Landmarks, Surrogate Models and Warm-Started Multi-Task Learning, Learning from Task Properties which is splitted into using Meta-Features and build new Meta-Models, and Learning from Previous Models where we have Transfer Learning and Few-Shot Learning.

# Chapter 3

## SOTA AutoML Algorithms

In this chapter we analyse the five algorithms chosen to take into account. These are AutoSklearn<sup>1</sup>, TPOT<sup>2</sup>, H2O<sup>3</sup>, MLJAR-Supervised<sup>4</sup> and AutoGluon<sup>5</sup>.

### 3.1 AutoSklearn

AutoSklearn is the most common AutoML tool and it has the characteristic of being built on the well-known *scikit-learn* ML package. Starting from the cons it has no capability of feature engineering process, thus this is done only from user input specification. In other words, it requires the user input to convert data into integer before any other transformation, e.g., using label encoder. On the other hand it has two major pros, it resolves the CASH optimization problem with the SAMC3 to efficiently perform Bayesian Optimization [40]. SMAC3 is the Python implementation of the Sequential Model-Based Algorithm Configuration, a tool to optimize algorithm parameters [1]. The main core of SMAC3 consists of Bayesian Optimization in combination with a simple racing mechanism on the instances to efficiently decide which of two configurations performs better [25]. And secondly it uses pre-processed meta-features.

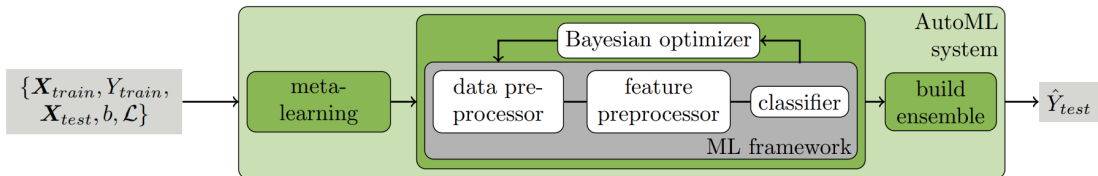


Figure 3.1: AutoSklearn Pipeline

As we can see from figure 3.1, AutoSklearn personalizes the classical AutoML pipeline of picture 2.2 by adding two components to the Bayesian hyperparameter optimization. The meta-learning step warmstart the Bayesian Optimization procedure and the automated ensemble step construct configurations evaluated during optimization [13].

<sup>1</sup><https://automl.github.io/auto-sklearn/master/>

<sup>2</sup><http://epistaslab.github.io/tpot/>

<sup>3</sup><https://docs.h2o.ai/h2o/latest-stable/h2o-docs/automl.html>

<sup>4</sup><https://supervised.mljar.com/>

<sup>5</sup><https://auto.gluon.ai/>

The meta-learning process is very quick in suggesting well performing features, but it is not able to provide detailed information on performance. On the other side Bayesian Optimization is slow to find the optimum hyperparameter for a given algorithm, but it can fine-tune performance over time. AutoSklearn implements a fusion of them by selecting  $k$  configuration of features by meta-learning and using their result to warmstart Bayesian Optimization [13].

As we know Bayesian hyperparameter optimization is one of the best tools for finding the most performing hyperparameters configuration, but it has the downside of being very wasteful if the scope of the problem is to make a good prediction. As so all models trained during the course of the search are lost. The AutoSklearn team decided to store all the models and use them to construct an ensemble with adjusted weight, using the prediction of all individual models. For the ensemble strategy they decided to use the *ensemble select* method [13].

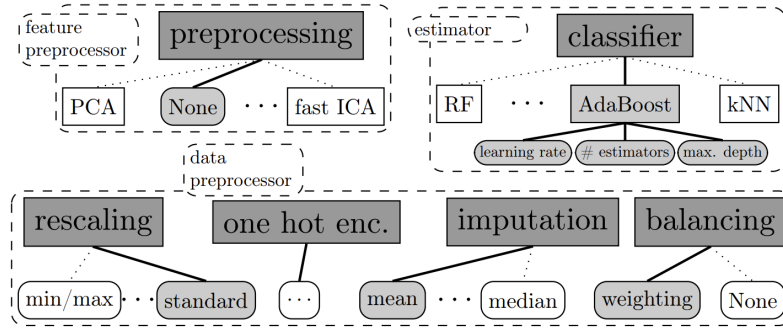


Figure 3.2: Configuration Space of AutoSklearn. Squared box denote parent hyperparameters, rounded box represent leaf hyperparameters and grey colored box mark active hyperparameters

AutoSklearn comprises 15 classification algorithms, 14 preprocessing methods and 4 data preprocessing methods, all of which are parameterized into 110 hyperparameters. In conclusion, unlike Auto-WEKA [39] AutoSklearn configuration space is focused on base classifiers and excluded meta-models and ensembles that themselves are parameterized by one or more base classifiers [13].



## 3.2 TPOT

TPOT, the abbreviation of Tree-based Pipeline Optimization Tool, automatically designs and optimizes machine learning pipelines using Genetic Programming a field that creates Genetic Algorithms 2.4.4. TPOT is a sort of wrapper of the common ML-package scikit-learn. Thus each machine learning pipeline operator and algorithm are from scikit-learn [30].

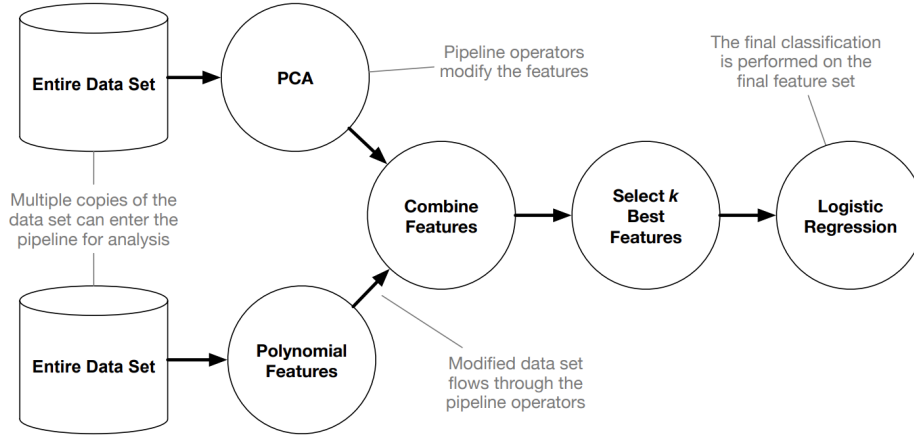


Figure 3.3: TPOT Pipeline

Figure 3.3 shows a tree-based pipeline example, where the Preprocessing, Decomposition, Feature Selection and Models Selection steps are combined and represented as a tree. Note that TPOT only accepts numeric features, so any data preprocessing and feature generation for those features must be done manually by the user [40]. Every tree-based pipeline gets one or more copies of the input dataset as the leaves of the tree, which is then fed into one of the four processes cited before [31]. In order to operate, we store three additional variables for each record in the data set [30]:

- Class: indicates the true label for each record.
- Guess: indicates the pipeline’s latest guess for each record.
- Group: indicates whether the record is to be used as a part of the internal training or testing set.

As the data is passed through the tree, it is modified by the process of each node. When multiple copies of the dataset are processed, it is possible to combine them in a single dataset. Each time a dataset flows into the sequence of process/nodes, the resulting classification/regression are stored, such that the most recent model to process the dataset overrides any previous predictions. Once the data has finished the computation flow of the pipeline, the final predictions are used to evaluate the overall performance of the pipeline [31]. In more detail, the GP algorithm forming TPOT works as follows: It initially generates 100 tree-based random pipelines and evaluates their balanced cross-validation score in the dataset. Subsequently, for each generation, the GA selects the first 20 pipelines from the populations, simultaneously maximizing the score and minimizing the number of operations in the pipeline. In every generation, the algorithm updates a Pareto section of the non-dominated solution at any point in the GP run. In conclusion the algorithm repeats

this evaluate-select-crossover-mutate process for 100 generations until it selects the highest scored pipeline from the Pareto section. This represents the best pipeline from the run [30].

### 3.3 H2O

H2O AutoML is a fully automated supervised learning algorithm implemented in H2O. H2O AutoML is available in Python, R, Java and Scala as well as through a web GUI. Though the algorithm is fully automated, many of the settings are exposed as parameters to the user, so that certain aspects of the modeling steps can be customized [24]. Regarding data pre-processing, H2O includes automatic imputation, normalization and one-hot encoding, and it supports group-splits on categorical variables. In addition it implements automatic text encoding using Word2Vec<sup>6</sup>, as well as feature selection and feature extraction for automatically dimensionality reduction.

H2O AutoML provides a vast sort of algorithms available for consultation via the link in the footnote<sup>7</sup>. For each algorithm, we identify which hyperparameters we consider to be *most important*, define ranges for those parameters and utilize random search to generate models [24]. The pre-specified models are included to give quick, reliable hyperparameters for each algorithm. The order of the trained algorithms can be customized by the user, however one of the first to be trained are the XGBoost models followed by GLMs for a quick reference point. From here is prioritized increasing diversity across our set of models introducing Random Forest, GBM and Deep Learning models. After this set of algorithms is trained and added to the leaderboard H2O starts a random search between those same algorithms [24].

After training the base models, two Stacked Ensemble models are trained using the H2O Staked Ensemble algorithm. The All models ensemble contains all the models, and the Best Family ensemble contains the best performing model from each algorithm class/family. The Best of Family ensemble is optimized for production cases since it only contains six base models and can generate predictions rapidly compared to the All Models ensemble [24]. By default, the meta-learner in the Stacked Ensemble is trained using k-fold cross-validated predictions from the base learners [24].

As mentioned on the Section 2.5 one of the methods to scale the AutoML algorithms is the use of clusters. H2O AutoML offers the "H2O Cluster", which is the Java process running H2O across one or many computers, in our case on only a single machine. The first step when working with H2O is initializing a cluster, remote or locally, it does not matter which. Once the cluster is built and running, data can be imported from the local storage into a special dataframe called "H2O Frames", also a part of the training is performed inside the H2O cluster [24].

---

<sup>6</sup><https://docs.h2o.ai/h2o/latest-stable/h2o-docs/data-science/word2vec.html>

<sup>7</sup><https://docs.h2o.ai/h2o/latest-stable/h2o-docs/data-science.html>

## 3.4 MLJAR-Supervised

MLJAR-Supervised is the only tool to have no official academic papers published, so everything below is from the correlated GitHub Repository or documentation [33, 34]. Firstly, MLJAR like AutoSklearn (3.1) is an AutoML tool based on the popular ML package "scikit-learn". Secondly MLJAR is a well formed tool for beginners because it does not implement the black box optimization problem, because it allows to consult every type of configuration tested on each kind of process, by simply reading the markdown files stored in the directory. These files identify the tested algorithm, or consult the stored graphs in the same folder.

Now we describe how the MLJAR pipeline works to get the best ML model of a problem.

1. The first step of the AutoML training process is to check the simplest algorithms to get quick insights. Baseline models provides baseline results, like the most frequent label from a classification task or the mean of the target for a regression task. Decision Tree provides result for simple tree and Linear provides simple ML model.
2. Next the Default Algorithms, e.g. Random Forest, Extra Trees, XGBoost, LightGBM, CatBoost, are trained with its standard hyperparameters to return a model.
3. MLJAR computes a Random Search on the hyperparameters of the algorithms specified at the previous step.
4. It computes a process of feature generation<sup>8</sup> followed by a retrain of XGBoost, CatBoost and LightGBM.
5. Then random features<sup>9</sup> are added to the original data, then the best model is selected with its hyperparameters and used to train the model with the newest features.
6. The best model for each algorithm cited in step 2 is selected and its hyperparameters are reused for training using only the selected features. Note that if all features are important this step is skipped.
7. A process of tuning of the hyperparameters of each algorithm takes forth. This creates new models with already used algorithms but with different limit of scanning, e.g. an XGBoost model with `max_depth=5` and another with `max_depth=10`.
8. In the end the Stacked Ensemble process is computed where all models from previous steps are ensembled.

---

<sup>8</sup>[https://supervised.mljar.com/features/golden\\_features/](https://supervised.mljar.com/features/golden_features/)

<sup>9</sup>[https://supervised.mljar.com/features/features\\_selection/](https://supervised.mljar.com/features/features_selection/)

### 3.5 AutoGluon

At this stage we introduce AutoGluon, a powerful AutoML Python tool based, with the capability of automatically recognizing the data type in each column for robust data preprocessing, including special handling of text fields. AutoGluon preprocessing task relies on two sequential stages: *model-agnostic* preprocessing to transform the input to all models, and *model-specific* preprocessing applied only to a copy of the data used to train a particular model. Model-based preprocessing starts by categorizing each feature: numeric, categorical, text or datetime, then it discards the uncategorized columns. Instead the text features are considered as columns of mostly unique strings, which averagely contains more than 3 non-adjacent strings. The values of each column are transformed into numeric vectors of  $n$  features. Datetime features are also transformed into numeric values. The missing discrete variables are marked by AutoGluon with the *unknown* category. Furthermore a copy of the resulting set of numerical and categorical features is passed to the model-specific methods for more in depth preprocessing [12].

AutoGluon uses a customized set of models. This ensures that reliability performance models such as random forest are trained prior to more expensive and less reliable models like k-nearest neighbors. The algorithms taken into account are LightGBM, CatBoost, Random Forest, Extremely Randomized Trees and k-Nearest Neighbors [12].

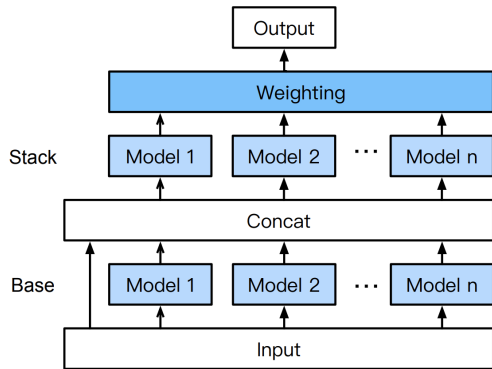


Figure 3.4: AutoGluon Multi-Layer Stacked Ensemble

AutoGluon makes an intense use of stacking ensembles. It introduces a multi-layer stack ensemble from as shown in figure 3.4. In the first layer we have multiple models, whose outputs are concatenated and then passed into the next layer, which in turn consists of multiple stack models. In addition to the classic stacking methods AutoGluon implements three changes to improve the resulting accuracy. To avoid another CASH problem it simply reuses all of the base layer model types, with the same hyperparameters, as stackers.

Thus the stacker models take as input not only the predictions of the models at the previous layer, but also the original data features themselves. As result the final stacking layer applies ensemble selection to aggregate the stacker models predictions in a weight manner [12].

AutoGluon furthermore improves its stacking performance by utilizing all of the available data for both training and validation, through k-fold ensemble bagging of all models done at the layers of the stack. There is variance reduction of the resulting prediction, because of random data partitioning into  $k$  chunks and substantially  $k$  copy training of a model with different chunk. AutoGluon bags all models and asks each one to produce out-of-fold (OOF) predictions on the unseen chunk during training [12]. Even so, k-fold bagging efficiently reuses training data, but it remains vulnerable from over-fitting. To migrate this problem, it implements a repeated bagging process, which works as follows. Given sufficient training time, AutoGluon

repeats the k-fold bagging process on a random partition of the training data and it averages all OOF predictions over the repeated bags. These predictions display less variance and are much less likely to be overfitted [12].

## Summary of Chapter 3

To sum up:

- **AutoSklearn:** has no feature engineering capability, it resolves the CASH Optimization problem with SAMC3 and implements meta-features
- **TPOT:** accepts only numerical features so any data preprocessing and feature generation have to be done by the user. It is represented as a tree-based pipeline, where every pipeline gets one or more copies of the input dataset like the leaves of a tree, which is then fed into one of the four processes. As the data is passed through the tree, it is modified by the process of each node. A dataset combination implementation process takes until the data has finished calculating, this combination is used for the final prediction.
- **H2O:** is a fully automated supervised learning algorithm, with a well formed data preprocessing step and a vast sort of algorithms, where the best hyperparameters search takes place. Pre-specified model training also takes place for a quick insight into the problem. Different algorithms run until two Stacked Ensembles are trained using the k-fold cross-validation. Note H2O implementation is scalable by the use of clusters.
- **MLJAR-Supervised:** is an AutoML tool based on scikit-learn which has the feature the feature of having no black box optimization problem. As so it allows to consult every type of configuration tested on each kind of process, through markdown files reading, or through viewing the graphs stored in directory identifying the tested algorithm.
- **AutoGluon:** has a well formed data preprocessing step with model-agnostic phase and model-specific phase. It is a high intense user of stacking ensembles which introduces a multi-layer stacked ensemble, the output of each layer is concatenated and passed to the next one. There is also strategy improvement through the use of all available data for both training and testing through k-fold ensemble bagging. Each model is asked to produce an out-of-fold prediction, followed by a repeated bagging process.

Tool	Release Date	Data Pre-Processing	Feature Engineering	CASH Strategy	Model Ensembling
AutoSklearn	2015	/	Only with Integer	Bayesian Optimization, Meta-Learning	Ensemble Selection
TPOT	2016	/	PCA, Polynomial Features, Missing Values	Genetic Programming	Stacking
H2O	2016	Numerical, Categorical, Datetime	Datetime, categorical preprocessing, imbalance, missing values, feature selection, reduction	Random Search	Stacking, Bagging
MLJAR-Supervised	2019	/	Golden Features <sup>1</sup> , Features selection	Random Search	Stacking
AutoGluon	2019	Numerical, Categorical, Datetime	Datetime, categorical preprocessing, imbalance, missing values, feature selection, reduction	Fixed Default (Set Adaptively)	Multilayer Staking, Repeated Bagging

<sup>1</sup> [https://supervised.mljar.com/features/features\\_selection/](https://supervised.mljar.com/features/features_selection/)

Table 3.1: Summary of Algorithms Characteristic

# Chapter 4

## Dash AutoML Benchmark

In this chapter we discuss the Automl application paying particular attention to the evolutions and choices that have allowed to arch achieve all goals.

### 4.1 Application Overview

Structure application analysis is done with the available interaction. The system runs locally and has been developed in Python 3.8.10. It consists of an open-source web application where the user can create benchmarks based on Kaggle dataset or OpenML dataset, with a complete analysis on scores and algorithms pipeline.

The two main actions are: run benchmarks on Kaggle or OpenML dataset and analysis done on the results of past benchmarks (Figure A.4). Analysis is done by consulting the scores tables and the initial and final execution times tables. Furthermore, with the use of Scatter Plots or Bar Plots we can have a graphic view of the same results shown in the scores tables. During verification, it is possible to check the models trained by each algorithm and the best pipeline found (Figure A.5), to obtain a better view of the algorithm power. The last possible interaction is the comparison between benchmarks with the same dataset tested but with different starting execution time, this is useful to understand the apprehension rate of each tool. But the most important task is obviously the benchmark for the five AutoML frameworks on Kaggle and OpenML dataset. Both benchmarks have the time life option and re-run option for each algorithm. The re-run is useful if the given time life is insufficient to get a pipeline, so the computation restarts with a higher amount. In the Kaggle benchmark (Figure A.2) we have the choice to choose from 8 kaggle competition dataset, 4 for each problem type. While in the OpenML benchmark (Figure A.1) we can type the sequence of known dataset IDs or set a unique number for both types of dataset problems we want to test and the minimum number of instances. Finally the last type is the Test Benchmark (Figure A.3), where a simple benchmark is run based on a dataset ID and a single algorithm or the all set.

For any doubts, please refer to the GitHub Repository<sup>1</sup>.

---

<sup>1</sup><https://github.com/zuliani99/AutoML-Benchmark>

## 4.2 Goals & Choices

Application goals have changed quite a bit during the development. The first one was to create a terminal application that allows the user to compare different AutoML algorithms on Kaggle and OpenML datasets with a complete analysis of pipelines and prediction scores. In this phase lots of major choices had been made. The two sources of dataset, Kaggle and OpenML, were chosen because with the first<sup>2</sup> we are able to run the five algorithms on competitions datasets, making the benchmark more interesting by the fact we can compare the result of the five tools with the first in the competition ranking. And secondly, OpenML<sup>3</sup> is a great platform for sharing datasets regarding real data, since the datasets have the advantages of being almost ready to be used, with the minimum preprocessing required. Next, the five algorithms were chosen after reading multiple academic papers about benchmarking automated machine tools, so I decided to try myself and see the result of all of them. Initially there were two different ones, Ludwig<sup>4</sup> and AutoKeras<sup>5</sup>. But during the tune of Ludwig and a posted issue in the GitHub repository, the creator informed me that Ludwig was not comparable with the other tools. In Ludwig we have a configuration and we find the hyperparameters with the hyperopt script, but it does not figure out the best configuration like the other tools do<sup>6</sup>. While Autokeras makes use of deep neural networks, a field not discussed in this thesis. After choosing the algorithms, I was faced with deciding what score function to use for the classification problems and regression problem. Initially there were only Accuracy for the classification and Root-Mean-Squared-Error (RMSE) for the regression. But after a consultation with the professor we agreed the best option was to add another score for each problem type. We have chosen F1-Score for classification problems to detect problems with imbalanced classes, and R2-Score for regression problems for having a better view of the score. On top of that, three other main technical decisions were taken in this step. Instead of downloading and installing every package by typing the name of each one, I have implemented a simple MakeFile in which are present two bash commands, one to download and the other to remove all the requirements from a text file. This approach is much easier than the Docker File, because it does not require any third party packages to create the image file and is easy to write. Regarding the installation of the package I use the Virtual Environment<sup>7</sup>. The major advantage had been the isolation of project packages from the global packages, resulting in no worries if installation errors occurs. So many package dependencies conflicts were resolved simply removing the entire project directory, re-downloading it with the packages version update and re-installing all of them. Besied that that I decide to use a .csv file rather than a database to store the benchmarks result of the classification and regression datasets. At that point the main objective was to try to simply implement all the features with no need for database storage. Finally the last choice made at that point was the structure of the storing score system. For each benchmark the application created a csv file for each scoring function and algorithm pipeline. Plus it created a csv file for the start and end of life time. This

---

<sup>2</sup><https://www.kaggle.com>

<sup>3</sup><https://www.openml.org/>

<sup>4</sup><https://ludwig-ai.github.io/ludwig-docs/>

<sup>5</sup><https://autokeras.com/>

<sup>6</sup><https://github.com/ludwig-ai/ludwig/issues/1145>

<sup>7</sup><https://docs.python.org/3/library/venv.html>



management made access to data very comprehensible and hierarchical.

Thus I decided to step into the next level and move the application from terminal interaction, where it is always tricky, to an easy to use web application. To implement this step I decided to use Dash Plotly<sup>8</sup> a popular framework based on Plotly.js and React.js to build data apps. I choose it rather than create the application from zero using Flask<sup>9</sup> or Django<sup>10</sup>, because Dash is already implemented on Flask and offers an easy to use framework to create dynamic web app, but secondly and most important it provides a vast library of function to create and personalized graph and table, which were lots useful in my case. This step was quite complicated because I was new to this tool. I spent plenty of time learning the basics of advanced methods like the Advanced and Pattern-Matching Callbacks.

As a result, I decided again to move the evolution forwards by trying to deploy the application on a cloud provider, e.g., Amazon Cloud Services, Microsoft Azure or Google Cloud. At that point my goal was to make the application public so that any data scientist, not necessarily an expert, could see with his own eyes which was the best AutoML tool for a given dataset.

In this stage I encountered multiple difficulties, first of all the budget cost. All the providers offer a weak base configuration of Virtual Machine and the only solution was to pay to have a better setup. Since the operation was very delicate and the risk of exceeding the service time with the consequent extra expenses, I decided to delete the step. Another backlash was the weight of the Python Packages. During the deployment try the only way to upload the app was using a Docker file that creates an image of the environment image. But the high count of algorithms dependencies took the docker file weight around the 8 GBs, much more than what the basic cloud account allows. Storage system evolution took place, because at the time I was searching to move the storage from local level, using the csv file, to a database level, but as I saw these difficulties I decided to stay with the csv files.

---

<sup>8</sup><https://dash.plotly.com>

<sup>9</sup><https://flask.palletsprojects.com/en/2.0.x/>

<sup>10</sup><https://www.djangoproject.com/>

## Summary of Chapter 4

In brief:

- **Application:**

- OpenML Benchmarks: execution time and re-run option for each algorithm, choice between entering the dataset ID set or the number of datasets to be tested with the minimum number of instances.
- Kaggle Benchmarks: execution time and re-run option for each algorithm, choice between 8 Kaggle dataset competition.
- Test Benchmarks: execution time and re-run option for each algorithm, test in a single dataset ID with only a tool or the all set.
- Past Result OpenML: analysis of past benchmarks run in OpenML dataset.
- Past Result OpenML: analysis of past benchmarks run in Kaggle dataset.

- **Goals timeline:**

1. Terminal application for benchmarking 5 AutoML framework on OpenML and Kaggle dataset.
2. Movement from Terminal to Local Web Application for a better usability.
3. Upload to the cloud to publish the application on the web.

# Chapter 5

## Benchmarks and Results

In this chapter we discuss the structure of the benchmarks with a focus on the results using boxplot and bar chart. With OpenML benchmarks, we run the 5 tools on 18 datasets (Table 5.1), 9 for each type of problem. Where the 18 datasets are divided by dimensional size, with datasets having less than 1000 rows, between 1000 and 100000, and more than 100000. Furthermore each dimensional category is run 3 times based on the execution time of the tools, 15 minutes, 30 minutes and 60 minutes. At the end each algorithm is run 54 times. The Kaggle Benchmarks were run with 8 competition datasets (Table 5.2). They are different from the OpenML benchmarks in problem type and the execution time of each framework, which are the same as those mentioned above. In this case each algorithm is run 24 times.

All benchmarks were run in a single machine, an HP EliteDesk Tower supplied with 8 cores and 16GB of RAM and 10GBs of Swap memory with Ubuntu Linux 20.04.02 LTS operating system.

## 5.1 Datasets

Dataset Name	Dataset ID	Problem Type	Number of Instances	Number of Features
anneal	2	Classification	898	39
arrhythmia	5	Classification	452	280
diabetes	37	Classification	768	9
cleveland	194	Regression	303	14
laser	42364	Regression	993	5
stock	223	Regression	950	10
JapaneseVowels	375	Classification	9961	15
yprop_4.1	416	Regression	8885	252
bank8FM	572	Regression	8192	9
sulfur	23515	Regression	10081	7
kdd_internet_usage	981	Classification	10108	69
artificial-characters	1459	Classification	10218	8
coverttype	180	Classification	110393	55
BNG(breastTumor)	1201	Regression	116640	10
BNG(vote)	143	Classification	131072	17
BNG(glass)	265	Classification	137781	10
black_friday	41540	Regression	166821	10
medical_charges_nominal	42559	Regression	163065	12

Table 5.1: OpenML Datasets

Dataset Name	Problem Type	Number of Instances	Number of Features
Titanic	Classification	892	25
CommonLit Readability Prize	Classification	7092	12
Forest Cover Type Prediction	Classification	15120	118
Ghouls Goblins and Ghosts Boo	Classification	372	7
Tabular Playground Series Jan 2021	Regression	300000	33
CommonLit Readability Prize	Regression	7092	12
Mercedes Benz Greener Manufacturing	Regression	4209	388
Global Energy Forecasting Competition 2012	Regression	18757	61

Table 5.2: Kaggle Datasets

## 5.2 OpenML Benchmarks

### 5.2.1 Smaller Datasets

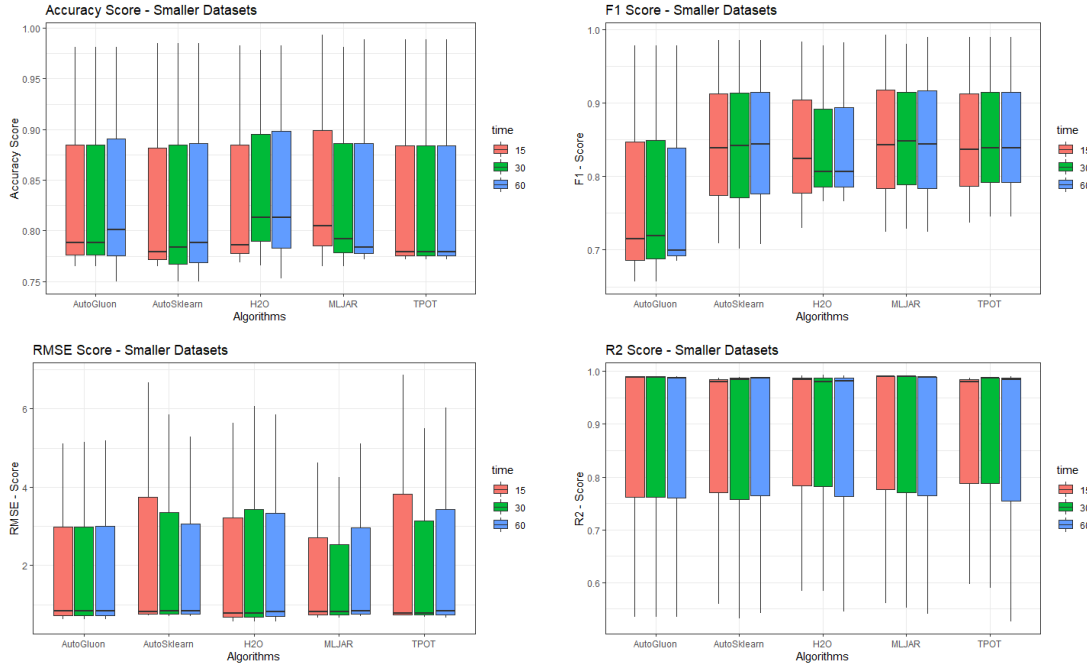


Figure 5.1: Results for smaller datasets

Starting with the first row of figures, we can see the classification results for a smaller data set. Generally each framework increases the accuracy prediction with a greater time given, while others have decreased the goodness results, such as MLJAR which instead has a stable F1-Score for all executions. H2O beats the other tools for the 30 and 60 minute time categories, while MLJAR wins for the 15 minute one. Then the second row of graphs represents the regression results. Here we can see more consistent predictions, regarding the R2 score on each tool that reaches the maximum score using the smallest execution time, with a smaller increase in the prediction results for AutoSklearn and TPOT. We can find the same result for the RMSE score function. So for the H2O classification problem it seems to be better than the others, while for the regression problem there is not a big difference between all the tools.

### 5.2.2 Medium-sized Datasets

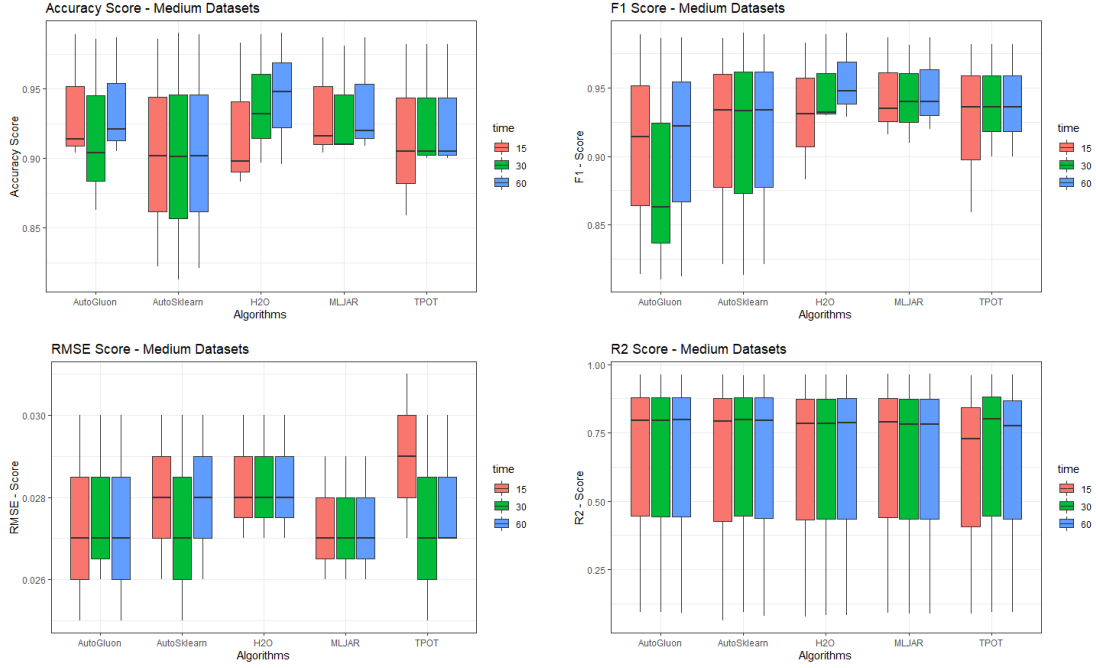


Figure 5.2: Results for medium datasets

We then consider the datasets with medium dimensionality. The first row, as before, describes the classification results and the second the regression results. In general for classification problems, all frameworks increase their prediction result with more given time. We note that the accuracy of AutoSklearn remains very stable and with a greater variance than the other tools even with the longer period of time given. On the other hand like before, H2O Accuracy shows great performance with a constant increase prediction reaching the top within the 60 minutes of given time. Similar results are shown in the F1-Score table. If we take into consideration the regression problems, here the situation is more stable, AutoGluon, H2O and MLJAR return about the same prediction results in all runs, while TPOT shows a decrease in RMSE with a reduction of the variability in the last run.

### 5.2.3 Large Datasets

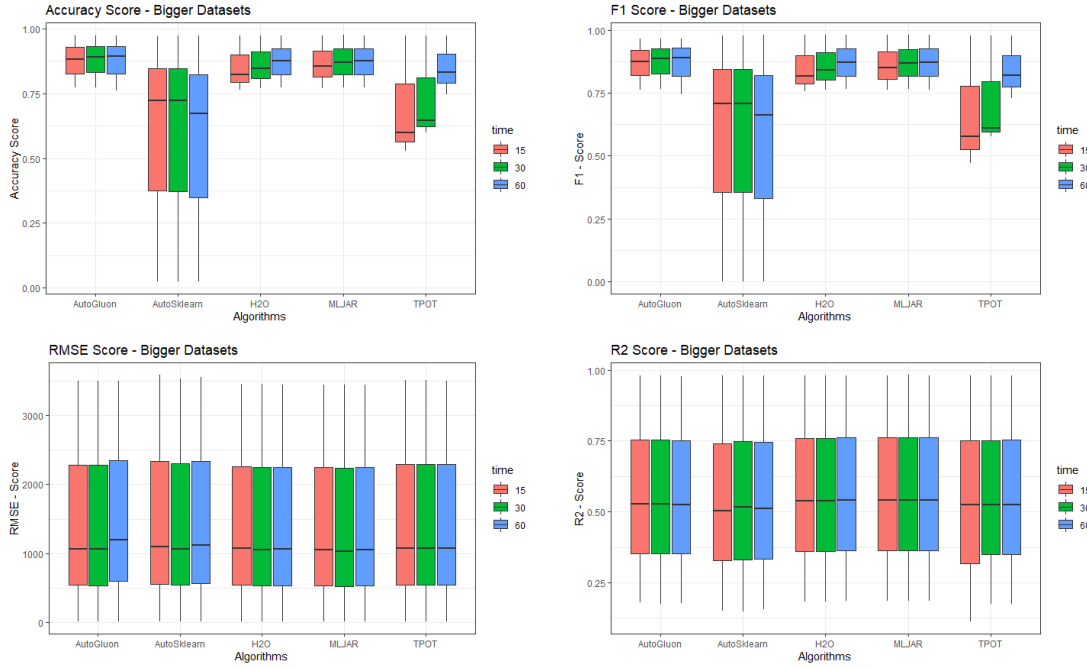


Figure 5.3: Results for bigger datasets

In this chart table we discuss the results for larger data sets, the first row represents the classification predictions and the second the regression predictions. As we can see AutoSklearn stands out for its high variability in all three executions, while the other tools have less variability but have greater accuracy. Unlike the previous results, AutoGluon is the best tool for the larger dataset, also note that its average accuracy in the three datasets tested during the 15 minutes of running is more accurate than the others. In the next line, for the regression problem H2O and MLJAR barely beat the other frameworks. But more generally, each instrument shows a high regression capacity with stable variance.

## 5.3 Kaggle Benchmarks

### 5.3.1 Classification Problems

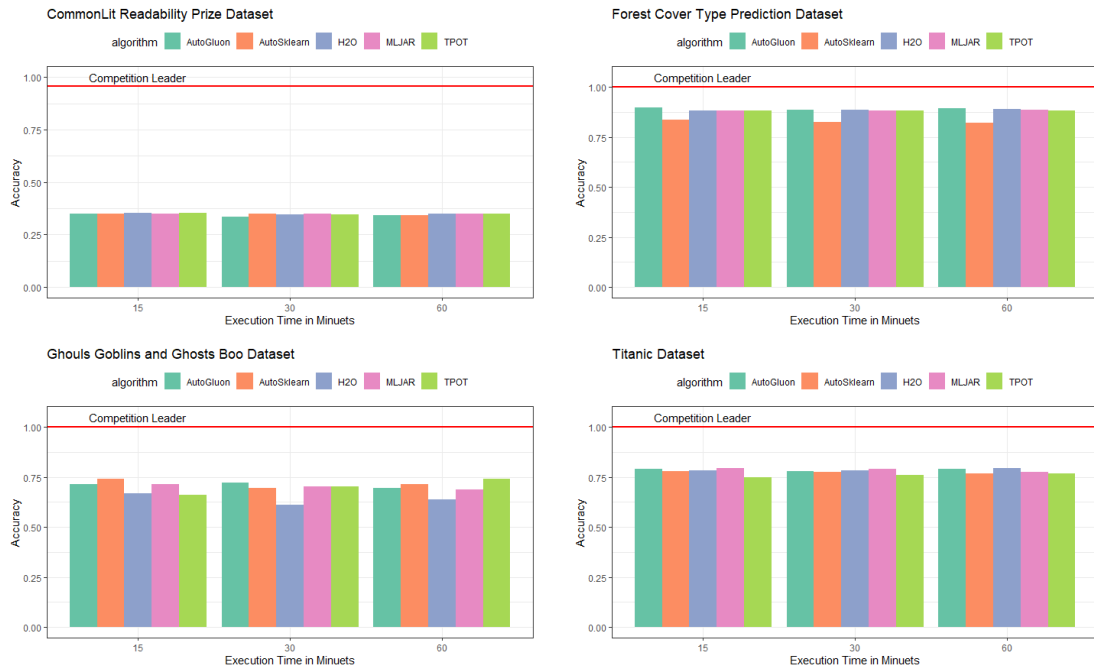


Figure 5.4: Kaggle Classification datasets results

In the classification competition all the five show poor prediction results with big gaps. Moreover in general there is not a big difference in results between the three time execution category.

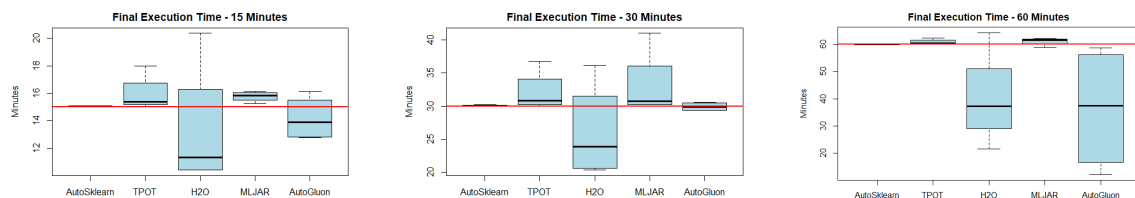


Figure 5.5: Boxplots of algorithms execution times for Kaggle Classification datasets

Averagely speaking H2O.ai uses less time and it gives back a reasonable result compared to the others, while TPOT and MLJAR exceed the amount of time. Note also AutoGluon in the 60 minutes runs uses less given time, and so it means, it gives the same results as the other frameworks, or even higher.



### 5.3.2 Regression Problems

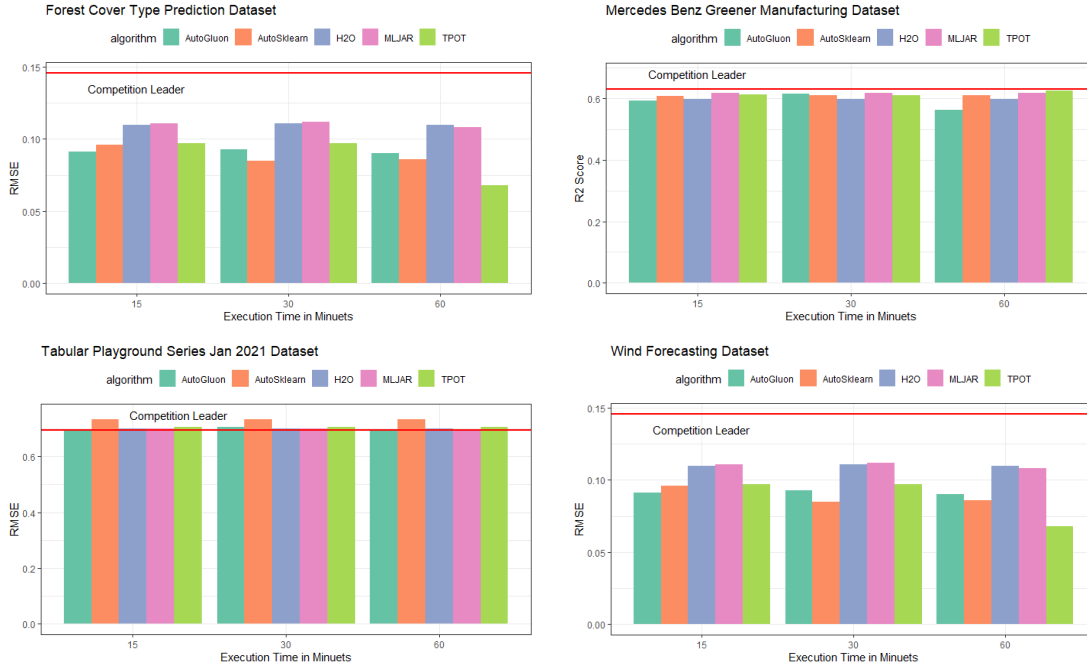


Figure 5.6: Kaggle Regression datasets results

Things change for regression problems, here all frameworks outperform the competition leader for the Forecast Type Prediction dataset and Wind Forecasting dataset. For the Tabular Playground AutoGluon, H2O and MLJAR slightly overtook the competition leader, instead for the Mercedes Benz Greener Manufacturing dataset only AutoGluon with the higher amount of time managed to get very close to the leader's result, but not surpass it. In general TPOT shows great performance with a continuous decrease of RMSE and increase of R2-Score, while using the higher execution time outperforms all other 4 tools.

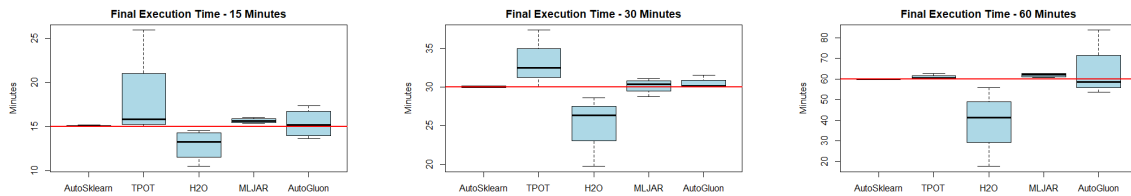


Figure 5.7: Boxplots of algorithms execution times for Kaggle Regression datasets

AutoSklearn, MLJAR and AutoGluon use the exact time given in all three runs, while TPOT during the 15 and 30 minute run exceeds the given time but without a visible increase in performance over the other tools. On the other hand, H2O uses less time given in all runs showing decent performance, achieving the same results as the other frameworks in the Mercedes and Tabular datasets.

## Summary of Chapter 5

To sum up:

- **OpenML Benchmarks:**

- For the smaller datasets every tool show increase in performance with higher time, but overall H2O outperform the others for classification task, while in regression problem no distinct difference appeared.
- For the medium datasets, in the classification task AutoSklearn shows high variance in all three runs, in which H2O constantly increases its performance. On the other hand, for the regression task, all tools show no increase in performance except for TPOT which decreases its RMSE and variance during the three runs.
- For the bigger datasets AutoGluon outperform the other tools for classification task, while MLJAR and H2O barely beat the others for regression task.

- **Kaggle Benchmarks:**

- In classification problems all five algorithms show poor performance, with H2O using less time but reaching the same level of the others.
- In Regression problems all tools show better performance with TPOT outperforming the others, and H2O using less time but reaching a decent results.

# Chapter 6

## Conclusion

In conclusion, I think there is still a lot of work to be done before considering AutoML algorithms comparable to human data scientists. The five tools we reviewed have great potential in the CASH problem, but data cleaning and feature engineering are still an Achilles' heel for them. In these two fields, experts have the advantage of knowing which features to create and remove to build a powerful pipeline. One example is the Titanic dataset which is the first competition we enter when we sign up for Kaggle, it has different kinds of features like text and numbers with lots of missing values. But none of the frameworks analysed managed to reach the maximum score.

Summarizing, we can say that for datasets with a small to medium dimensionality range the best tools are H2O for classification activity and TPOT for regression activity. On the other hand, for larger data sets the best are AutoGluon for classification problems and MLJAR for regression problems.



# Special thanks

I would like to thank:

- **Professor Claudio Lucchese** supervisor of this thesis, for the opportunity given.
- **My family and all my friends**, for supporting me during these three years.
- **The team of each AutoML tools** for answering my issues published on the relative GitHub Repository.
- **Giulio Moretto**, for giving me a powerful computer to run each analysed benchmarks.
- **You, reader**, for reading this thesis, I hope it was helpful to have more knowledge about AutoML algorithms.



# Bibliography

- [1] AutoML.org. *SMAC*. URL: <https://www.automl.org/automated-algorithm-design/algorithm-configuration/smac/>.
- [2] Think Autonomus. *How Google's Self-Driving Cars Work*. URL: <https://www.thinkautonomous.ai/blog/?p=how-googles-self-driving-cars-work>.
- [3] Dan Bochman. *Featuretools Machine Learning Data Science Open-source Spotlight 2*. URL: <https://www.youtube.com/watch?v=Q5U9rEKHIsk>.
- [4] Jason Brownlee. *A Gentle Introduction to Ensemble Learning Algorithms*. URL: <https://machinelearningmastery.com/tour-of-ensemble-learning-algorithms/>.
- [5] Jason Brownlee. *Combined Algorithm Selection and Hyperparameter Optimization (CASH Optimization)*. URL: <https://machinelearningmastery.com/combined-algorithm-selection-and-hyperparameter-optimization/>.
- [6] Jason Brownlee. *Stacking Ensemble Machine Learning With Python*. URL: <https://machinelearningmastery.com/stacking-ensemble-machine-learning-with-python/>.
- [7] DeepMind. *Making History - AlphaGo*. URL: <https://deepmind.com/research/case-studies/alphago-the-story-so-far>.
- [8] Benoit Descamps. *SuccessiveHalving*. URL: <https://towardsdatascience.com/tuning-hyperparameters-part-i-successivehalving-c6c602865619>.
- [9] Thomas Dinsmore. *Automated Machine Learning: A Short History*. URL: <https://www.datarobot.com/blog/automated-machine-learning-short-history/>.
- [10] Niklas Donges. *Gradient Descent: An Introduction to 1 of Machine Learning's Most Popular Algorithms*. URL: [https://www.tensorflow.org/agents/tutorials/intro\\_bandit](https://www.tensorflow.org/agents/tutorials/intro_bandit).
- [11] Radwa Elshawy, Mohamed Maher, and Sherif Sakr. *Automated Machine Learning: State-of-The-Art and Open Challenges*. 2019. arXiv: 1906.02287 [cs.LG].
- [12] Nick Erickson et al. *AutoGluon-Tabular: Robust and Accurate AutoML for Structured Data*. 2020. arXiv: 2003.06505 [stat.ML].
- [13] Matthias Feurer et al. "Efficient and Robust Automated Machine Learning". In: *Advances in Neural Information Processing Systems*. Ed. by C. Cortes et al. Vol. 28. Curran Associates, Inc., 2015. URL: <https://proceedings.neurips.cc/paper/2015/file/11d0e6287202fced83f79975ec59a3a6-Paper.pdf>.

- [14] Elad Hoffer, Itay Hubara, and Daniel Soudry. *Train longer, generalize better: closing the generalization gap in large batch training of neural networks*. 2018. arXiv: 1705.08741 [stat.ML].
- [15] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. *Automated Machine Learning: Methods, Systems, Challenges*. 1st. Springer Publishing Company, Incorporated, 2019, pp. 82, 115. ISBN: 3030053172.
- [16] Ihab Ilyas. *A Scalable Prediction Engine for Automating Structured Data Prep, Ihab Ilyas*. URL: <https://youtu.be/z2RQz9zdpy0>.
- [17] Prateek Joshi. *A Hands-On Guide to Automated Feature Engineering using Featuretools in Python*. URL: <https://www.analyticsvidhya.com/blog/2018/08/guide-automated-feature-engineering-featuretools-python/>.
- [18] Saurav Kaushik. *Introduction to Feature Selection methods with an example (or how to select the right variables?)* URL: <https://www.analyticsvidhya.com/blog/2016/12/introduction-to-feature-selection-methods-with-an-example-or-how-to-select-the-right-variables/>.
- [19] Udayan Khurana et al. “Cognito: Automated Feature Engineering for Supervised Learning”. In: *2016 IEEE 16th International Conference on Data Mining Workshops (ICDMW)*. 2016, pp. 1304–1307. DOI: 10.1109/ICDMW.2016.0190.
- [20] Will Koehres. *How to automatically create machine learning features*. URL: <https://towardsdatascience.com/automated-feature-engineering-in-python-99baf11cc219>.
- [21] Will Koehrsen. *A Conceptual Explanation of Bayesian Hyperparameter Optimization for Machine Learning*. URL: <https://towardsdatascience.com/a-conceptual-explanation-of-bayesian-model-based-hyperparameter-optimization-for-machine-learning-b8172278050f>.
- [22] J. R. Koza. “Survey of genetic algorithms and genetic programming”. In: *Proceedings of WESCON’95*. 1995, pp. 589–. DOI: 10.1109/WESCON.1995.485447.
- [23] Sanjay Krishnan and Eugene Wu. *AlphaClean: Automatic Generation of Data Cleaning Pipelines*. 2019. arXiv: 1904.11827 [cs.DB].
- [24] Erin LeDell and Sebastien Poirier. “H2O AutoML: Scalable Automatic Machine Learning”. In: *7th ICML Workshop on Automated Machine Learning (AutoML)* (June 2020). URL: [https://www.automl.org/wp-content/uploads/2020/07/AutoML\\_2020\\_paper\\_61.pdf](https://www.automl.org/wp-content/uploads/2020/07/AutoML_2020_paper_61.pdf).
- [25] Marius Lindauer et al. *SMAC v3: Algorithm Configuration in Python*. <https://github.com/automl/SMAC3>. 2017.
- [26] Subir Mansukhani. *HyperOpt: Bayesian Hyperparameter Optimization*. URL: <https://blog.dominodatalab.com/hyperopt-bayesian-hyperparameter-optimization>.
- [27] Meredith Mante. *Cognito - Transformation Graph Exploration*. URL: <https://www.coursera.org/lecture/ibm-rapid-prototyping-watson-studio-autoai/cognito-transformation-graph-exploration-ub00U>.

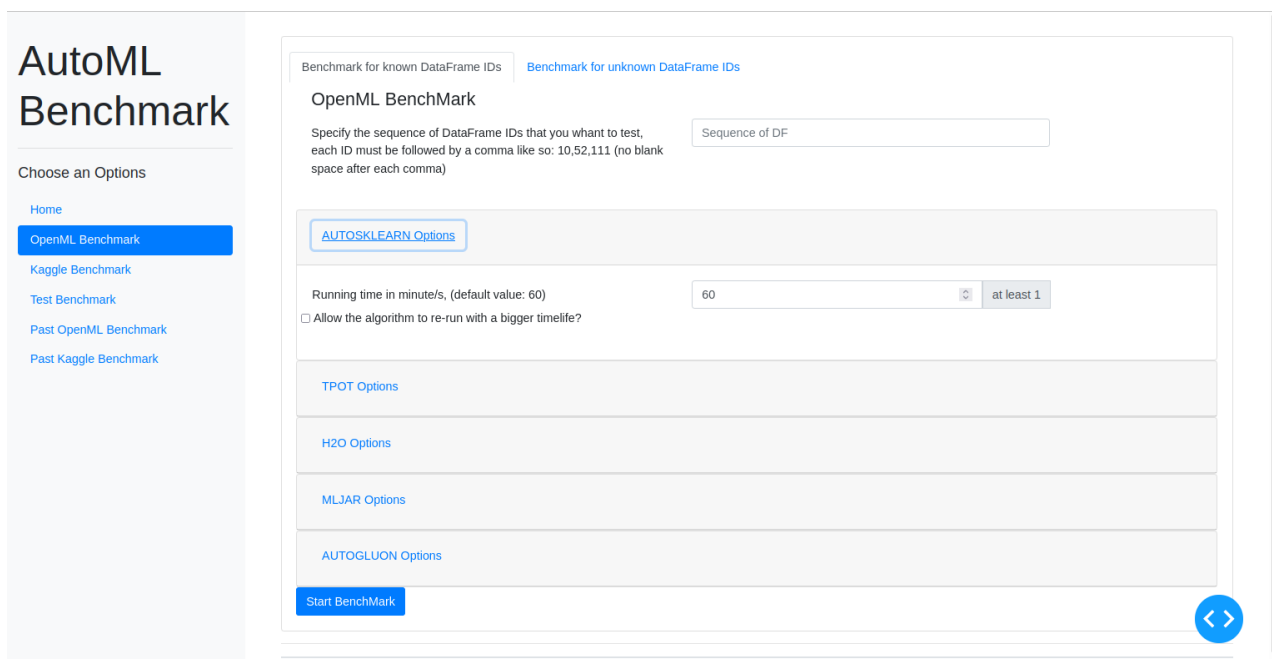


- [28] Meredith Mante. *Cognito - Transforms and the Transformation Graph*. URL: <https://www.coursera.org/lecture/ibm-rapid-prototyping-watson-studio-autoai/cognito-transforms-and-the-transformation-graph-RBH4d>.
- [29] Kjell Johnson Max Kuhn. “Feature Engineering and Selection: A Practical Approach for Predictive Models”. In: *Annalen der Physik* 1.1 (2019), pp. 96–97. DOI: <https://doi.org/10.1201/9781315108230>.
- [30] Randal S. Olson and Jason H. Moore. “TPOT: A Tree-based Pipeline Optimization Tool for Automating Machine Learning”. In: *Proceedings of the Workshop on Automatic Machine Learning*. Ed. by Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. Vol. 64. Proceedings of Machine Learning Research. New York, New York, USA: PMLR, June 2016, pp. 66–74.
- [31] Randal S. Olson et al. *Evaluation of a Tree-based Pipeline Optimization Tool for Automating Data Science*. 2016. arXiv: 1603.06212 [cs.NE].
- [32] Izgi Arda Ozsubasi. *Few-Shot Learning (FSL): What it is and its Applications*. URL: <https://research.aimultiple.com/few-shot-learning/>.
- [33] Aleksandra Płońska and Piotr Płoński. *MLJAR: State-of-the-art Automated Machine Learning Framework for Tabular Data. Version 0.10.3*. Łapy, Poland, 2021. URL: <https://github.com/mljar/mljar-supervised>.
- [34] Aleksandra Płońska and Piotr Płoński. *Steps of AutoML*. URL: <https://supervised.mljar.com/features/automl/>.
- [35] Gil Press. *Cleaning Big Data: Most Time-Consuming, Least Enjoyable Data Science Task, Survey Says*. URL: <https://www.forbes.com/sites/gilpress/2016/03/23/data-preparation-most-time-consuming-least-enjoyable-data-science-task-survey-says/?sh=7ec1fb1f6f63>.
- [36] Win Pure. *Why should I automate my data cleaning tasks?* URL: <https://winpure.com/blog/why-should-i-automate-my-data-cleansing-tasks/>.
- [37] TensorFlow. *Introduction to Multi-Armed Bandits*. URL: [https://www.tensorflow.org/agents/tutorials/intro\\_bandit](https://www.tensorflow.org/agents/tutorials/intro_bandit).
- [38] Chris Ré Theo Rekatsinas Ihab Ilyas. *HoloClean: Weakly Supervised Data Repairing*. URL: <https://holoclean.github.io/gh-pages/blog/holoclean.html>.
- [39] Chris Thornton et al. *Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms*. 2013. arXiv: 1208.3719 [cs.LG].
- [40] Anh Truong et al. “Towards Automated Machine Learning: Evaluation and Comparison of AutoML Approaches and Tools”. In: *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)* (Nov. 2019). DOI: 10.1109/ictai.2019.00209. URL: <http://dx.doi.org/10.1109/ICTAI.2019.00209>.
- [41] Joaquin Vanschoren. *Meta-Learning: A Survey*. 2018. arXiv: 1810.03548 [cs.LG].
- [42] Marc-André Zöller and Marco F. Huber. *Benchmark and Survey of Automated Machine Learning Frameworks*. 2021. arXiv: 1904.12054 [cs.LG].



# Appendix A

## Application Screenshots



The screenshot displays the OpenML Benchmark web interface. On the left, a sidebar titled 'AutoML Benchmark' contains a 'Choose an Options' section with links for Home, OpenML Benchmark (highlighted), Kaggle Benchmark, Test Benchmark, Past OpenML Benchmark, and Past Kaggle Benchmark. The main content area is divided into two tabs: 'Benchmark for known DataFrame IDs' (selected) and 'Benchmark for unknown DataFrame IDs'. Under the selected tab, the 'OpenML BenchMark' section prompts the user to 'Specify the sequence of DataFrame IDs that you want to test, each ID must be followed by a comma like so: 10,52,111 (no blank space after each comma)' and includes a text input field labeled 'Sequence of DF'. Below this, the 'AUTOSKLEARN Options' section features a 'Running time in minute/s, (default value: 60)' input field set to 60, a checkbox for 'Allow the algorithm to re-run with a bigger timelife?', and a 'Start BenchMark' button. Further down are sections for 'TPOT Options', 'H2O Options', 'MLJAR Options', and 'AUTOGLUON Options', each with a corresponding button. A blue circular navigation button with left and right arrows is located in the bottom right corner.

Figure A.1: Screenshot of the OpenML Benchmark Interface

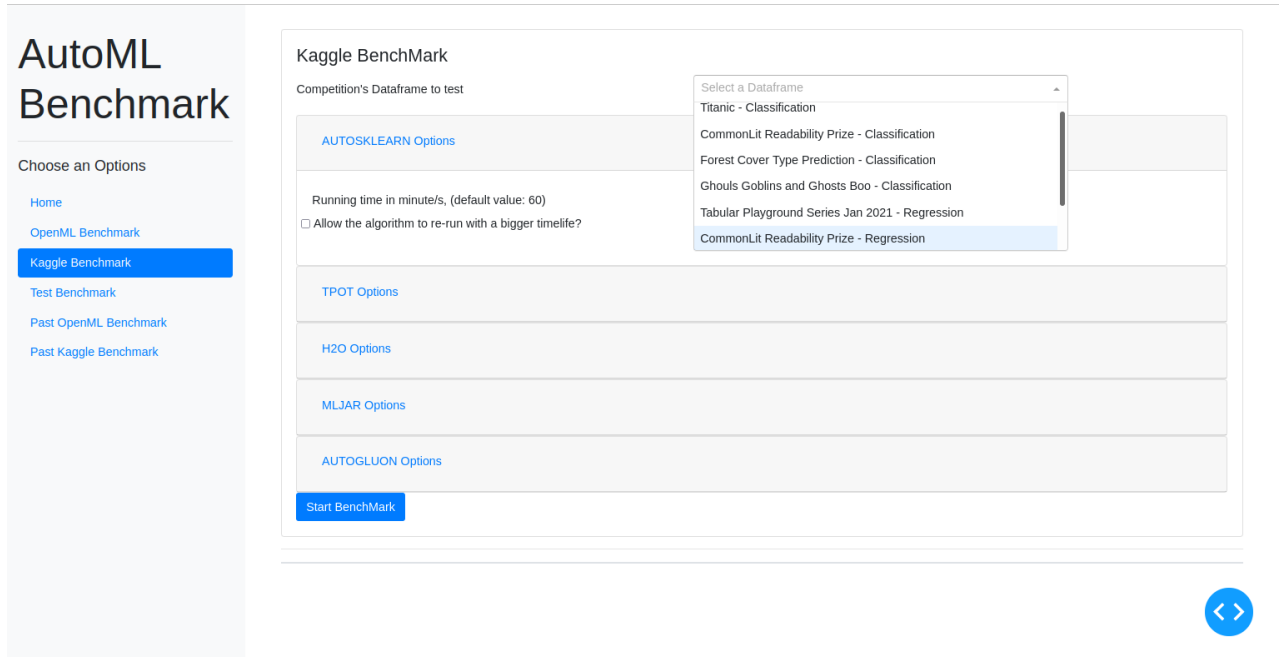


Figure A.2: Screenshot of the Kaggle Benchmark Interface

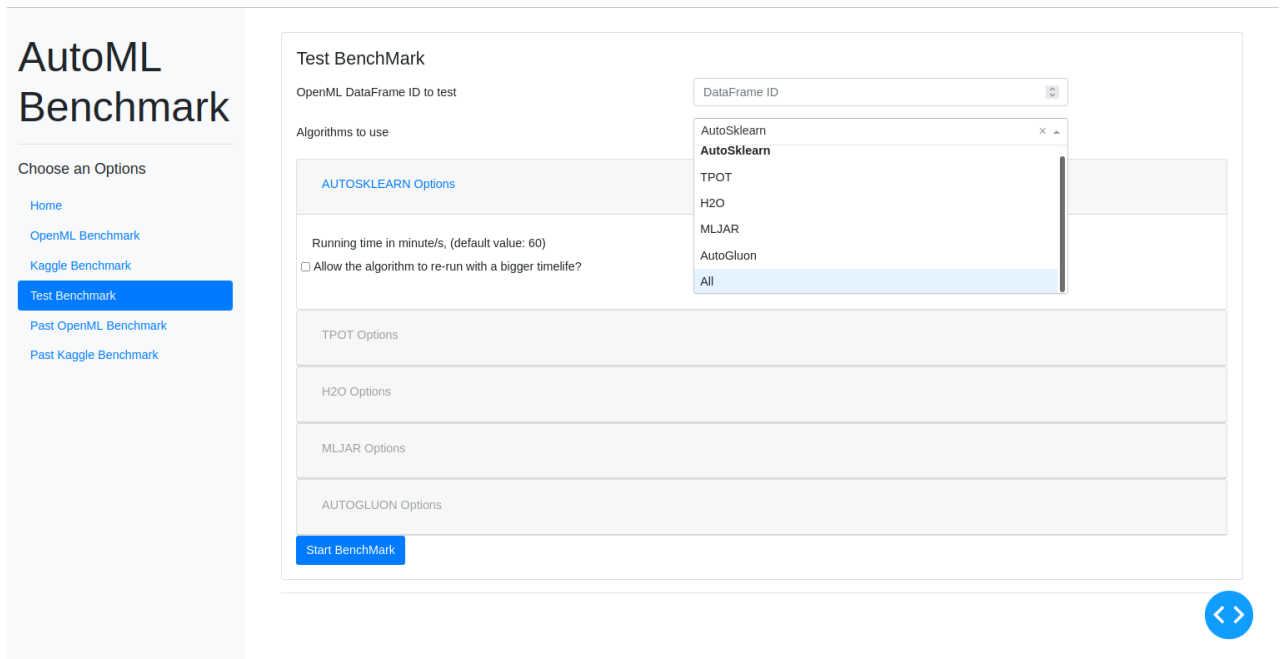


Figure A.3: Screenshot of the Test Benchmark Interface



Figure A.4: Screenshot of the Analysis of Past Benchmark

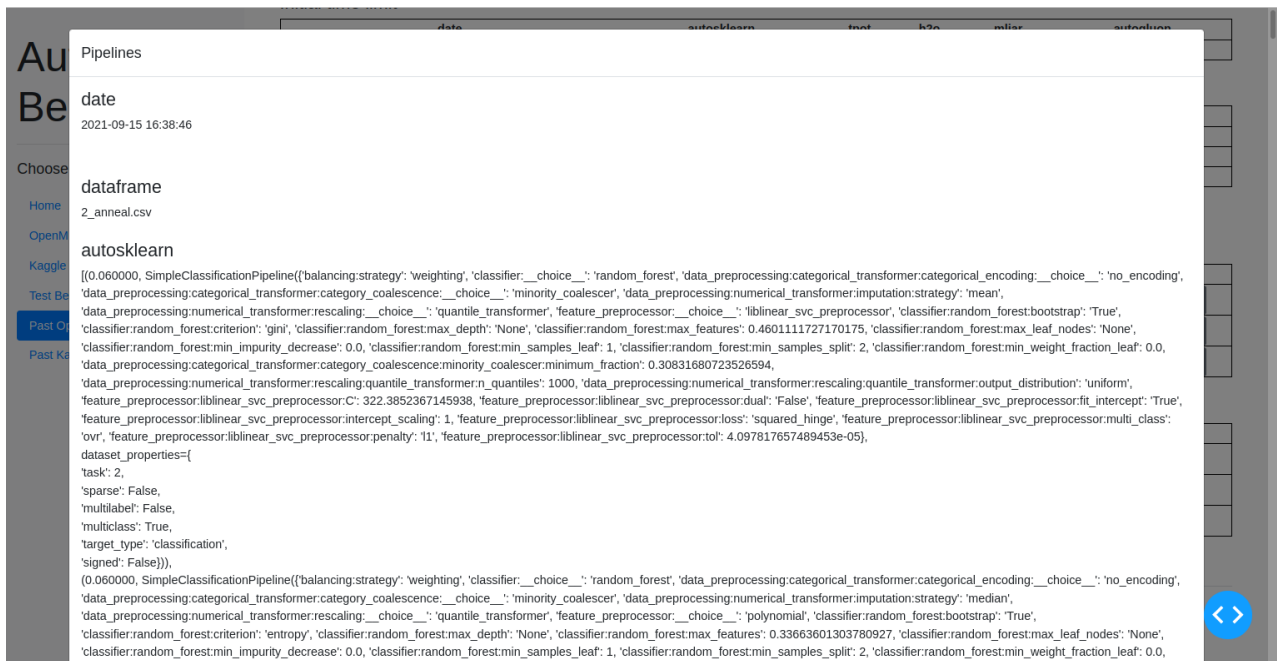


Figure A.5: Screenshot of the pipeline view