TU WIEN Informatics

Automatisiertes maschinelles Lernen mit metaheuristischen Algorithmen

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Data Science

eingereicht von

Gent Rexha, B.Sc.
Matrikelnummer 11832486

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Priv.-Doz. Dr. Nysret Musliu

Wien, 12. August 2021

Gent Rexha	Nysret Musliu

TU WIEN Informatics

Automated Machine Learning using Metaheuristic Algorithms

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Data Science

by

Gent Rexha, B.Sc.
Registration Number 11832486

to the Faculty of Informatics

at the TU Wien

Advisor: Priv.-Doz. Dr. Nysret Musliu

Vienna, 12th August, 2021

Gent Rexha	Nysret Musliu

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Gent Rexha, B.Sc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 12. August 2021

Gent Rexha

Acknowledgements

I would like to convey my sincere appreciation to my advisor, Priv.-Doz. Dr. Nysret Musliu. This thesis would not have been feasible without his constant assistance, encouragement, feedback, and extensive understanding in the subject.

My deepest appreciation goes to my parents and family, for their continuous support, encouragement, and who have always been there for me and helped me through many difficult times. It would not have been possible to complete this work without them.

Last but not least, I would want to thank Majlinda not only for her love, but also for her patience, encouragement, thoughtful comments, and emotional support while I was working and writing on this thesis.

This work was supported by the Austrian Science Fund (project: KIRAS-PreMI D192020-4003)

Kurzfassung

Maschinelles Lernen wird zu einem integralen Bestandteil jeder modernen Softwareanwendung. Sein Erfolg ist direkt mit der Auswahl des richtigen Algorithmus für die verschiedenen wichtigen Lernaufgaben verbunden. Der Prozess der Algorithmenauswahl birgt jedoch seine eigenen Herausforderungen, denn das "no free lunchTheorem besagt, dass "jede höhere Leistung bei einer Klasse von Problemen durch die niedrigere Leistung bei einer anderen Klasse ausgeglichen wird". Mit anderen Worten, ein Algorithmus kann nicht für alle Arten von Problemen die beste Lösung sein. Ein Algorithmus kann für ein Problem eine optimale und für ein anderes eine schlechte Lösung sein.

Diese Arbeit konzentriert sich auf die Kategorie des überwachten maschinellen Lernens, bei dem die Daten in Eingabe- und Ausgabevariablen aufgeteilt und dem Algorithmus übergeben werden. Das Ziel darin besteht, Muster zu erkennen, bei denen nur die Eingabevariablen die Ausgabevariable vorhersagen können. Die Notwendigkeit, optimale Lösungen für diese Art von Problemen zu finden, hat zum Aufkommen von Automated Machine Learning (AutoML) geführt. Die AutoML-Domäne befasst sich mit der Ermittlung des leistungsfähigsten Algorithmus für ein bestimmtes maschinelles Lernproblem sowie mit der Bestimmung anderer kritischer Schritte wie Vorverarbeitung, Featureextraktion und Featureauswahl.

Das vorgeschlagene Framework mit dem Namen *MetaheuristicSklearn* ermöglicht die Entwicklung und Reproduktion von mehrstufigen, kontrollierten Klassifizierungspipelines auf zusammenhängender Weise. Das Framework bietet eine Standardmethode zur Implementierung und Integration von Pipelineschritten und Parametern unter Verwendung verschiedener Techniken. Darüber hinaus wurden die metaheuristischen Algorithmen: (i) Simulated Annealing, (ii) Tabu Search und (iii) Iterated Local Search (ILS) im Rahmen von Solver-Algorithmen zur Ermittlung optimaler AutoML-Lösungen angewendet und evaluiert. Die drei Lösungsverfahren wurden in einer großen Datensatz-Benchmark-Sammlung eingesetzt, um das Framework zu evaluieren. Die Leistung der Algorithmen wurde bewertet und mit den modernsten AutoML-Frameworks verglichen. Darüber hinaus haben wir mehrere neighborhood operators vorgeschlagen, verschiedene Algorithmenkonfigurationen bewertet und die einzelnen Komponenten untersucht.

Basierend auf unseren Experimenten mit 31 Datensätzen aus der OpenML-CC18 Benchmarking Suite, schneiden Tabu Search und ILS besser ab als Simulated Annealing. Tabu Search war der beste Algorithmus für 15 von 31 Datensätzen, ILS für 13 von 31, und

Simulated Annealing war nur in 3 von 31 Datensätzen der beste Algorithmus. Der Algorithmus-Parameter-Tuning-Prozess erwies sich ebenfalls als recht effektiv, wobei die Gesamtverbesserung des F1-Score im Vergleich zu den Standardparametern durchschnittlich 7% betrug.

Schließlich bietet das vorgeschlagene MetaheuristicSklearn-Framework im Vergleich zu hochmodernen AutoML-Frameworks in 9 von 31 Fällen eine leistungsfähigere Pipeline. Darüber hinaus war die Genauigkeit des MetaheuristicSklearn-Frameworks für alle Datensätze etwa 2% schlechter als die des leistungsstärksten Frameworks.

Abstract

Machine learning is becoming an integral part of every modern software application. Its success is directly linked to the appropriate algorithm selection when dealing with various essential learning tasks. But the algorithm selection process has its own challenges, as the “no free lunch” theorem states that "any elevated performance over one class of problems is offset by performance over another class". In other words, an algorithm can not be the best performing solution for all types of problems. An algorithm may be an optimistic answer for one problem and a poor answer for another.

This thesis focuses on the supervised machine learning category, where the data is separated into the input and output variables, both given to the algorithm, where the objective is to identify patterns in which only the input data can predict the output variable. The need of finding the optimal solutions to those types of problems has led to the rise of Automated Machine Learning (AutoML). The AutoML domain is concerned with identifying the best performing algorithm for a particular machine learning issue, as well as determining other critical steps such as preprocessing, feature extraction, and feature selection.

The proposed framework, named *MetaheuristicSklearn*, allows multi-step controlled classification pipelines to be developed and reproduced in a cohesive manner. The framework provides a standard way of implementing and integrating pipeline steps and parameters using various techniques. Furthermore, the metaheuristic algorithms: (i) Simulated Annealing, (ii) Tabu Search, and (iii) Iterated Local Search (ILS) have been applied and evaluated in the context of solver algorithms for finding optimum AutoML solutions. The three-solver techniques were utilized in a large dataset benchmark collection, the OpenML-CC18 Benchmarking Suite, to assess the framework. The performance of the algorithms has been assessed and compared with the state-of-the-art AutoML frameworks. Additionally, we proposed several neighborhood operators, evaluated several algorithm configurations, and examined the individual components.

Based on the experiments using 31 datasets from the benchmark collection, Tabu Search and ILS perform better than Simulated Annealing. Tabu Search was the best performing algorithm for 15 out of 31 datasets, ILS for 13 out of 31, and Simulated Annealing was only the best performing algorithm in 3 out of 31 datasets. The algorithm parameter tuning process was also proven to be quite effective, where the overall improvement in F1-Score was 7% on average when compared to the default parameters.

Finally, the proposed MetaheuristicSklearn framework compared to state-of-the-art AutoML frameworks gives a better performing pipeline of 9 out of 31 cases. In addition, the precision of the MetaheuristicSklearn framework was around 2% worse than the best performing framework for all datasets.

Contents

CHAPTER 1

Introduction

Machine Learning (ML) is one of the most exciting and fast-growing fields in computer science. It's a sub-field of artificial intelligence that focuses on algorithms that change as they process data, enabling computers to improve without being explicitly programmed. ML has been successfully applied to many different tasks, including image recognition as well as natural language processing. In a nutshell, ML is the future of smart computing: it enables applications to respond automatically to new information or situations. Generally, machine learning approaches are categorized into three specific categories:

1. Supervised Machine Learning - The data is divided into the input variables and the output variable, both given to the algorithm, where the goal is to find patterns such that the output variable can only be predicted by the input data.
2. Unsupervised Machine Learning - Here the output variable is absent, the goal here is to identify patterns or clusters in the data only on the basis of patterns contained in the input data without depending on the output variable.
3. Reinforcement Learning - Has three main components: the agent, the environment and the reward. The objective of the agent is to learn a policy from his interactions with the environment and then maximize the reward provided based on those interactions.

In this thesis we are going to concentrate on the most popular amongst the three, Supervised Machine Learning. Depending on the form of outcome variable, supervised learning can be separated into two subcategories: Classification and Regression. Before we continue, we need to define some notations for our problem [BJP20, Spa03]: the input data, also called training data, is denoted as $\{x_1, \ldots, x_n\} \subseteq \mathbb{R}^p$ and their labels $(y_1, \ldots, y_n) \in \{1, \ldots, k\}^n$, and finally g, which is the function to find a correlation between input and output variable $g : X \to Y$.

The output variable in the Supervised Classification is a category $(y_1, \ldots, y_n) \in \{1, \ldots, k\}^n$, where if $k = 2$ it is a binary classification problem or when $k > 2$ it is presented as a multi classification problem. The goal here is to identify patterns in the data that connect the input data to the output category. One of the common examples is whether an email is spam or not. For this task an algorithm can be modeled from previously tagged spam emails and other non-spam emails, where said model attempts to find some correlation between the text and the data within the email and their respective tags, spam or no spam. From the created model, which helps us to identify new unseen emails. This modeling of the correlation between input data and output category is made possible by classification algorithms. Some of the most common ones are: Deep Learning Networks, Support Vector Machines, Decision Trees, Random Forests, and K-Nearest Neighbor.

At the other hand, when it comes to supervised regression, the output variable is continuous $Y = \{y \in \mathbf{R}\}$. The most common method is the linear regression model [Fil19, Vap95], which has the form: $y = f(x_1, x_2, \ldots, x_p) + \varepsilon$, with the linear function $f(x_1, x_2, \ldots, x_p) = \beta_0 + \sum_{j=1}^{p} x_j \beta_j$. The goal is to find a functional relation f which justifies $y \approx f(x_1, x_2, \ldots, x_p)$. The β_j are unknown parameters or coefficients, which will be estimated from given data. The variables x_j can come from different sources, for example, one would try to model housing prices on the basis of inputs such as size of the house, distance to the nearest schools, number of rooms, etc. Other popular models include: Lasso Regression, Ridge Regression, Logistic Regression, and Regression Trees. The benefit of these models is that they are usually simple and can also be interpreted by the human.

Understandably, the number of algorithms is not limited to those listed above. As if that amount of variety was not enough, the problem of machine learning has an extra layer of complexity in the configuration of each algorithm in its own. These algorithm parameters, also known as hyper-parameters, are what determine the rules within the algorithms like: tree size, number of neighbors, kernel type, etc. The problem of finding the right set of hyper-parameters for an algorithm is known as hyper-parameter optimization.

One would assume that this is just a matter of finding and applying the best algorithm and parameters. Unfortunately, this is not exactly the case, the "no free lunch theorem" [WM97] states that "any elevated performance over one class of problems is offset by performance over another class". Meaning that there is no win it all algorithm, an algorithm might be a great solution to one problem, which might be the opposite case for a different problem. It is therefore common for machine learning experts to try out different techniques to the problem and settle for the algorithm with the best performance.

Many different approaches to solving AutoML problems have been presented in the literature: TPOT [OM19], AutoWEKA [THHLB13], and Auto-Sklearn [FKE+15] utilizing diverse methodologies such as Bayesian Optimization, Genetic Programming, and Meta-Learning. Despite the fact that several techniques have been published, it is scientifically interesting to investigate novel metaheuristic approaches for this type of problem, as there is still a need for improvement for current methods.

1.1 Aims of this Thesis

The aim of this thesis to develop an Automated Machine Learning framework that uses metaheuristic algorithms to find the best performing pipeline for a given dataset instance. Moreover, the aim is to explore different well-known metaheuristic algorithms based on local search techniques such as Simulated Annealing, Tabu Search, and Iterated Local Search. Another aim of this thesis is to compare the proposed framework with other state-of-the-art frameworks and find the optimum metaheuristic algorithm settings and components for AutoML.

To conclude, the main research questions of this thesis to be answered are:

- What are the key concepts and methods in AutoML?
- Which metaheuristic algorithm is the most appropriate for this kind of task?
- How would a metaheuristic AutoML framework perform and would it be able to achieve similar and/or better performance to already existing frameworks?
- What are optimal parameters, configurations, and components for metaheuristics algorithms to solve AutoML problems?

1.2 Contribution

The main contributions of this thesis are:

- An in-depth practical and theoretical review of the most recent state-of-the-art AutoML frameworks.
- Based on Simulated Annealing, Iterated Local Search, and Tabu Search we implemented a novel metaheuristic framework for solving various AutoML classification problems. Additionally, we proposed various neighborhood operators, analyzed the different components and their impacts on solving AutoML problems.
- An empirical assessment of the developed framework and state-of-the-art frameworks on a machine learning classification benchmarks containing over 100 datasets, as well as a reference to state-of-the-art results.
- Using specific experiment conditions, the results obtained by state-of-the-art frameworks are matched/improved for the majority of the datasets considered.

1.3 Organization

The second chapter discusses the AutoML concept and offers an outline of similar studies from the literature as well as an in-depth review of the currently most common frameworks.

The third chapter goes into depth about the proposed framework for solving AutoML problems using metaheuristic algorithms. The novel metaheuristic approach's high-level description and AutoML problem-solving approach is then explained in chapter four. In chapter five, an experimental study of using the proposed framework in conjunction with the most common framework mentioned previously is conducted using more than 100 datasets, while also the results of using the proposed framework to solve problem instances are reported and compared to state-of-the-art results. Finally, in chapter six, concluding remarks are given.

CHAPTER 2

Problem Statement and Related Work

While algorithm selection and hyper-parameter tuning are very important for choosing the right model to solve problems, it's not the whole story. The machine learning pipeline consists of more components than just algorithm selection and hyper-parameter tuning, which can be seen in Figure 2.1. The feature engineering step is a critical measure in the machine learning pipeline and can very often have a significant impact on the performance of the model. This is where the experience of the machine learning specialist pays off. Finding the right preprocessing methods, choosing the right feature extraction and selection approaches, and selecting and tuning the right model is the product of experience, domain knowledge, past results and expertise.

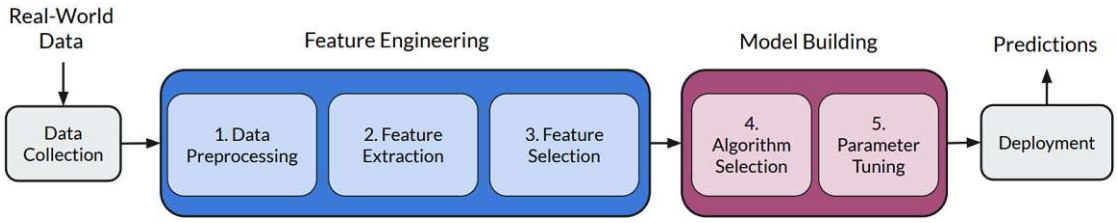

Figure 2.1: A depiction of a typical machine learning pipeline [EMS19]

With the great success of machine learning in many different problems, the demand for good machine learning systems that can be used by non-experts has increased exponentially. Therefore, there is a critical need for automating the process of building good machine learning models. In the recent years, there has been a big surge of techniques and frameworks for solving and automating the process of Combined Algorithm Selection and Hyper-parameter tuning (CASH) in the machine learning domain [EMS19]. The aim of these systems is to make machine learning more accessible for the non-expert.

These systems must be able to choose the correct algorithm, including the correct hyper-parameters, while also being able to apply the right pre-processing steps for a new data set to be successful in practice.

Continuing below, the automated machine learning problem itself will be defined and we'll go through the most common Automated Machine Learning (AutoML) frameworks and break down their respective architectures in detail.

2.1 Problem Statement

Every machine learning algorithm has hyperparameters, and to automatically adjust such hyperparameters to maximize output is the most fundamental activity in AutoML.

Let $\mathcal{A}$ denote a machine learning algorithm with N hyperparameters. We denote the domain of the n-th hyperparameter by Λ_n and the overall hyperparameter configuration space as $\mathbf{\Lambda} = \Lambda_1 \times \Lambda_2 \times \ldots \Lambda_N$. A vector of hyperparameters is denoted by $\lambda \in \mathbf{\Lambda}$, and $\mathcal{A}$ with its hyperparameters instantiated to λ is denoted by $\mathcal{A}_\lambda$ [FH19].

The domain of these values can real-valued (γ in SVC), integers (number of neighbours in kNN), categorical (type of distance to use in kNN) or binary (whether or not to prune decision trees). Furthermore, some of these parameters can be conditional on each other meaning that they may be relevant only beneath another hyperparameter e.g.: pruning error metric is used only if pruning itself is on in decision trees.

The optimization process of finding the best hyper-parameters in regard to a loss function is called hyper-parameter optimization.

Some of the baseline methods in hyper-parameter optimization are grid search, random search and gradient estimation. There are many advantages to general grid search [BB12]: it's easy to implement, there's no technical overhead compared to other optimization techniques, it usually delivers better performance than setting hyperparameters manually and is reliable in low complexity search spaces. There are also different variations in grid search, e.g. random grid search, where optimization values are chosen randomly.

This means that the hyper-parameter optimization problem can be easily extended into more complex optimization problems, if we just add the step we want to optimize as a new top-level parameter. As in our example algorithm selection, creating the Combined Algorithm Selection and Hyper-parameter Optimization Problem (CASH).

Given a set of algorithms $\mathcal{A} = \left\{A^{(1)}, \ldots, A^{(k)}\right\}$ with associated hyperparameter spaces $\mathbf{\Lambda}^{(i)}, \ldots, \mathbf{\Lambda}^{(k)}$, we define the combined algorithm selection and hyperparameter optimization problem (CASH) as computing [THHLB13]:

$$A^*_{\lambda^*} \in \underset{A^{(j)} \in \mathcal{A}, \boldsymbol{\lambda} \in \mathbf{\Lambda}^{(j)}}{\operatorname{argmin}} \frac{1}{k} \sum_{i=1}^{k} \mathcal{L}\left(A^{(j)}_{\boldsymbol{\lambda}}, \mathcal{D}^{(i)}_{\text{train}}, \mathcal{D}^{(i)}_{\text{valid}}\right) \tag{2.1}$$

We note that this problem can be reformulated as a single combined hierarchical hyperparameter optimization problem with parameter space $\mathbf{\Lambda} = \mathbf{\Lambda}^{(1)} \cup \cdots \cup \mathbf{\Lambda}^{(k)} \cup \{\lambda_r\}$, where λ_r is a new root-level hyperparameter that selects between algorithms $A^{(1)}, \ldots, A^{(k)}$ [THHLB13].

2.2 Related Work

There are now several frameworks for automated machine learning that one may employ. Nevertheless, the choice of whatever framework to utilize depends on the task you are attempting to accomplish. The most prevalent AutoML frameworks are broken down into components and discussed in depth in combination with their respective architectures.

2.2.1 TPOT

At its core, TPOT [OUA+16a] is a wrapper for the Python machine learning package, scikit-learn. With respect to hyperparameter optimization, TPOT can only handle categorical parameters; meaning all continuous hyper-parameters have to be discreted, equivalent to grid search. In contrast to grid search, TPOT does not test all the different combinations exhaustively but uses genetic programming to fine-tune an algorithm again.

TPOT uses genetic programming to get the best possible optimization for its machine learning problems. An overview of the full TPOT machine learning process can be found in Figure 2.2.

Genetic Programming

Genetic programming (GP) [GJ05] is, in artificial intelligence, a strategy of evolving algorithms, beginning from a population of inadequate (usually random) algorithms, suited for a specific purpose by adding operations similar to normal genetic processes to the program population. This is basically a heuristic search strategy often described as 'hill climbing', i.e. searching through all programs for an ideal or at least appropriate program.

The operations are: selection of the most suitable reproductive programs (crossover) and mutation based on a predefined fitness test, usually ability at the desired function. The crossover process involves switching random pieces of chosen pairs (parents) to generate new and separate offspring which will become part of the current system generation. Mutation means swapping a random part of a program with another random part of a program. Many systems which are not chosen for reuse are recycled to the next generation from the existing one.

Framework Process Flow

First, the population is selected randomly and 100 samples are taken. Each pipeline is evaluated on the target variable and each pipeline is ranked on the basis of its performance,

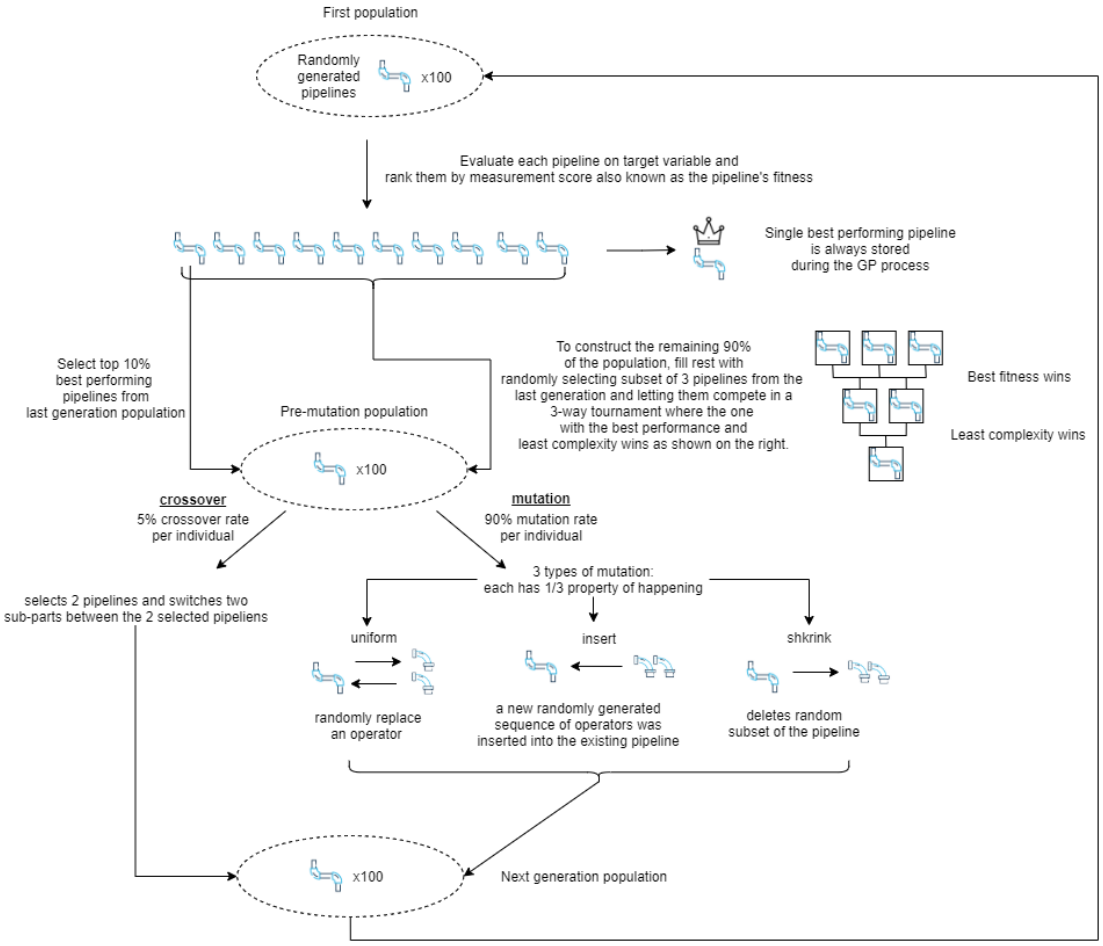

Figure 2.2: TPOT Machine Learning System Overview Diagram

also known as the fitness of the pipeline. The top 10% of the best-performing pipelines are selected from the ranked pipeline pool for the pre-mutation population. The remaining 90% of the pre-mutation population is constructed with randomly selected pipelines from the previous generation and let them compete in a 3-way tournament where the one with the best performance and the least complexity wins.

The mutation occurs after the pre-mutation population is selected. After which there are two options for mutation: crossover and mutation. Crossover has a rate of 5% for each occurrence and, when the event occurs, two pipelines are selected and two sub-parts are switched between the selected pipelines. On the other hand, mutation has a 90% rate per individual, where there are three different types of mutation, all with a 1/3 chance of occurring. Uniform mutation, the pipeline operator is randomly replaced by another random operator. Insert mutation, a newly generated sequence of operators is inserted into the existing pipeline. Shrink mutation, a random subset of the pipeline is removed.

After which 100 new pipelines remain for the population of the next generation. This process has been repeated 100 times. During this, the single best-performing pipeline is always saved.

Machine Learning Pipeline Operators

Being one of the most mature AutoML frameworks, TPOT offers a wide range of classifiers for various machine learning problems. The full list can be found in Table 2.1.

Operator	Methods
Supervised Classifier	DecisionTree, RandomForest, eXtreme Gradient Boosting Classifier (from XGBoost), LogisticRegression, KNearestNeighborClassifier
Feature Preprocessing	StandardScaler, RobustScaler, MinMaxScaler, MaxAbsScaler, RandomizedPC, A Binarizer, PolynomialFeatures
Feature Selection	VarianceThreshold, SelectKBest, SelectPercentile, SelectFwe, Recursive Feature Elimination (RFE)

Table 2.1: TPOT Automated Machine Learning Pipeline Operators [OUA+16b]

Application of TPOT

Being a python package, setting up a TPOT is very simple and user-friendly task. After all the installation pre-requisites have been installed, TPOT can be installed using a single line of code:

```
pip install tpot
```

After which an example of iris dataset can be easily run as shown in Listing 1. Further configuration options can be found in Table 2.2.

Configuration Parameter	Description
generations	Number of iterations to the run pipeline optimization process.
population_size	Number of individuals to retain in the genetic programming population every generation.
offspring_size	Number of offspring to produce in each genetic programming generation.
mutation_rate	Mutation rate for the genetic programming algorithm.
crossover_rate	Crossover rate for the genetic programming algorithm
scoring	Function used to evaluate the quality of a given pipeline for the classification problem.
cv	Cross-validation strategy used when evaluating pipelines.
subsample	Fraction of training samples that are used during the TPOT optimization process.
n_jobs	Number of processes to use in parallel for evaluating pipelines during the TPOT optimization process.
max_time_mins	How many minutes TPOT has to optimize the pipeline.
max_eval_time_mins	How many minutes TPOT has to evaluate a single pipeline.
random_state	The seed of the pseudo random number generator used in TPOT.
config_dict	A configuration dictionary for customizing the operators and parameters that TPOT searches in the optimization process.
template	Template of predefined pipeline structure.
warm_start	Flag indicating whether the TPOT instance will reuse the population from previous calls to fit().
memory	If supplied, pipeline will cache each transformer after calling fit
use_dask	Whether to use Dask-ML's pipeline optimiziations.
periodic_checkpoint_folder	If supplied, a folder in which TPOT will periodically save pipelines in pareto front so far while optimizing.
early_stop	How many generations TPOT checks whether there is no improvement in optimization process.
verbosity	How much information TPOT communicates while it's running.
disable_update_check	Flag indicating whether the TPOT version checker should be disabled.
log_file	Save progress content to a file.

Table 2.2: TPOT Automated Machine Learning Configuration Parameters [noac]

```python
from tpot import TPOTClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
import numpy as np

iris = load_iris()
X_train, X_test, y_train, y_test = train_test_split(
    iris.data.astype(np.float64),
    iris.target.astype(np.float64),
    train_size=0.75,
    test_size=0.25,
    random_state=42,
)
tpot = TPOTClassifier(
    generations=5,
    population_size=50,
    verbosity=2,
    random_state=42
)
tpot.fit(X_train, y_train)
print(tpot.score(X_test, y_test))
```

Listing 1: TPOT iris dataset example [noac]

2.2.2 Auto-WEKA

Auto-WEKA [THHLB13] is a tool which selects and optimizes a combined algorithm over the algorithms of classification and regression in WEKA. More precisely, Auto-WEKA uses model-based optimization techniques, with a particular dataset, to investigate settings on hyperparameters of several algorithms and suggest the best generalization results for the user.

The developers of Auto-WEKA were one of the first to use Bayesian optimization for hyperparameter optimization in the magnitude that Auto-WEKA has implemented. Where the full-fledged machine learning pipeline is being streamlined and turned into a combined algorithm and hyperparameter search problem (CASH). That means that Auto-WEKA only applies another dimension to the hyperparameter search problem by introducing a classifier and feature selection as top-level parameters, therefore expanding the hyperparameter search problem to the CASH problem.

Bayesian Optimization

There are many problems with optimization in machine learning, where the objective function $f(x)$ is a black box function [Moc94, BCdF10a]. We have no analytical expression for f, nor are we familiar with its derivatives. The estimation of the feature is confined to a point x sampling and the potential noisy reaction.

If f would be cheap to evaluate, we could easily search through many points with a normal grid search to get the best performance possible. However, usually the function f is expensive, meaning that an evaluation of f would cost either time or money. Just as in our case, where the training budget is limited, many needless evaluations of f would hinder our performance.

This is the setting where Bayesian methods in optimisation are most effective. With a minimal number of iterations they seek to reach the global optimum or as in our case the best possible configuration λ. The Bayesian optimisation involves a prior confidence in the problem of samples from f and changes the previous versions to a prior which is similar to f. The model used to simulate the target function is known as the surrogate model. Bayesian optimization also uses the acquisition function to direct the sample to areas where the best observation is likely to be improved.

Gaussian processes (GPs) are a common surrogate model for Bayesian optimization. GPs describe a function before and we will use it to combine prior views about the objective function. The GP posterior is cheap to measure and is used to indicate points where samples are supposed to change in the search space.

To summarize, Bayesian optimization [BCDF10b, EFH$^+$13] creates a probabilistic model $\mathcal{M}$ of f, centered on the evaluations of points in f and any previous knowledge accessible. Bayesian optimization uses the acquisition function $a_{\mathcal{M}} : \mathbf{\Lambda} \to \mathbb{R}$, which uses the $\mathcal{M}$ model predictive distribution to quantify the useful knowledge about hyperparameter configuration $\lambda \in \Lambda$ to select its next hyperparameter configuration using $\mathcal{M}$. This function is maxed over $\mathbf{\Lambda}$ to select the λ configuration that is most useful to evaluate next.

Sequential Model-based Algorithm Configuration (SMAC)

One of the more popular GP optimization methods is SMAC [EFH$^+$13]. Sequential Model based Algorithm Configuration is a general framework for minimizing black-box functions f. Random forests are used by SMAC in modeling $p_{\mathcal{M}}(f \mid \boldsymbol{\lambda})$ as a Gaussian Distribution whose mean and variance are the empirical mean and variance over the predictions of forest's trees. To save resources, for hyper optimization issues with cross validation, SMAC assesses loss of configurations at a single fold at a time. Configurations are compared with each other based on the evaluation of the folds on each. SMAC works with categorical, continuous and conditional parameters.

It is used to configure many combinatorial optimization algorithms and it was the best performing optimizer for Auto-WEKA [THHLB13]. The pseudo code of the algorithm

and how it works can be seen in Algorithm 2.1.

Algorithm 2.1: Sequential Model-based Bayesian Optimization (SMBO) [BBBK11]

while *time budget for optimization has not been exhausted* **do**
  $\boldsymbol{\lambda} \leftarrow$ determine candidate configuration from $\mathcal{M}$
  $i \leftarrow$ select cross-validation fold
  Compute c=L$\left(A_{\lambda}, \mathcal{D}^{(i)}_{\text{train}}, \mathcal{D}^{(i)}_{\text{valid}}\right)$
  H $\leftarrow \mathcal{H} \cup \{(\boldsymbol{\lambda}, c)\}$
  Update $\mathcal{M}$ given $\mathcal{H}$
end
return λ *from* $\mathcal{H}$ *with minimal* c *across cross-validation folds*

Framework Process Flow

Auto-WEKA provides a wide variety of hyperparameter search spaces by introducing 768 potential hyperparameters. Many of them come from the usual classifiers, but also from newly introduced meta-methods and ensemble methods. Auto-Weka was developed with classification in mind, but can quickly be applied to regression problems as well.

The biggest challenge of such a large hyperparameter search space is that most of the learning algorithms further expose hyperparameters specific to the said learning algorithm. For example, specific algorithm hyperparameters are used to describe the number of neighbors to be included in the KNN, the number of leaves to be used in the decision tree, the kernel function to be used in the SVM. These hyperparameters are treated in a "outer loop" and are handled as hierarchical parameters.

To solve this problem Auto-Weka uses Bayesian optimization. in particular Sequential Model-Based Optimization (SMBO), a versatile stochastic optimization framework that can work explicitly with both categorical and continuous hyperparameters, and that can exploit hierarchical structure stemming from conditional parameters [THHLB13].

Auto-WEKA is considered to be agnostic to the optimizer, but SMAC ended up being the default optimization algorithm due to better performance.

Machine Learning Pipeline Operators

Having WEKA as an existing system with a broad base of Auto-WEKA classifiers to use, Auto-WEKA provides the widest range of algorithms offered, including ensemble and meta algorithms as well. The same is true of the feature preprocessing methods. A full list can be found in Table 2.3. The configuration options can be found in Table 2.4.

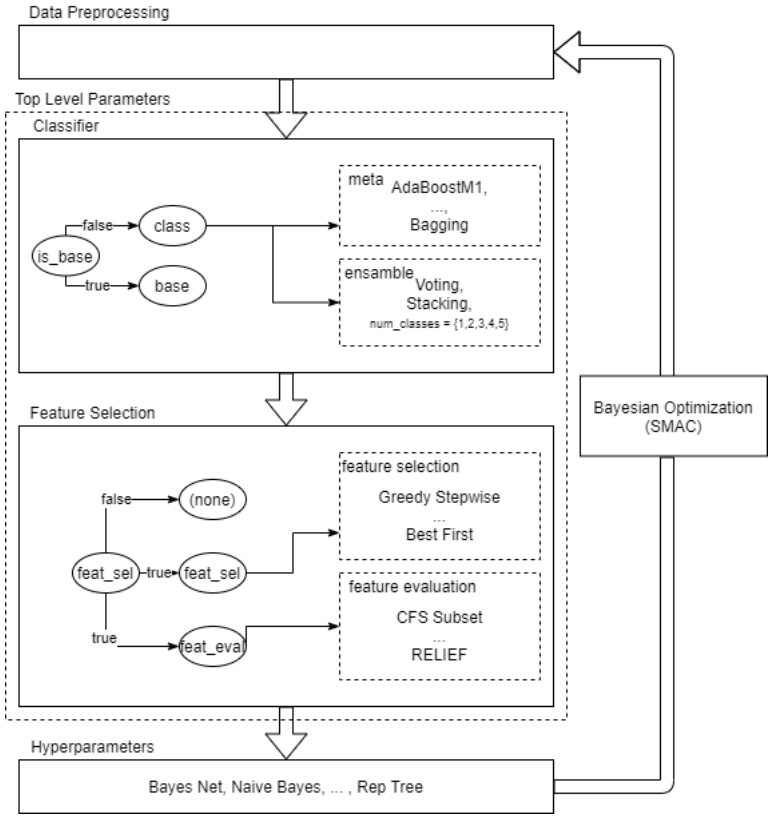

Figure 2.3: Auto-Weka Machine Learning System Overview Diagram

Application of Auto-WEKA

Running Auto-WEKA is very straightforward through the WEKA application. Just install it through in-app package manager and you're good to start experimenting with your own pipelines.

Operator	Methods
Supervised Classifier	Bayes Net, Naive Bayes, Naive Bayes Multinomial Gaussian Process, Linear Regression, Logistic Regression, Single-Layer Perceptron, Stochastic Gradient Descent, SVM, Simple Linear Regression, Simple Logistic Regression, Voted Perceptron, KNN, K-Star, Decision Table, RIPPER, M5 Rules, 1-R, PART, 0-R, Decision Stump, C4.5 Decision Tree, Logistic Model Tree, M5 Tree, Random Forest, Random Tree, REP Tree, Locally Weighted Learning*, AdaBoost M1*, Additive Regression*, Attribute Selected*, Bagging*, Classification via Regression*, LogitBoost, MultiClass Classifier Random Committee*, Random Subspace*, Voting+, Stacking+
Feature Preprocessing	Best First*, Greedy Stepwise*, Ranker*, CFS Subset Eval, Pearson Correlation Eval, Gain Ratio Eval, Info Gain Eval, 1-R Eval, Principal Components Eval, RELIEF Eval, Symmetrical Uncertainty Eval

Table 2.3: Auto-Weka Automated Machine Learning Pipeline Operators [THHLB13]

Configuration Parameter	Description
`seed`	the seed for the random number generator
`memLimit`	the memory limit for runs (in MiB)
`parallelRuns`	the number of runs to perform in parallel
`numDecimalPlaces`	The number of decimal places to be used for the output of numbers in the model.
`batchSize`	The preferred number of instances to process if batch prediction is being performed.
`timeLimit`	the time limit for tuning (in minutes)
`debug`	If set to true, classifier may output additional info to the console.
`nBestConfigs`	How many of the best configurations should be returned as output
`doNotCheckCapabilities`	If set, classifier capabilities are not checked before classifier is built
`metric`	the metric to optimise

Table 2.4: Auto-WEKA Automated Machine Learning Configuration Parameters

2.2.3 Auto-Sklearn

Auto-Sklearn offers supervised machine learning out-of-the-box. Constructed around the machine learning library scikit-learn, Auto-Sklearn searches for the right learning algorithm and optimizes its hyperparameters for a new machine learning dataset. Auto-Sklearn extends its idea of configuring an efficient global optimization while extending it with extra steps like meta learning and ensemble building. Auto-Sklearn uses meta-learning to define and use information gathered in the past to accelerate the optimization process. Auto-Sklearn constructs a ensemble of all models evaluated in the global optimization process to enhance generalization. It wraps 15 classification algorithms, 14 feature algorithms, and handles data scaling, categorical encoding and also handles missing values.

Meta-Learning

Meta-Learning [KH00], is a sub-field of machine learning where automatic learning algorithms are applied on metadata about machine learning experiments. The main goal is to use such metadata to understand how automatic learning can become flexible in solving learning problems, hence to improve the performance of existing learning algorithms or to learn (induce) the learning algorithm itself, hence the alternative term learning to learn.

Ensemble Learning

Ensemble Learning [D$^+$02, ZM12, Pol12], is a machine learning method where multiple basic learners are used together to overcome each other's weaknesses and solve machine learning problems. Ensemble learning is split into:

- Bagging - Boostrap Aggregating, models are trained in parallel to many different subsets of data and then averaged to make meaningful predictions.
- Boosting - Improvements involve slowly teaching a weak learner where the paradigm relies intuitively on experiences that its previous variations had trouble modelling through time. One very popular boosting algorithm is Adaboost [Sch13].

Framework Process Flow

The two new additions are Meta-Learning and ensemble building. Meta-Learning is just a way to learn from previous knowledge. Trying to simulate the real-world machine learning process, where experienced and knowledge-based practitioners can arbitrarily better choose machine learning classifiers based on the dataset they are currently working on. The Auto-Sklearn team tackled this problem by taking 160 datasets from the UCI Machine Learning repository, conducting supervised machine learning, and extracting meta-features from the results for each dataset. This newly acquired meta information is then used to start a hyperparameter search where, once a new dataset is received for

training, meta features are extracted and the nearest 25 results are taken as starting points for a hyperparameter search space through KNN.

On the other hand, the Ensemble Building Step makes the multiple training of models advantageous during training by combining them into Ensemble Methods. Where, during normal Bayesian hyperparameter optimization, all other models beside the best are lost. This is very important because the difference in prediction error is often very low in high ranking models. Although it is known that the stacking of classifiers often results in a lot of over-fitting of training data, the proposed method seeks to avoid this by using ensemble selection. To put it in a nutshell, ensemble selection is a greedy procedure which starts from an empty ensemble and adds each model which minimizes the validation loss of the ensemble (with uniform weight but which allows repeatings).

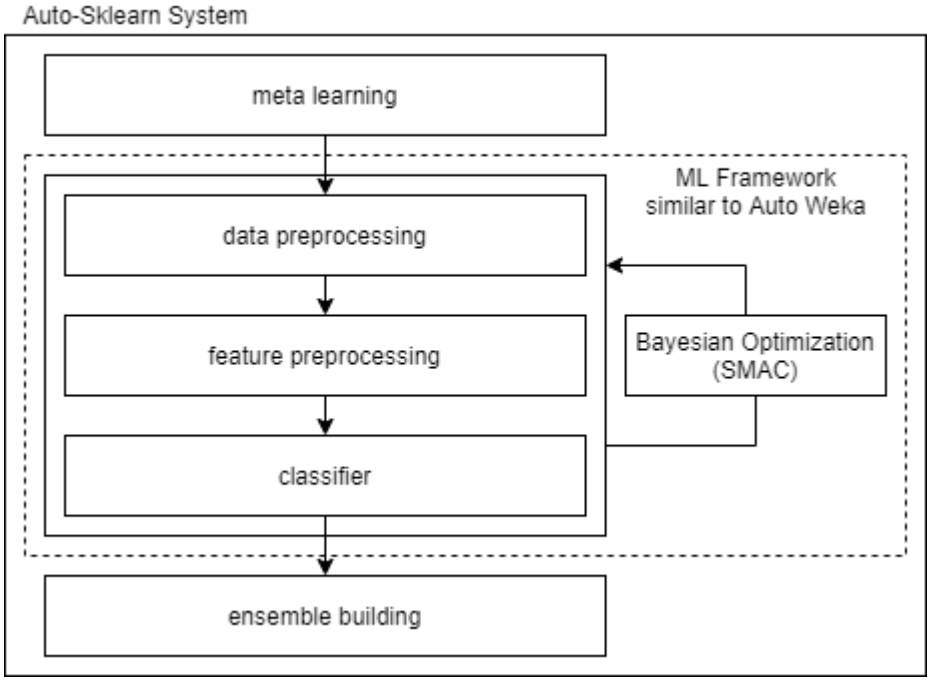

Figure 2.4: Auto-Sklearn Machine Learning System Overview Diagram

Machine Learning Pipeline Operators

Auto-Sklearn also offers a wide range of supervised classifiers and feature preprocessing methods. The full list can be found in Table 2.5

Application

Being a python package, setting up Auto-Sklearn is more of a problem than one might think at first glance. Failing support for Windows and MacOS, it is limited to Ubuntu distributions only, while also being dependent on some older versions of scikit-learn and

Operator	Methods
Supervised Classifier	AdaBoost (AB), Bernoulli naive Bayes, decision tree (DT), extreml. rand. trees, Gaussian naïve Bayes, gradient boosting (GB),KNN, LDA, linear SVM, kernel SVM, multinomial naive Bayes, passive aggressive, QDA, random forest (RF), Linear Class. (SGD)
Feature Preprocessing	Extreml. rand. trees prepr., fast ICA, feature agglomeration, kernel PCA, rand. kitchen sinks, linear SVM prepr., no preprocessing, nystroem sampler, PCA, polynomial, random trees embed., select percentile, select rates, one-hot encoding , imputation, balancing, rescaling

Table 2.5: Auto-Sklearn Automated Machine Learning Pipeline Operators [FKE+15]

pandas making it a problem to maintain with other AutoML systems in one environment. After getting over all that one can install Auto-Sklearn through pip:

```
pip install auto-sklearn
```

After which an example of iris dataset can be easily run as shown in the Listing 2. Further configuration options can be found in Table 2.6.

Configuration Parameter	Description
time_left_for_this_task	Time limit in seconds for the search of appropriate models.
per_run_time_limit	Time limit for a single call to the machine learning model.
initial_configurations_via_metalearning	Initialize the hyperparameter optimization algorithm with this many configurations which worked well on previously seen datasets.
ensemble_size	Number of models added to the ensemble built by Ensemble selection from libraries of models. Models are drawn with replacement.
ensemble_nbest	Only consider the ensemble_nbest models when building an ensemble.
max_models_on_disc	Defines the maximum number of models that are kept in the disc.
ensemble_memory_limit	Memory limit in MB for the ensemble building process.
seed	Used to seed SMAC
ml_memory_limit	Memory limit in MB for the machine learning algorithm.
include_estimators	If None, all possible estimators are used.
exclude_estimators	If None, all possible estimators are used.
include_preprocessors	If None all possible preprocessors are used.
exclude_preprocessors	If None all possible preprocessors are used
resampling_strategy	how to to handle overfitting, might need 'resampling_strategy_arguments/'
resampling_strategy_arguments	Additional arguments for resampling_strategy.
tmp_folder	folder to store configuration output and log files.
output_folder	folder to store predictions for optional test set.
delete_tmp_folder_after_terminate	remove tmp_folder, when finished
delete_output_folder_after_terminate	remove output_folder, when finished
shared_mode	Run smac in sharedmodelnode.
n_jobs	The number of jobs to run in parallel for fit().

```python
from autosklearn.classification import AutoSklearnClassifier
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris
from sklearn.metrics import accuracy_score

iris = load_iris()
X_train, X_test, y_train, y_test = train_test_split(
    iris.data.astype(np.float64),
    iris.target.astype(np.float64),
    train_size=0.75,
    test_size=0.25,
    random_state=42,
)
automl = AutoSklearnClassifier()
automl.fit(X_train, y_train)
y_hat = automl.predict(X_test)
print("Accuracy score", accuracy_score(y_test, y_hat))
```

Listing 2: Auto-Sklearn iris dataset example [noaa].

2.2.4 Hyperopt-Sklearn

Hyperopt-Sklearn [KBE14] is a new software project that provides automatic algorithm configuration of the scikit-learn machine learning library. It takes the view that the choice of classifier and even the choice of preprocessing module can be taken together to represent a single large hyperparameter optimization problem. Hyperopt is used to define a search space that encompasses many standard components (e.g. SVM, RF, KNN, PCA, TF-IDF) and common patterns of composing them together.

Framework Process Flow

The Hyperopt framework offers optimization algorithms for search spaces that arise in algorithm configuration. These spaces are characterized by a variety of types of variables (continuous, ordinal, categorical), different sensitivity profiles (e.g. uniform vs. logarithmic), and conditional structure (when there is a choice between two classifiers, the parameters of one classifier are irrelevant when the other classifier is chosen). To use Hyperopt, a user must define/choose three things [KBE14]:

1. a search domain,
2. an objective function,
3. an optimization algorithm.

Where the search domain is defined by random variables which distributions are chosen in the manner of having higher probability for more promising results. This domain includes predefined Python functions and operators for convenience of the objective function.

The objective function averages these random variables together to a scalar score that will aim to minimize the optimization algorithm. Having selected a search domain, an objective function, and an algorithm for optimization, the 'fmin' method of Hyperopt performs optimization and saves the search results into a database (for instance either a plain Python list or an instance of MongoDB).

Machine Learning Pipeline Operators

Hyperopt-Sklearn offers the usual wide range of supervised classifiers, regressors, and feature preprocessing methods. While, also as a highlight, offering some preprocessing methods for natural language processing (NLP) such as the TF-IDF preprocessing method. The full list can be found in Table 2.7

Operator	Methods
Supervised Classifier	svc, svc_linear, svc_rbf, svc_poly, svc_sigmoid, liblinear_svc, knn, ada_boost, gradient_boosting, random_forest, extra_trees, decision_tree, sgd, xgboost_classification, multinomial_nb, gaussian_nb, passive_aggressive, linear_discriminant_analysis, quadratic_discriminant_analysis, one_vs_rest, one_vs_one, output_code
Supervised Regressor	svr, svr_linear, svr_rbf, svr_poly, svr_sigmoid, knn_regression, ada_boost_regression, gradient_boosting_regression, random_forest_regression, extra_trees_regression, sgd_regression, xgboost_regression
Preprocessing	pca, one_hot_encoder, standard_scaler, min_max_scaler, normalizer, ts_lagselector, tfidf, rbm, colkmeans

Table 2.7: Hyperopt-Sklearn Automated Machine Learning Pipeline Operators [noa20].

Application

Hyoperopt-Sklearn is divided into two packages, hyperopt where the configuration search algorithms are located and hpsklearn where the AutoML system, which is based on the search algorithms, is located. Both are needed to be able to configure the Hyopert-Sklearn AutoML framework. Once both are installed, an example of iris dataset can be easily run as shown in Listing 3. Further configuration options can be found in Table 2.8.

Configuration Parameter	Description
`preprocessing`	This should evaluate to a list of sklearn-style preprocessing modules (may include hyperparameters)
`ex_preprocs`	This should evaluate to a list of lists of sklearnstyle preprocessing modules for each exogenous dataset.
`classifier`	This should evaluates to sklearnstyle classifier (may include hyperparameters).
`regressor`	This should evaluates to sklearnstyle regressor (may include hyperparameters)
`algo`	hyperopt suggest algo (e.g. rand.suggest)
`max_evals`	Fit() will evaluate up to thismany configurations. Does not apply to fit_iter, which continues to search indefinitely.
`loss_fn`	A function that takes the arguments (y_target, y_prediction) and computes a loss value to be minimized
`continuous_loss_fn`	When true, the loss function is passed the output of predict_proba() as the second argument.
`trial_timeout`	Kill trial evaluations after this many seconds.
`fit_increment`	Every this-many trials will be a synchronization barrier for ongoing trials, and the hyperopt Trials object may be check-pointed.
`fit_increment_dump_filename`	Periodically dump self.trials to this file (via cPickle) during fit() Saves after every 'fit_increment' trial evaluations.
`seed`	If int, the integer will be used to seed a RandomState instance for use in hyperopt.fmin.
`use_partial_fit`	If the learner support partial fit, it can be used for online learning.
`refit`	Refit the best model on the whole data set.

Table 2.8: Hyperopt-Sklearn Automated Machine Learning Configuration Parameters [noa20].

```python
from hpsklearn import HyperoptEstimator, any_classifier, any_preprocessing
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from hyperopt import tpe
import numpy as np

iris = load_iris()
X_train, X_test, y_train, y_test = train_test_split(
    iris.data.astype(np.float64),
    iris.target.astype(np.float64),
    train_size=0.75,
    test_size=0.25,
    random_state=42,
)
estim = HyperoptEstimator(
    classifier=any_classifier("my_clf"),
    preprocessing=any_preprocessing("my_pre"),
    algo=tpe.suggest,
    max_evals=100,
    trial_timeout=120,
)
estim.fit(X_train, y_train)
print(estim.score(X_test, y_test))
```

Listing 3: Hyperopt-Sklearn iris dataset example [noa20].

2.2.5 Google Cloud AutoML Tables

Google Cloud AI [Bis19] offers cloud services for businesses and individuals to leverage pre-trained models for custom artificial intelligence tasks through the use of REST APIs. It also exposes services for developing custom models for domain use cases such as AutoML Vision for image classification and object detection tasks and AutoML tables to deploy AI models on structured data.

Neural Architecture Search

With the recent success of Deep Learning and its different types of neural networks in all areas of Machine Learning [Den14], their fine tuning to achieve the best possible performance has become one of the top priorities in the field, leading to the creation of a sub-branch of hyperparameter optimization called Neural Architecture Search (NAS). NAS [EHH18] is a technique for automating the design of artificial neural networks (ANN), a widely used model in the field of machine learning. As a result, becoming one of the most complex optimization problems there is due to the very complex nature of

the Neural Networks themselves. And, due to the many different parameters and their very wide range of values they can take.

Framework Process Flow

Although much of the actual search algorithm process is left in the dark, allowing us only to assume the details. In the presentation of the AutoML Tables Introduction, it is said that [TyHCPP]: the search algorithm is based on the Neural Network Architecture Search already existing at Google Brain, from which they have extended their search algorithm with tree structures and automated feature engineering methods.

While, on the other hand, Google Cloud has made sure that everyone can get started on their platform without any prior machine learning background. After registering for the beta and accessing the AutoML Tables Console, the process of inputting your data until you get your prediction output is very straight forward.

One starts by giving their data either as a CSV or BigDataQuery, after which one defines their target variable and can view different statistics for the target variable, such as correlation with other variables and missing data. The biggest strength of AutoML Tables is that it can also handle missing data, a feature that is missing from many other AutoML systems. Then one can actually continue training their model , allowing for some configuration, such as train-test splitting, and for how long to train their models.

A lot of the specifics of the actual search algorithm are not published, the process is divided into three phases from the diagram presented in the opening speech of the AutoML Tables [TyHCPP]:

1. Preprocessing - Assuming where the missing data is being handled and different scales are being tested.
2. Architecture Search and Tuning - The most important step in which all feature selection and embedding is performed, where the model type is selected and trained on the basis of the loss function.
3. CV, Bagging, Ensemble - Cross-validation of results and creation of ensembles for the best possible performance.

After this is done, the model can be evaluated on the basis of the validation set on various performation metrics. With which model, one can either deploy it using their integrated web services from Google, or use it to predict further new test data, either via BigQuery or CSV tables.

Machine Learning Pipeline Operators

Although, not many details are shown to the public regarding Google's Cloud AutoML Tables approach, some of the pipeline operators are mentioned in the documentation of the system.

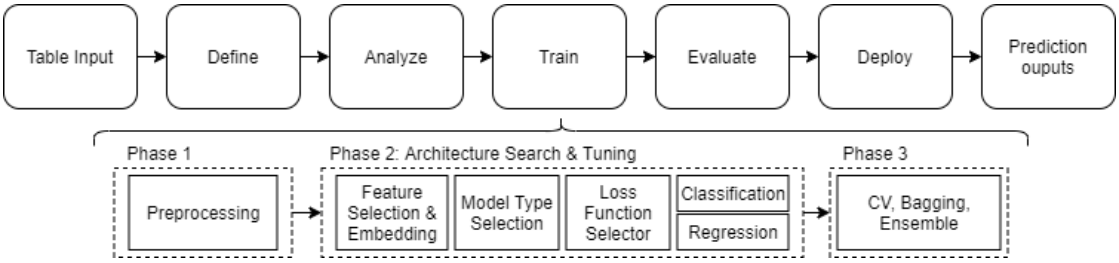

Figure 2.5: Google AutoML Tables System Overview Diagram

Operator	Methods
Supervised Model	Linear, Feedforward deep neural network, Gradient Boosted Decision Tree, AdaNet, Ensembles of various model architectures
Feature Engineering	Normalize and bucketize numeric features, Create one-hot encoding and embeddings for categorical features, Perform basic processing for text features, Extract date- and time-related features from Timestamp columns

Table 2.9: Google AutoML Tables Automated Machine Learning Pipeline Operators [noab]

Application

It's very easy to get going in the Google Cloud environment, and Google also offers a $300 registration bonus to get yourself going. To be noted is that Google AutoML Tables defined itself as being on the more expensive side of things, with a 19$ cost per hour of model training. Most of the configuration options that can be found in Table 2.9 are more for the top-level parameters of the pipeline, considering all of the specifics regarding the optimization algorithm are hidden and done by AutoML Tables.

2.2.6 Amazon SageMaker Autopilot

Similar to its biggest rival Google, Amazon recently also started providing a scalable AutoML solution called Amazon SageMaker Autopilot [DPI+20]. A fully managed system providing an automated machine learning solution that can be modified when needed. Given a tabular dataset and the target column name, Autopilot identifies the problem type, analyzes the data and produces a diverse set of complete machine learning pipelines including feature preprocessing and machine learning algorithms, which are tuned to generate a leader-board of candidate models.

Configuration Parameter	Description
`Data Split`	The way your dataset is split between training, validation, and test subsets.
`Weight Column`	By default, each row in your dataset is weighted equally. To create a custom weighting scheme, add a column to your training dataset with numeric weights, then select it as the weight column.
`Optimization Objective`	Select different optimization metrics.
`Early Stopping`	Ends model training when Tables detects that no more improvements can be made.
`Budget`	Amount of hours to train model for (number between 1 and 72).
`Input Feature Selection`	Sub-select only a part of your features.

Table 2.10: Google AutoML Tables Configuration Parameters [noab]

CHAPTER 3

MetaheuristicSklearn - An AutoML Framework

This chapter will include a brief summary and introduction to MetaheuristicSklearn, an automated machine learning framework developed as part of this thesis. To recap the definition given in the previous chapter: Machine Learning is the process by which machines can learn how to make decisions based on input data and their algorithms. This definition might seem vague but it captures the basic idea of what machine learning entails. This leads us to the definition of automated machine learning (AutoML) as the process of (automatically) building machine learning pipelines from raw data. This is what the proposed framework named as MetaheuristicSklearn does.

The aim of this framework is to help computer programmers without deep understanding of machine learning (with a focus on Python programmers) in the task of automating machine learning. The scope of this thesis is limited to supervised learning. This means that the framework can only work on data sets that contain a nominal target variable. As we explained earlier, the word supervised implies that the target variable has been manually provided by a human expert.

This chapter will include an in-depth description of MetaheuristicSklearn, which offers an AutoML framework that is used for supervised machine learning but employs metaheuristic algorithms to find the optimal feature selection method, scaling method, classifier and their respective hyperparameters, also known as AutoML tasks. It will be able to employ both supervised classification and regression learning, but initially the framework only supports supervised classification. The framework can be used for a wide variety of supervised machine learning problems. Figure 3.1 depicts a high-level description of the system that has been implemented.

The main components of the framework are:

- Instance and solution representation
- Initialization and base estimator evaluation
- Automated preprocessing, feature selection, model selection, and hyperparameter tuning through metaheuristic algorithms.

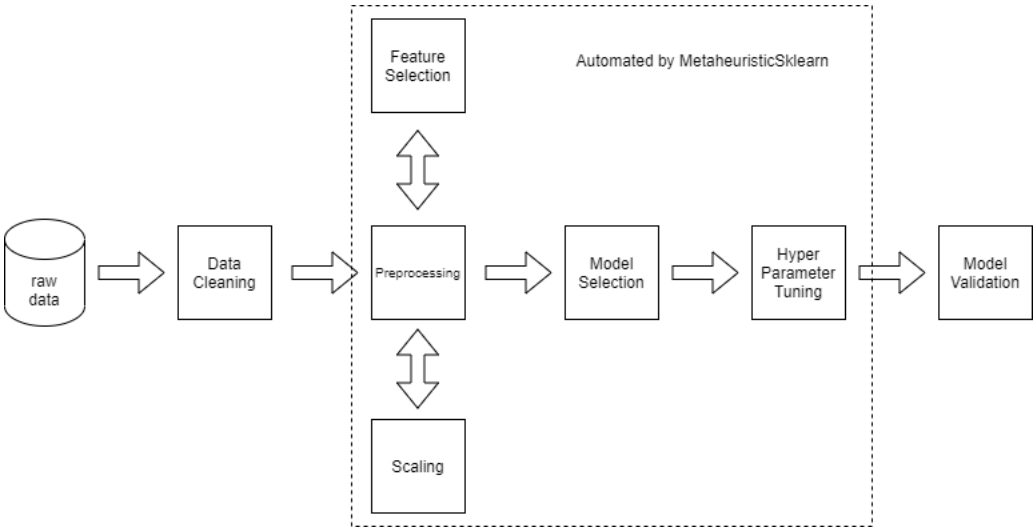

Figure 3.1: High-level System Design Diagram that highlights the framework components

The main advantage of MetaheuristicSklearn is the ease with which it can implement new methods and algorithms without having to re-create everything from scratch every time. However, there are already many feature selection methods, scaling methods, and estimators included from scikit-learn, and also the three metaheuristic algorithms that make use of those methods which will be described in more detail in Chapter 4.

This chapter will discuss how the AutoML framework was created by modularizing its functionality into separate components, and give a more holistic approach to the framework by examining its overall architecture and function.

3.1 Instance and Solution Representation

An instance is a specific example of a problem. In the domain of machine learning this is also known as the dataset. An instance may be represented as a vector, matrix or table consisting of the features of the instance and the corresponding labels. This is also known as the training data set. The training data refers to the set of instances in which we are trying to find the best solution. This best configuration which we call solution is found by using AutoML. The training data is generally set up as an input X and output Y matrix with a column for each feature in the dataset, and a row for every possible output values Y.

3.1.1 Instance Representation

Each instance is represented by one single target function/variable that is of interest and many variables are given as input. The framework extrapolates this as an input matrix and a target vector. Inside the framework this is used for finding the best possible solution configuration to the training data of that instance. An example of how such a dataset can be called in connection to the framework is shown in Listing 4, more precisely in rows 5-7.

The system state represents everything that the system needs to know about an instance at any point in its lifetime. The state, therefore, is composed of the instance and all previous steps and their parameter configurations.

```
1  from sklearn import datasets
2  from mhsklearn import SimulatedAnneal
3
4  # Load the Iris data set
5  iris = datasets.load_iris()
6  X = iris.data
7  y = iris.target
8
9  # Initialize Simulated Annealing and fit
10 sa = SimulatedAnneal()
11 sa.fit(X, y)
12
13 # Print the best score and the best config
14 print(
15     "Best score", sa.best_score,
16     "Best pipeline config", sa.best_pipeline_configuration,
17 )
```

Listing 4: Initialization MetaheuristicSklearn Example

3.1.2 Solution Representation

In the AutoML case of an problem a solution differs slightly from a traditional machine learning solution where only the target variable is considered. In AutoML the solution represents the whole pipeline configuration including all the necessary preprocessing steps such as feature selection, scaling selection, model selection and hyperparameter tuning steps to conclude to the final model solution.

Considering that the solution consists of multiple steps, each step having multiple parameters, each solution consists of a series of steps, in the order applied, and their respective configurations. Listing 5 shows a solution of an example instance. The

solution is represented as a list of tuples. Each tuple is one step in the order applied. In our example, we can see that in line two the scaling method is defined, line three displays the selected feature selection method, lines 4-12 are part of the estimator and it's configuration, and lines 12-12 are the hyperparameters.

The framework makes heavy use of sklearn's pipeline sub-module, which provides a convenient way of running multiple transformation functions in sequence per problem instance, and also allows us to concatenate multiple transformations into one object we can evaluate upon.

```
1  {
2     "scaling":"passthrough",
3     "feature_selection":SelectKBest(k=4),
4     "est":SVC(C=1505.008120090211,
5     coef0=0.33333333333333326,
6     degree=4,
7     gamma=0.0019531249999999996,
8     kernel="poly",
9     max_iter=100000000.0,
10    tol=0.1),
11    "hyperparams":{
12       "SVC__C":1505.008120090211,
13       "SVC__kernel":"poly",
14       "SVC__degree":4,
15       "SVC__gamma":0.0019531249999999996,
16       "SVC__coef0":0.33333333333333326,
17       "SVC__shrinking":True,
18       "SVC__tol":0.1,
19       "SVC__max_iter":100000000.0
20    }
21 }
```

Listing 5: Solution Representation Example

3.2 Initialization and the Base Estimator

The implemented metaheuristic algorithms all share similarities, and have a similar procedure at the beginning of their work. Therefore, an initialization function, which will be described in more detail in Subsection 3.2.1, is created to perform all these actions in a uniform way, and a base estimator `base_estimator` is used while training, described in 3.2.2. The `base_estimator` is stored inside the data structure of the framework, and is instantiated directly during initialization.

Table 3.1: Shared Parameters for all metaheuristic algorithms, descriptions, and their default values

Parameter	Description	Default Value
`random_seed`	Random Seed for reproducibility	`42`
`scoring`	Scoring metric to use for inside iteration performance evaluation	`"f1_macro"`
`max_runtime`	Time after which to stop looking for a better solution	`3600`
`cv`	Number of folds to use inside iteration cross validation	`5`
`verbose`	Print more information	`False`
`retrain`	Retrain the best performimg pipeline on the whole dataset given	`True`

Table 3.2: Best variables accessible after having finished the optimization process, and their descriptions

Variable	Description
`best_score`	Best cross-validated score calculated during the optimization process
`best_hyperparams`	Best hyper parameters for each classifier used during the optimization process
`best_pipeline_configuration`	Best performing pipeline configuration found
`best_pipeline`	Best sklearn pipeline object of the `best_pipeline_configuration`
`total_iter`	The amount of iterations completed
`grid_scores`	List of gridscores for each iteration
`runtime`	Optimization runtime in seconds

3.2.1 Initialization

The initialization function is used to perform all the necessary actions before the main function of each metaheuristic algorithm is called, this includes calling the base estimator evaluation and calculating a first round evaluation, randomly initializing all of the model hyperparameters and creating the necessary variables. It is important to note that the framework starts by using the baseline as the initial state, and then randomly moves to the next solution configuration to be evaluated.

3.2.2 The Base Estimator

To evaluate how well a set of possible solutions fits the problem, an initial evaluation pipeline consisting of no feature selection, a standard scaler, and an SVM classifier was defined, presenting itself as a starting baseline to which further configurations created from the metaheuristics can be compared against. The initialization in this context only deals with a single problem instance and sets the stage for the scaling selection, feature selection, model selection, and hyperparameter tuning.

Before going into detail for the automated machine learning inside the framework, some implementation details that are valid for all algorithms are going to be defined. All of these are included in the `BaseEstimator` class of the framework and are described in more detail in Table 3.1. It creates a foundation for the specific metaheuristic algorithms to be implemented but also presents itself a good starting point for other search algorithms in the future. A high level overview of this class has been listed in the Listing 6.

In Listing 6 we can see the BaseEstimator class and how it has been implemented inside the MetaheurisitcSklearn framework. In lines 2-5 we can see the `__init__` function of the class, which does the initialization of the parameters to the instance variables. Continuing to line seven, where the `fit` function is defined, which is called independent of the metaheuristic used. It is responsible of generating the pipeline grid as seen in line nine, selecting initial random values for each hyperparameter in lines 12-15, updating the current configuration with line 18, computing the initial score with the `fit_score` function in line 21, and then finally updating the global variables necessary after line 24.

Considering the important role reproducibility present in an experiment a `random_seed` parameter is implemented in the framework which is used for the metaheuristic optimization algorithms and for generating the cross-validation splits inside each iteration. The default value is set to 42, which means that the same random seed is generated each time the framework is used, but it can be changed to any number.

To control for how long an optimization algorithm searches for better pipeline configurations, the `max_runtime` parameter was introduced. This is a hard upper bound for the maximum time a search algorithm should take and is set to 3600 (seconds) in this version, but can be changed through the `max_runtime` parameter.

For debugging purposes a verbose parameter was introduced as well, this parameter controls how many values are printed, the default value is False. Logging inside the

```python
class BaseEstimator:
    def __init__(self,random_seed=42,...):
        # Initialize Instace Variables
        self.random_seed = random_seed
        ...

    def fit(self, X, y):
        # Setup hyperparameters
        self.pipeline_grid = self.generate_pipeline_grid(num_cols)

        # Select initial random values for each hyperparameter
        for est, hyperparams in self.pipeline_grid["hyperparams"].items():
            self.old_hyperparams[est] = dict(
                (k, np.random.choice(val)) for k, val in hyperparams.items()
            )

        # Update configuration
        self.old_pipeline_configuration = {...}

        # Compute the initial score
        self.old_score, self.old_std = CVFolds(...).fit_score(X, y)

        # Variables to hold the best score, hyperparams and configuration
        self.best_score = self.old_score
        ...

    ...
```

Listing 6: High Level overview of the BaseEstimator Class

metaheuristic-sklearn framework makes use of python's default logging module and therefore supports level logging.

In addition to the variables mentioned above a retrain parameter is also defined to control if the best performing pipeline is retrained on the whole dataset without any cross-validation after the search algorithm is completed. The default value of retrain is true, which means that that the best performing pipeline is always refitted on the instance given.

In this work we use a 5-Fold Mean Cross Validated score. Meaning that the data is split into 5 folds, and each fold is once used as a test set and the remaining folds as test sets, then the 5 scores obtained are averaged. This allows for every step of the classification pipeline to be distributed evenly on the train and test sets. Thereby making

every iteration independent from each other, not favoring any individual split where there could be an outlier of performance evaluation.

The metric used for this evaluation is based on the scoring parameter, which is the average of the performance values resulted from each fold of the cross-validation. The default scoring method is `"f1_macro"`, as seen in Formula 3.1, which determines the unweighted mean of the metrics for each label using p-precision and r-recall [YL99]. However, does not account for label imbalance. Which is usually the desired result in underrepresented classes. All algorithms also accept optional parameters that can be used to alter the behavior of an algorithm.

$$F_1(r, p) = \frac{2rp}{r + p}. \tag{3.1}$$

After an optimization algorithm has finished searching, and if desired has refitted the best performing pipeline on the whole instance provided, one can also access various other information from the trained model object via the public `best_` variables, such as best score, best hyperparameters for each classifier, best pipeline configuration, best pipeline, list of grid scores and the runtime. A detailed list of these variables can be found in Table 3.2.

The next sub-sections look at the individual components of the automated machine learning part inside the algorithms and what their specificities are.

3.3 Automated Machine Learning

Scikit-Learn was chosen, as one of the most well-developed and used machine learning libraries, to build robust AutoML systems as the underlying ML platform. It provides a wide variety of well-developed ML algorithms and is easy to use for professionals as well as beginners. This also resulted in the name of the resulting framework of this thesis, MetaheuristicSklearn.

It is implemented in python as a modular framework of metaheuristics that can be used to solve an AutoML problem. The idea is to provide a variety of metaheuristic algorithms, so the user can easily select the most suitable algorithm for his/her problem, without needing to know how it works under the hood. The code is flexible and can be used for a wide variety of supervised classification problems in the machine learning domain.

Currently a part of the framework is also the preprocessing steps, where currently feature selection and scaling are supported. These are encoded as top-level parameters into the framework and are independent of other steps, which allows easy integration of other preprocessing steps if wanted.

The next sections will give more context and details as to how the visualization in Figure 3.2 is created and what the individual components do.

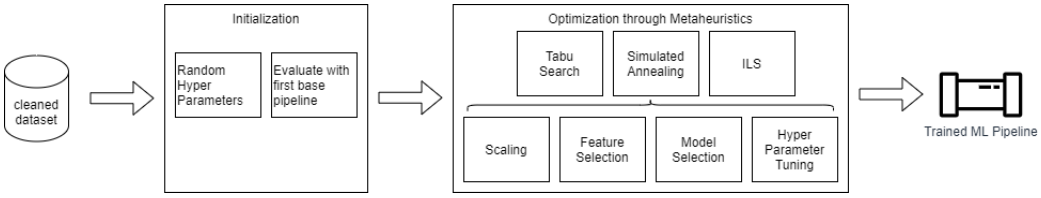

Figure 3.2: High-level system design diagram that highlights the framework components

3.3.1 Scaling

Scaling is a very important step in the machine learning pipeline. Most machine learning models work with numerical variables as input and output. Therefore, the data has to be scaled for this to make sense. For example, if one would want to predict the amount of money someone owns, based on their income, it is obvious that the value of 0.00000001 does not make sense. It is best to scale the input values in this case to something like a factor of 10,000. The scaling function takes any instance sequence and converts the data into the desired format.

Scaling methods are supported through the scikit-learn preprocessing module which scales the input data based on the method. Currently implemented are the `StandardScaler` and `MaxAbsScaler` methods. `StandardScaler` standardizes features by removing the mean and scaling to unit variance, while `MaxAbsScaler` scales each feature by its maximum absolute value.

This allows easy integration of other preprocessing steps if wanted. For evaluation purposes also a passthrough scaling method was included where no scaling is done just in case the data is already scaled or scaling does affect performance negatively.

3.3.2 Feature Selection

Another part of the preprocessing step is feature selection, being included in the automated preprocessing pipeline generation. Feature Selection is important because it reduces the number of parameters associated with the model and can therefore lead to a better fitting model. Currently is supported through the `SelectKBest` from the feature selection module of scikit-learn, which is a wrapper feature selection method and uses the test chi-squared statistic from the input data and ranks the feature by calculated importance. The range of features used is between 1 and the total amount of input variables. However, other algorithms could be easily integrated due to the high modularity of the framework.

3.3.3 Model Selection

Model selection is an advanced procedure that selects the best or the best-ranked candidate ML model from a set of candidate models. Usually, model selection is the responsibility of the practitioner, based on the data, domain knowledge and their expertise. There are various techniques available for choosing the right predictor from a pool of candidates.

In this case, the framework is cross-validation to figure out which models are viable. Cross-validation allows use to make a robust estimation of model performance with respect to predicting unseen data. The goal of cross-validation is to find a set of feasible sets where each set is given an equal share of training data.

The framework supports the classifiers listed in Table 3.3 for selection, their names are based on the implementation names in Scikit-Learn. The classifiers are taken from the Scikit-Learn package, and are implemented through the Scikit-Learn pipelines module and each classifier has a built-in cross-validation function which is used to evaluate the model performance based on the available data. Other classifiers can be added simply by adding extra classifiers to the pipeline.

Table 3.3: Classifiers and their respective Hyperparameters included in the Metaheuristics-Sklearn framework

Classifier	**Hyperparameters**
`LinearSVC`	`loss`, `tol`, `C`
`RandomForestClassifier`	`bootstrap`, `max_features`, `min_samples_leaf`, `min_samples_split`
`RidgeClassifier`	`alpha`
`SGDClassifier`	`loss`, `penalty`, `alpha`, `l1_ratio`, `tol`, `epsilon`, `learning_rate`, `eta0`, `power_t`, `average`
`BernoulliNB`	`alpha`, `fit_prior`
`KNeighborsClassifier`	`weights`, `n_neighbors`, `p`
`SVC`	`C`, `kernel`, `degree`, `gamma`, `coef0`, `shrinking`, `tol`, `max_iter`
`DecisionTreeClassifier`	`criterion`, `min_samples_split`, `min_samples_leaf`

3.3.4 Hyperparameter Tuning

One of the most difficult steps to automate is hyperparameter tuning. Hyperparameter tuning includes the part of optimizing values of certain model parameters for improving performance. Models can have a large number of hyperparameters and tuning these to get good fitting models can be a tricky task. There is also the possibility that for certain classes of problem, certain sets of hyperparameters are best suited. Even though there are various techniques available to tune hyperparameters, these techniques require a lot of expertise to make them work correctly.

The proposed framework takes care of this part of the machine learning pipeline for the user and automatically finds the best hyper parameters for each algorithm implemented

in the ML framework. This is done by evaluating models for max/min performance across different hyper parameter values through the metaheuristic optimization algorithms.

CHAPTER 4

AutoML through Metaheuristic Algorithms

The approach to solving AutoML problems using metaheuristic algorithms is set out in the following chapter. Section 4.1 will continue with a brief overview of metaheuristic algorithms and their components. Following, are the specific sections for each algorithm and their respective implementation specifics, as well as an explanation of the necessary AutoML problem-specific improvements.

4.1 Introduction

The original concept of metaheuristic algorithms, also known as stochastic approximation algorithms, was introduced in 1951 by Robbins and Monro [RM51]. These algorithms are a way to find a near optimal solution for the problem using lower computational cost than brute-force methods and without necessarily testing all possible solutions. This is done by mimicking natural evolution processes such as Darwinian evolution, genetic algorithms, swarm intelligence, and ant colony optimization (ACO) [Yan10].

Most metaheuristics use some form of "non-directive search", which means that the algorithm does not have any predefined search pattern or knowledge about the solution's properties when searching for an approximate result. Instead it will move around the search space and choose promising areas to explore further. Another term for this kind of algorithms is "heuristic" (from Greek: heurístikos = "dealing with discovery") [OL96].

Metaheuristic algorithms such as Genetic Algorithms (GA) use the concepts of reproduction, mutation, and survival to evolve their solutions. Modern evolutionary algorithms, even if they use GA as a sub-algorithm, are designed with more advanced features. For example: segmentation: dividing the problem into smaller parts; elitism: keeping parts

of the best individuals; and co-evolution: simultaneously evolving solutions for multiple sub-problems. [Yan10]

Metaheuristic algorithms [FD02, HSCS19, BLS13], have numerous applications, from designing new machines to optimize production processes to solving very general problems in engineering analysis and optimization. Furthermore, recent developments in metaheuristics [HSCS19] provide significant improvements in the solution accuracy for highly nonlinear problems when compared with other more classical approaches.

Metaheuristic algorithms are relatively easy to implement and can be used to solve many kinds of problems. The main advantage over algorithms being that on the expense of a little bit more computer time, faster convergence rates can be obtained or a greater variation of the solution search space may be searched. On the other hand, a disadvantage of metaheuristics can be that they don't guarantee optimal solutions or even near optimal solutions. Another serious problem is if the performance measure is not well-defined. Metaheuristic algorithms also need a good initialization method in order to start the local search phase. The results are often poor if this is done incorrectly.

There are many different metaheuristic algorithms in use today, and dozens of articles on the theory of each of them [HSCS19, FD02, BLS13]. Here, we are going to focus on Iterated Local Search, Simulated Annealing and Tabu Search and how they were implemented for solving AutoML problems. All of these are roughly based on the same concept but there are crucial differences, for example the way that the algorithm controls its own search process. Before going into detail for each of these algorithms, an overview of shared concepts between the algorithms will be given before continuing into further details.

All of the algorithms share these three major components:

- Initialization
- Search-space exploration
- Evaluation

4.2 Initialization

The initialization phase is the phase before the first step of the algorithm. It is used to generate the initial solution and initialize the initial values for important parameters such as mutation rates, exploration rates, or crossover probabilities.

The selection of the best values for the initial parameters is different in every algorithm. The simplest and most popular way to generate an initial solution is the random search, which is also used for the MetaheuristicSklearn framework. In the framework, for each hyperparameter of each classifier their values are picked randomly from a normal distribution.

4.3 Search-space Exploration

Metaheuristic algorithms often employ some kind of mechanism to avoid local minima and explore the search space more thoroughly. This is usually done through combinations of local search, neighborhood generation, move operators, and perturbation [OL96]. This is the heart of metaheuristics. The algorithm explores the search space by means of moves in order to discover new candidate solutions.

The search-space exploration phase is the most important phase of any algorithm. If the initial conditions are good, then the search process will yield better solutions than for a local optimum found in the initial phase. This is what defines metaheuristics: searching through a large space of all possible local optima, finding them and then improving them with an appropriate penalty system.

4.3.1 Local Search

In the Local Search phase, the algorithm searches around the current solution candidate in order to find more promising sub-problem sets. This can be done by a number of methods:

- hill climbing
- tabu search
- simulated annealing

In hill climbing one starts from the current solution and iterably improves it until it finds a local minima. If the hill is not too steep, then the local minimum is almost impossible to miss; once its location has been detected, it will be used as an approximation for other nearby points.

In random local search, one explores the problem space by randomly generated new points in a neighborhood of the best known solution so far. A solution (or a subset of solutions) is accepted based on the acceptance criterion. It is worth mentioning that this approach is very slow and may result in excessively high number of iterations [HS04, KGV83, AL97] making it an very inefficient approach.

Ideally, the best solution will be located in the first iteration. However, because of the randomness used in the selection method, this is an highly unlikely event. The algorithm will need several iterations before it finds a relatively good solution to the current problem instance. The selection method dictates the speed of convergence. If a poor selection method is used, the algorithm will go through many iterations without getting anywhere.

4.3.2 Move Operators

Move operators [HSCS19], can be used to improve the current solution or move towards a more promising sub-problem set. The logic behind a move operator is to change the value(s) of a problem's variables one at a time, with evaluation of the solution resulting from these changes.

For example, if the objective function is to minimize and there are 2 variables in play (v1, v2), then one would change the value of either variable to lower the objective function when compared to the previous variable values.

4.3.3 Neighbourhood Generation

In order to solve an optimization problem, we have to generate a neighborhood of solution. The neighborhood is a list of points where we are allowed to move.

The neighborhood generation process is an essential part of the metaheuristic search algorithm. The goal of the algorithm is to figure out good values for the new candidate solutions. There are multiple approaches to generating new high quality neighborhoods [LLG12]. One could try to change only one part of the solution configuration, or on the other hand like Tabu Search not allow a certain solution to be considering in the neighborhood for a fixed number of iterations.

4.3.4 Perturbation

Another form of search space exploration is the perturbation step in a search algorithm, which involves moving away a greater distance than a move operator or neighborhood generation method [BS91].

The goal here is for the search algorithm to land in a new part of the search space where it can begin a better search than it did previously. This is an excellent method for avoiding being trapped in a local optimum. This method is heavily used by the Iterated Local Search algorithms.

4.4 Evaluation

The evaluation phase is crucial, since this is where the algorithm decides whether a candidate solution should be discarded or not. This is usually done in terms of the objective function value. All metaheuristic algorithms rely on evaluation methods to make this decision. Evaluation methods can either be based on objective functions or some other way of judging a solution. The search process will stop when a candidate solution with an objective function value below some cutoff level has been found or if the maximum number of allowed iterations has passed without any improvements to the current best solution being found.

The most basic or common evaluation method is simply to compare the value of the new solution against that of the best known solution [AL97]. This is done to see if the new

value is better or not. Usually the best solution when found is used as a starting point for another search cycle. For example, in an iterated local search, if the best known solution is 12 and the new candidate solution is 13, then 13 is used as a starting point for the next iteration.

In order to make this decision a metaheuristic algorithm needs some way to compare the two solutions. The way how is to calculate or compare the new solution with the best known solution. There are different techniques to perform this comparison. The amount of change that is made to the solution in each iteration is crucial for how well the algorithm performs. The most common method for evaluating solutions is the fitness function, which returns a number representing how good a solution is. The larger the better for maximization, or the small the better for minimization problems.

The fitness function can be based on any kind of information. It can be the objective value or any other indicator. The most common fitness function is to return the quality of a solution by evaluating a function of the objective value.

4.4.1 Acceptance Criterion

Although it might seem intuitive to always choose the candidate with the highest returning fitness function value, this leads to getting caught in local minima in search problems with vast search spaces. Therefore acceptance criterions get introduced, which are rules that define when for an advantage in search space exploration, one also might accept solutions that perform worse than already found solutions before.

Simulated Annealing is one of many metaheuristic algorithms that use this component, in which it often tries to continue with a worse solution than the one already found in order to explore the search space early on, before later concentrating on optimizing the already found better solution.

4.5 Simulated Annealing

In this section, we'll go over a the metaheuristic algorithm Simulated Annealing, which is used to solve automated machine learning problems inside the MetaheuristicSklearn framework. We'll also discuss advantages and disadvantages of using this technique for solving these types of problems. A quick overview of simulated annealing as a problem-solving and optimization method, as well as the theory behind it, will be given. Following that, an analysis into how it can be used to solve automated machine learning problems. Finally, at the end of this section, general assumptions regarding simulated annealing and what makes it efficient or inefficient at solving automated machine learning problems are discussed.

Simulated Annealing [VLA87, KGV83, Rut89], is a problem-solving and optimization technique that has been used to solve very complex mathematical problems, optimize systems, and generalize search processes. It can be thought of as a process that early

favours search space exploration, while later on favours the optimization of the already founds solutions with only better ones.

The idea behind this approach is that early search space exploration is a more favorable action due to the high possibility of getting stuck in a local minima, where as the longer the search goes on the higher the probability that a solution near the global optimum has been found.

This approach also makes sense from a solving AutoML problem viewpoint. Where a lot of time in a practitioner work is spent finding the right classifier and preprocessing steps, while later on tuning his selected methods through hyper parameter tuning for optimal performance.

An overview on how Simulated Annealing was implemented for solving AutoML problems inside the MetaheuristicSklearn package can be seen in Figure 4.1, while on the other hand the respective parameters for tuning the algorithm are shown in Table 4.1.

Table 4.1: Parameters for the Simulated Annealing algorithm, descriptions, and their default values

Parameter	**Description**	**Default Value**
`T`	Starting Temperature	`100`
`T_min`	Ending Temperature	`1e-5`
`cooling_rate`	Cooling rate by which to lower the temperature in each iteration	`0.99`

As with every AutoML problem this one starts with an instance, consisting of data called X and a target variable Y. These along with some algorithm specific parameters are given to the object instance of MetaheuristicSklearn which then starts its work. The first thing the MetaheuristicSklearn instance does is to evaluate the fitness of the instance with a baseline pipeline as described in earlier sections, which can be thought of as its evaluation function value. This evaluation function value is a float value that represents the estimated performance of a certain pipeline for this dataset.

After that all of the hyperparameters for the containing classifiers are randomly initialized and the move operator is called. Inside the move operator a random pipeline step is chosen and its value is changed randomly. After which the new pipeline configuration is evaluated and its score is saved into the new_score variable. Having evaluated the new pipeline its performance is compared with the previous one and if it is better performing it is chosen as the new configuration.

The next step is the acceptance probability, and can be considered as the most important step in the Simulated Annealing algorithm, also giving it its name. The probability of transitioning from the current configuration to the new pipeline configuration depends

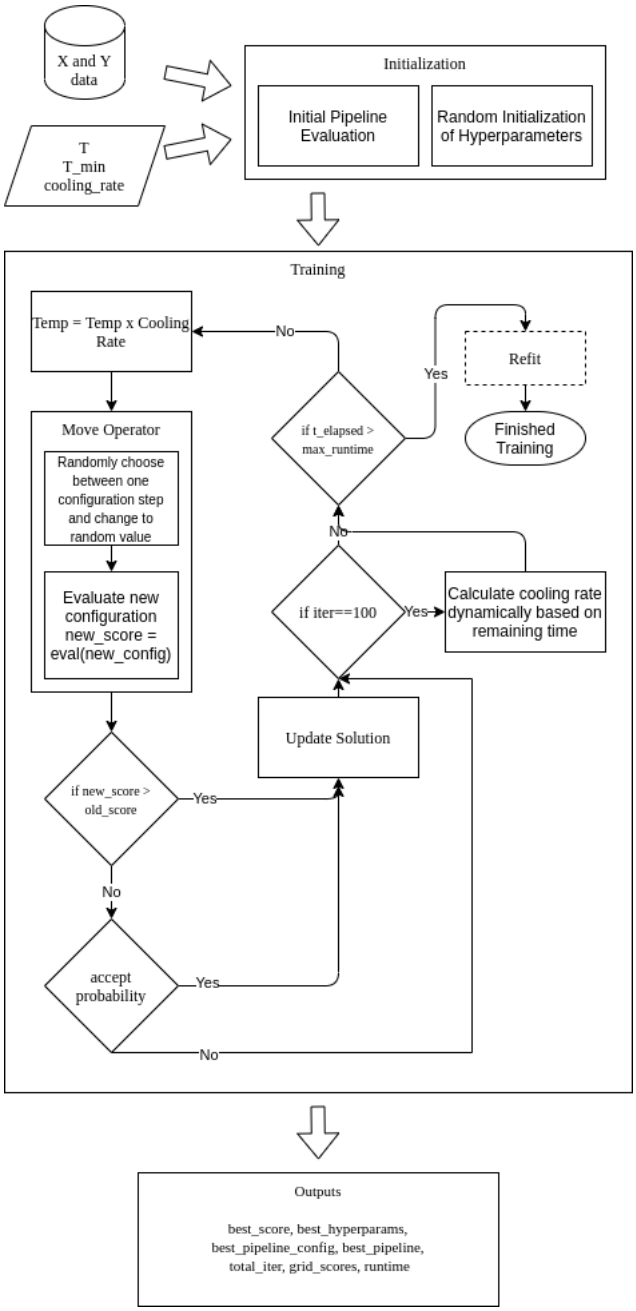

Figure 4.1: A flowchart diagram of the Simulated Annealing implementation inside MetaheuristicSklearn for solving AutoML problems.

on an appropriate probability function dependent on the temperature and the quality of the two configurations. The exact formula is listed below:

$$random(0,1] < e^{\frac{eval(conf_c)-eval(conf_n)}{T}} \tag{4.1}$$

From Formula 4.1, for maximization problems, we can observe that the probability of accepting a worse solution when the temperature is high is higher. On the other hand, when the temperature falls the probability also goes down. With a temperature of zero, the method limits itself to the greedy hill climbing algorithm, which just transitions uphill.

After that, the temperature is decreased through the cooling rate and the algorithm iteratively repeats the same steps in the next iterations. To make use of the full running time specified through max_runtime, a dynamic calculation of the cooling rate was implemented into the algorithm. After a 100 iterations the algorithms makes use of all previous iterations and their running times and calculates the optimal cooling rate so that the temperature is able to go down all the way to zero at the end of the running time. This was done with idea that the algorithm should not get stuck on a high temperature, while being able to hill climb in later stages of the search for the most optimal performance.

After having finished the search the best performing pipeline so far is refitted on the whole data and the AutoML search is done.

4.6 Tabu Search

In this section we will go over the Tabu Search metaheuristic algorithm which is employed inside the MetaheuristicSklearn framework, for solving automatic machine learning problems. We will also address the benefits and drawbacks of using this method to solve certain problems.

Tabu Search [Glo90, Glo95], is a metaheuristic search algorithm used in combinatorial optimization. Because it works by iteratively improving solutions that are already close to the local optimum, but also accepting moves that are further away, it's often viewed as a special case of the Simulated Annealing (SA) method. It belongs to the family of stochastic search algorithms and uses what is called a "tabu list." Using this tabu list, forbidden areas or states are created and certain unsuccessful moves which have already been performed are eliminated from consideration for a given number of iterations (epochs). This way the exploration phase (of the search) is reduced as iterations go by, and thus certain target solutions can be found more quickly. In the context of MetaheuristicSklearn it's employed for solving Machine Learning related problems.

The algorithm starts with an initial solution to a problem and an empty tabu list. Then all neighbourhood solutions of the current solution are generated. The best solution in the neighbourhood is found. If the solution is not tabu, the new solution is selected as the current solution, the tabu list is updated, and we continue with the next iteration.

Else, we check for an aspiration criteria, if it is fulfilled we update the current solution and update the tabu list. Finally, if all of the previous checks were false, we find the best not tabu solution in the neighbourhood select it as the current solution and update the tabu list.

The whole process is repeated until the terminate condition is fulfilled. Figure 4.2 provides an outline of how an example AutoML problem is solved by Tabu Search inside the MetaheuristicSklearn program, while Table 4.2 shows the corresponding algorithm parameters.

Table 4.2: Parameters for the Tabu Search algorithm, descriptions, and their default values

Parameter	**Description**	**Default Value**
`tabu_size`	Size of the tabu list	`10`
`probabilities`	Defining the probability distribution for when generating a new neighbourhood for for steps ["scaling", "feature_selection", "est", "hyperparams"]	`[0.25, 0.25, 0.25, 0.25]`

As explained at the beginning of Section 4.6, Tabu Search begins the search process in the same way as the Simulated Annealing algorithm. When the parameters and data are obtained from the user, a baseline pipeline configuration is evaluated and the starting point for subsequent steps within the algorithm is defined.

Having defined the starting point of the algorithm, the training phase of the algorithm start. At the the beginning of each iteration, the neighborhood of the current solution is generated. The neighborhood generation method consist of multiple steps. Initially a random pipeline step is chosen with which the neighborhood is generated, for example if the random step is the scaling step, the current pipeline is evaluated with all possible scaling methods present and then ranked by performance to find the best solution inside the neighborhood. This randomness between chosing different pipeline steps can be configured manually through the probabilities parameter, which defines the probability distribution for new neighborhood generation.

If the best solution from the newly generated neighbourhood is not Tabu, it is compared with the current best solution found so far, if it performs better the best solution is updated else, it will be accepted as a new current solution and we continue with the next iteration. On the other hand, if this solution is Tabu, but it fulfills the aspiration criteria of our algorithm, in this case the the current solution is better than any other solution we found so far, we circumvent the Tabu list, accept the solution, and update the best solution anyhow considering it is the best performing solution so far and continue with the next iteration.

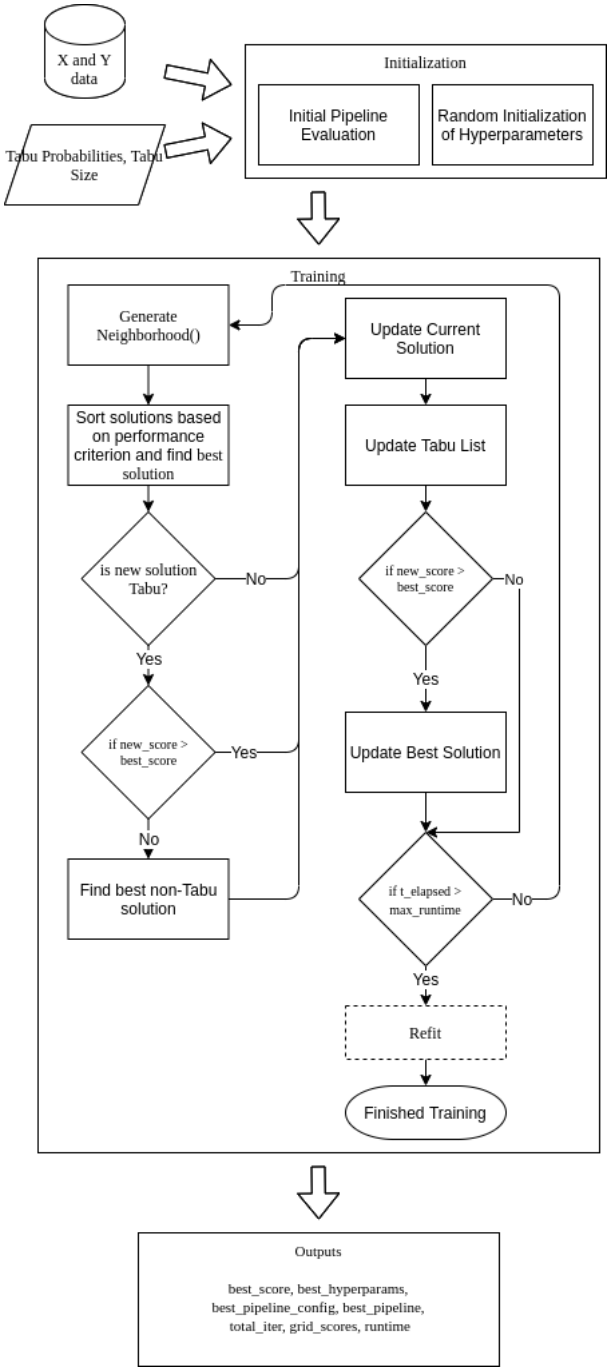

Figure 4.2: A flowchart diagram of the Tabu Search implementation inside MetaheuristicSklearn for solving AutoML problems.

Else, if the best performing solution in the newly generated neighborhood is Tabu and not better than the current best performing solution, find the best performing non-Tabu solution, select that solution as the starting point for the next iteration, and update the Tabu list. Inside the Tabu list are stored pipeline step operators. Such an operator for example would be for the scaling step the MinMaxScaling method, which once chosen would be put into the Tabu list and could not be chosen for evaluation until it get outs of the Tabu list.

The Tabu list size is controlled by the tabu_size parameter, the default parameter value is 10, once the Tabu list is full with operators, the first added operator gets kicked out and the new one is inserted at the end of the queue in a First In First Out (FIFO) type of queue.

Once the algorithm spends its running time, the best performing pipeline is then refitted on the whole training data and exposed to the user with various other outputs such as best hyperparameters, grid scores, best pipeline configuration, etc.

4.7 Iterated Local Search (ILS)

This section will give an introduction to the Iterated Local Search (ILS) algorithm, discuss its origin and approach, and present a complete analysis of how it is applied to solve AutoML problems in the MetaheuristicSklearn Implementation.

Iterated Local Search (ILS) [RLMS00, BS91, LMS03], is one of the most successful Metaheuristic approaches, possibly because it is able to apply an evaluation function that can take into account the previous results of the algorithm. The three main components of ILS are:

- Local Search
- Perturbation
- Acceptance Criterion

In ILS usually a local search based processor tries to improve the solution for a number of iterations. To prevent from getting stuck in local optima, after a certain number of local search iterations the solution is perturbated to move to a new space in the search space. After each perturbation and local search, the acceptance criterion compares solutions, and handles the direction of the search space in the right direction.

An overview on how ILS was implemented for solving AutoML problems inside the MetaheuristicSklearn framework can be seen in Figure 4.3, while on the other hand the respective parameters for tuning the algorithm are shown in Table 4.3.

As the previous algorithms, ILS starts it search process in the same way. After having received the parameters and training data from the user, inside the instance a baseline

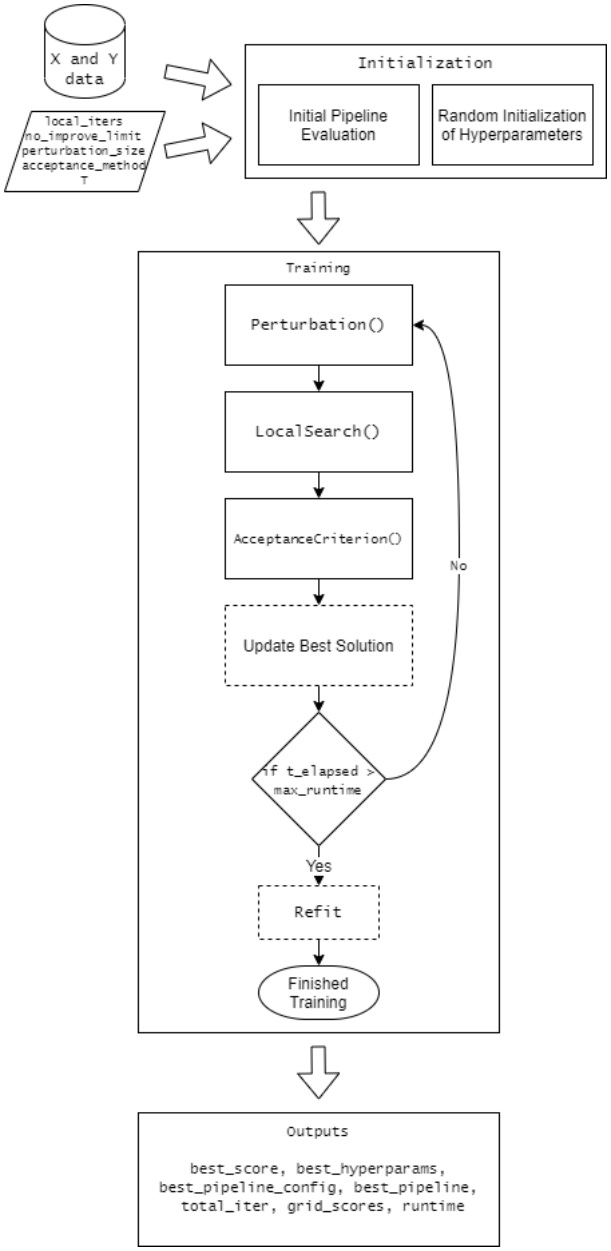

Figure 4.3: A flowchart diagram of the Iterated Local Search implementation inside MetaheuristicSklearn for solving AutoML problems.

Table 4.3: Parameters for the Iterated Local Search algorithm, descriptions, and their default values

Parameter	Description	Default Value
`local_iters`	Number of iterations to do inside LocalSearch	`100`
`no_improve_limit`	Limit of after how many iterations to stop without any improvement	`25`
`perturbation_size`	Percentage of hyperparams to replace in best performing model	`0.25`
`acceptance_method`	Method to choose the acceptance_criterion. Choices: ["default", "annealing", "always"]	`"default"`
`T`	Temperature for annealing acceptance_method	`5`

pipeline configuration is evaluated and serves as the starting reference point for future steps inside the algorithm. That solution then is perturbated, a perturbation is a configuration move which serves as a way to get itself unstuck from local minima. A perturbation inside an algorithm should not be easily undone by the local search component. It is very important to fine tune this step as a too strong perturbation can lead to a random restart of the search in a whole new neighborhood, while a too weak perturbation can lead to keep being stuck in the same local minima.

The perturbation method in the implementation of MetaheuristicSklearn includes a perturbation_size parameter given to the instance before training. This parameter controls the size of the perturbation on the best performing configuration and depending on the size of it, the amount of changed hyperparameters changes. A whole perturbation consists of a random change of either scaling method or number of features selected, while also changed perturbation_size parameters.

After perturbation, the new resulting configuration is put through a local search where the algorithm tries to search new possible combinations while altering the classifier and hyperparameter steps inside the configuration. This amount of local search iterations is given through the local_iters parameter, which controls the amount of iterations. To prevent from wasting too much time on a wrongly perturbated solution inside the local search it is kept track for how many iterations there was not any improvement. After no_improve_limit iterations without improvements in performance the local search is stopped. This number can be controlled through the no_improve_limit parameter of

the algorithm.

Having finished the local search the acceptance criterion method is called which can have three various modes:

- default: this compares the score of the best solution found inside the local search with the previous configuration and in case the new pipeline is better performing it is chosen as the next solution. Which is just a greedy hill climbing alternative.
- simulated annealing: This acceptance method makes use of the acceptance probability method used inside the simulated annealing algorithm, which supports selecting worse performing configurations early on for early search space exploration and avoidance of local minima. A detailed description of the simulated annealing acceptance probability can be found in section 4.5.
- always: This acceptance method always accepts the solution found after having local searched the new perturbated configuration independent of performance.

Depending on the acceptance method the best solution gets updated, it is checked if there is remaining search time for the algorithm and a new iteration is started if there is, else the the best performing pipeline is refitted on the whole training data and then exported with other various outputs to the user. This is where the algorithm finishes the AutoML search.

CHAPTER 5

Evaluation

This chapter describes the experimental environment used to test the proposed AutoML framework for supervised machine learning classification problems, as well as details on the evaluation of the framework in comparison to other state-of-the-art frameworks.

Section 5.1 provides a comprehensive presentation of the experimental settings and the problem benchmarks chosen. Moreover, in Section 5.2, we go through the experiments that were carried out using the proposed framework and other state-of-the-art frameworks using the benchmarks, while reporting findings about the framework and evaluations.

5.1 Experiment Settings

Note that MetaheuristicSklearn's implementation is purely a single thread and only one core is used during the calculation process. Thus all other implementations have been assessed on the same single-thread basis but some architectures have support with multiple threads. MetaheuristicSklearn's memory use depends heavily on the training data given at the beginning of the training.

All experiments were executed on a Linux machine with an AMD Opteron(tm) Processor 6308 CPU single-threaded at 1.4 - 3.5 GHz, using 189GB of memory. Algorithm were evaluated on both the training benchmark which was also used for tuning the algorithm parameters inside MetaheuristicSklearn and an unseen test benchmark providing an equal performance evaluation for all frameworks.

All of the experiments that are mentioned in this chapter were conducted using the two benchmark suites taken from OpenML [VvRBT13, MF]. For training and metaheuristic parameter optimization purposes the OpenML-CC18 benchmark suite was chosen which according to the authors contains all OpenML datasets from mid-2018 that satisfy a large set of clear requirements for thorough yet practical benchmarking. It includes

datasets frequently used in benchmarks published over the last years, so it can be used as a drop-in replacement for many benchmarking setups.

The suite is defined as the set of all verified OpenML datasets that satisfy the following requirements [BCF+19]:

- the number of observations are between 500 and 100000 to focus on medium-sized datasets, that are not too small and not too big,
- the number of features does not exceed 5000 features to keep the runtime of algorithms low,
- the target attribute has at least two classes
- the ratio of the minority class and the majority class is above 0.05, to eliminate highly imbalanced datasets which require special treatment for both algorithms and evaluation measures.

While datasets that were excluded [BCF+19]:

- are artificially generated (not to confuse with simulated)
- cannot be randomized via a 10-fold cross-validation due to grouped samples or because they are time series or data streams
- are a subset of a larger dataset
- have classes with less than 20 observations
- have no source or reference available
- can be perfectly classified by a single attribute or a decision stump
- allow a decision tree to achieve 100% accuracy on a 10-fold cross-validation task
- have more than 5000 features after one-hot-encoding categorical features
- are created by binarization of regression tasks or multiclass classification tasks, or
- are sparse data (e.g., text mining data sets)

On the other hand to not favorize the metaheuristic algorithms which parameters were optimized on the training benchmark, which will be explained in more detail later, a testing benchmark was selected as well. The outcomes of the two benchmark suites were compared over each other for both machine learning tasks and the testing benchmark. The chosen testing benchmark is called AutoML Benchmark More Classification which is a new benchmark that gathers the most relevant datasets that were missing from

the OpenML CC-18 benchmark and was also used for evaluation purposes inside the OpenML AutoML Benchmark [GLP+19].

The two major classification benchmark collections were chosen because of their wide coverage and variety of supervised machine learning problems, and having various different properties considering number of Instances, Features, Classes, Missing values, Instances with missing values, Numerical features and Symbolic features. More details about those can be found in Table 5.1 and Table 5.2, where all of the properties for each dataset inside each benchmark collection are listed.

Table 5.1: List of train benchmark datasets and their properties

Dataset Name	Instances	Features	Classes	NanValues	InstancesWithNanValues	BinaryFeatures	NumericFeatures	SymbolicFeatures
Bioresponse	3751	1777	2	0	0	1	1776	1
CIFAR_10	60000	3073	10	0	0	0	3072	1
Devnagari-Script	92000	1025	46	0	0	0	1024	1
Fashion-MNIST	70000	785	10	0	0	0	784	1
GesturePhaseSegmentation-Processed	9873	33	5	0	0	0	32	1
Internet-Advertisements	3279	1559	2	0	0	1556	3	1556
MiceProtein	1080	82	8	1396	528	3	77	5
PhishingWebsites	11055	31	2	0	0	23	0	31
adult	48842	15	2	6465	3620	2	6	9
analcatdata_authorship	841	71	4	0	0	0	70	1
analcatdata_dmft	797	5	6	0	0	1	0	5
balance-scale	625	5	3	0	0	0	4	1
bank-marketing	45211	17	2	0	0	4	7	10
banknote-authentication	1372	5	2	0	0	1	4	1
blood-transfusion-service-center	748	5	2	0	0	1	4	1

Table 5.1: List of train benchmark datasets and their properties - continued

breast-w	699	10	2	16	16	1	9	1
car	1728	7	4	0	0	0	0	7
churn	5000	21	2	0	0	3	16	5
climate-model-simulation-crashes	540	21	2	0	0	1	20	1
cmc	1473	10	3	0	0	3	2	8
cnae-9	1080	857	9	0	0	0	856	1
connect-4	67557	43	3	0	0	0	0	43
credit-approval	690	16	2	67	37	5	6	10
credit-g	1000	21	2	0	0	3	7	14
cylinder-bands	540	40	2	999	263	4	18	22
diabetes	768	9	2	0	0	1	8	1
dna	3186	181	3	0	0	180	0	181
dresses-sales	500	13	2	835	401	1	1	12
electricity	45312	9	2	0	0	1	7	2
eucalyptus	736	20	5	448	95	0	14	6
first-order-theorem-proving	6118	52	6	0	0	0	51	1
har	10299	562	6	0	0	0	561	1
ilpd	583	11	2	0	0	2	9	2
isolet	7797	618	26	0	0	0	617	1
jm1	10885	22	2	25	5	1	21	1
jungle_chess_2pcs-_raw_endgame_complete	44819	7	3	0	0	0	6	1
kc1	2109	22	2	0	0	1	21	1
kc2	522	22	2	0	0	1	21	1
kr-vs-kp	3196	37	2	0	0	35	0	37
letter	20000	17	26	0	0	0	16	1
madelon	2600	501	2	0	0	1	500	1
mfeat-factors	2000	217	10	0	0	0	216	1
mfeat-fourier	2000	77	10	0	0	0	76	1
mfeat-karhunen	2000	65	10	0	0	0	64	1
mfeat-morphological	2000	7	10	0	0	0	6	1
mfeat-pixel	2000	241	10	0	0	0	240	1
mfeat-zernike	2000	48	10	0	0	0	47	1

Table 5.1: List of train benchmark datasets and their properties - continued

nomao	34465	119	2	0	0	3	89	30
numerai28.6	96320	22	2	0	0	1	21	1
optdigits	5620	65	10	0	0	0	64	1
ozone-level-8hr	2534	73	2	0	0	1	72	1
pc1	1109	22	2	0	0	1	21	1
pc3	1563	38	2	0	0	1	37	1
pc4	1458	38	2	0	0	1	37	1
pendigits	10992	17	10	0	0	0	16	1
phoneme	5404	6	2	0	0	1	5	1
qsar-biodeg	1055	42	2	0	0	1	41	1
satimage	6430	37	6	0	0	0	36	1
segment	2310	20	7	0	0	0	19	1
semeion	1593	257	10	0	0	0	256	1
spambase	4601	58	2	0	0	1	57	1
splice	3190	61	3	0	0	0	0	61
steel-plates-fault	1941	28	7	0	0	0	27	1
texture	5500	41	11	0	0	0	40	1
tic-tac-toe	958	10	2	0	0	1	0	10
vehicle	846	19	4	0	0	0	18	1
vowel	990	13	11	0	0	1	10	3
wall-robot-navigation	5456	25	4	0	0	0	24	1
wdbc	569	31	2	0	0	1	30	1
wilt	4839	6	2	0	0	1	5	1

Table 5.2: List of test benchmark datasets and their properties

Dataset Name	Instances	Features	Classes	NanValues	InstancesWithNanValues	BinaryFeatures	NumericFeatures	SymbolicFeatures
Bioresponse	3751	1777	2	0	0	1	1776	1
Click_prediction_small	39948	12	2	0	0	1	5	7
Diabetes130US	101766	50	3	0	0	9	13	37
GesturePhaseSegmentation-Processed	9873	33	5	0	0	0	32	1
Higgs	1000000	29	2	0	0	1	28	1
Internet-Advertisements	3279	1559	2	0	0	1556	3	1556
KDDCup09-Upselling	50000	15001	2	25108569	50000	1	14811	1
KDDCup99	4898431	42	23	0	0	5	32	10
PhishingWebsites	11055	31	2	0	0	23	0	31
Satellite	5100	37	2	0	0	1	36	1
ada	4147	49	2	0	0	1	48	1
amazon-commerce-reviews	1500	10001	50	0	0	0	10000	1
arcene	100	10001	2	0	0	1	10000	1
churn	5000	21	2	0	0	3	16	5
cmc	1473	10	3	0	0	3	2	8
dna	3186	181	3	0	0	180	0	181
eucalyptus	736	20	5	448	95	0	14	6
first-order-theorem-proving	6118	52	6	0	0	0	51	1
gina	3153	971	2	0	0	1	970	1
kick	72983	33	2	149271	69709	4	14	19
madeline	3140	260	2	0	0	1	259	1
micro-mass	571	1301	20	0	0	0	1300	1

Table 5.2: List of test benchmark datasets and their properties - continued

okcupid-stem	50789	20	3	154107	48622	1	2	18
ozone-level-8hr	2534	73	2	0	0	1	72	1
pc4	1458	38	2	0	0	1	37	1
philippine	5832	309	2	0	0	1	308	1
porto-seguro	595212	58	2	846458	470281	24	26	32
qsar-biodeg	1055	42	2	0	0	1	41	1
sf-police-incidents	2215023	9	2	0	0	1	3	6
steel-plates-fault	1941	28	7	0	0	0	27	1
wilt	4839	6	2	0	0	1	5	1
wine-quality-white	4898	12	7	0	0	0	11	1
yeast	1484	9	10	0	0	0	8	1

We shall go through all AutoML Systems one more time and list their advantages and disadvantages below. Each framework has some features which are also described below. An overview of the functionality of all AutoML systems can be found in Table 5.3.

TPOT is one of the most mature and modern AutoML System offering the widest range of functionality including neural networks and Dask for parallel training. On the other hand, one disadvanteg of TPOT is that it can not handle missing data. Therefore, the handling of missing values always requires human intervention before continuing to solve your TPOT problem. Therefore, always adding an extra step to your pipeline when for example compared to Auto-Sklearn and Google AutoML Tables.

Auto-WEKA might scare some of the younger machine learning practitioners, due to the outdated looking Java GUI, Auto-WEKA is perfectly packaged in the WEKA application. It is worth noting that it requires a Java version smaller than 11. With the applications graphical user interface, it is easy to supply .csv dataset and configure the system and start searching for solutions, and even the machine learning beginner can get going and test various algorithms to see which is better suited to their approach.

Auto-Sklearn is very similar to Auto-Weka, which can be seen in Figure 2.4 where the ML Framework is almost the same, except perhaps in the diversity of the different classifiers it uses. Auto-Weka uses Ensemble methods and meta-learning, which Auto-Sklearn removed and only added Ensemble Step at the end of the training through Ensemble Stacking. Except for those two new additions and the removal of the meta methods, Auto-Sklearn is every other way analogous to Auto-WEKA. The diversity is reduced, mostly due to the great diversity of configurations in scikit-learn machine learning algorithm, therefore reducing the search space consists to 110 parameters.

Hyperopt-Sklearn at first glance, tends to be an automatic machine learning system, given the higher configuration and flexibility factor, it is more like a specialized hyper-

Table 5.3: Comparison of evaluated AutoML Systems

AutoML Systems	Functionality										
	Scikit Back-end	Preprocessing	Parallel Training	Meta Learning	Ensemble Learning	Neural Networks	Windows OS	MacOS	Linux	Set Training Time	Handles Missing Values
TPOT	✓	✓	✓			✓	✓	✓	✓	✓	
Auto-Weka			✓		✓		✓	✓	✓	✓	
Auto-Sklearn	✓	✓	✓	✓	✓			✓	✓	✓	✓
Hyperopt-Sklearn	✓	✓					✓	✓	✓		
AutoML Tables		✓	✓		✓	✓	✓	✓	✓	✓	✓
MetaheuristicSklearn	✓	✓					✓	✓	✓	✓	

parameter optimization tool for an advanced machine learning practitioner who knows what they're is looking for. Where as, compared to the previous automated machine learning systems, where you can see the best outcomes in the dataset by just providing the dataset.

Google AutoML Tables, although very guided in the whole automated machine learning process, leaves still room for further specific configurations in the AutoML Tables pipeline, for the more experienced machine learning practitioners. The top-level parameters of the pipeline leave a lot to be desired with regard to the specification of the actual search algorithm.

For evaluating said benchmarks a train-test split was chosen for each dataset with also a preprocessing step included, an overview of this experiment setup can be viewed in Figure 5.1. The preprocessing steps include an Imputer for missing values and One Hot Encoder for categorical columns, although some of the AutoML frameworks support the handling of missing values and feature encoding as showed in Table 5.4, with the idea to setup an even starting ground for all frameworks it was decided that all of the datasets receive the same treatment.

Having finished being preprocessed datasets are split into stratified train and test splits with a 0.75 and 0.25 percentages respectably, where the train split serves as training data for the AutoML frameworks and the test split for evaluation. It also worth mentioning that if possible after training the frameworks were instructed to refit the best performing model on the whole train split after finishing training if the allowed by the framework.

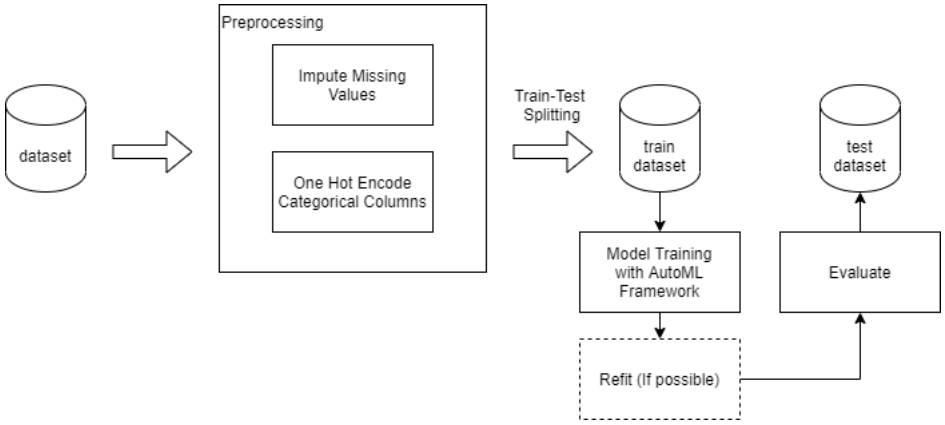

Figure 5.1: An overview of total experiment settings for the AutoML framework evaluation

All algorithms have been trained with the default setting parameters and evaluated on the F1 Score where possible, except for Google AutoML Tables and Auto-Weka due to either the missing option for that metric or the missing function for that metric. The complete configuration of each AutoML system can be found in Table 5.4.

After refitting on the training data, the resulting model is then evaluated on the test split and the mean of the three different seeded runs is reported as the final score. This is due to the the randomness included in some of the heuristics and therefore to account for any results which could be outliers due to a favored lucky outcome.

Furthermore, as part of the training the implemented metaheuristics algorithms and their parameters were optimized on the training benchmark using SMAC. SMAC is a configuration tool for algorithms to optimize arbitrary algorithm parameters in an set of instances [HHLB11]. All of the algorithms were trained on the training benchmark for 100 hours each, with a running time for each dataset of 1 hour. SMAC could choose between datasets randomly and find a global optimal value for each parameter. SMAC chose the best combination of parameters for each algorithm. The values chosen, their respective parameter spaces and default values can be found in Table 5.5.

5.2 Results

Using the above experimental settings it was possible to evaluate the analyzed set of state-of-the-art AutoML frameworks, as well as the metaheuristics algorithms inside the proposed AutoML framework between each other.

Before anything else, the metaheuristics algorithms implemented inside the proposed framework will be evaluated and compared between each other. After which, we will evaluate our results with state-of-the-art results from the set of AutoML frameworks analyzed.

Table 5.4: Experiment AutoML Frameworks and Configurations

AutoML System	Version	Configuration
TPOT	0.11.5	TPOTClassifier(verbosity=2, random_state=42, scoring="f1_weighted", cv=10, n_jobs=1, max_time_mins=60)
Auto-Weka	2.6.3 (Weka 3.8.3)	(batchSize=100, debug=False, doNotCheckCapabilities=False, memLimit=4086, metric=errorRate, nBestConfigs=1, numDecimalPlaces=2, parallelRuns=1, seed=123, timeLimit=60)
Auto-Sklearn	0.8.0	AutoSklearnClassifier(n_jobs=1, time_left_for_this_task=3600, resampling_strategy="cv", resampling_strategy_arguments="folds": 10, metric=autosklearn.metrics.f1_weighted, seed=42)
Hyperopt-Sklearn	0.3.0	HyperoptEstimator(trial_timeout=120, classifier=any_classifier("my_clf"), preprocessing=any_preprocessing("my_pre"), algo=tpe.suggest, max_evals=30, seed=42, loss_fn=f1_score)
AutoML Tables	Beta	Custom Train/Test Split and 1 Hour Train Time

5.2.1 In-between comparison of metaheuristic algorithms for solving AutoML problems

This section evaluates the implemented metaheuristic algorithms inside the proposed framework individually and compares them between each other. Since the problem datasets should be difficult to solve, another benchmark collection of datasets was used as a benchmark, to not favour the optimized parameter variants of our algorithms which have been trained on the train benchmark collection of datasets as part of the optimization process.

In order to evaluate each individual metaheuristic algorithm inside our proposed framework all the datasets inside the test collection have been used to validate their success rates and efficiency. Two performance metrics were used, the F1-Macro and Precision scores of the resulting best model from the framework after one hour of training. A comparison matrix was created for each performance metric, which contains the performance values of each algorithm per metric. This comparison matrix will be analyzed in order to evaluate how much improvement does each algorithm bring over the other, and if the

Table 5.5: Metaheuristic parameters, parameter ranges for SMAC and optimized values.

Metaheuristic	Parameter	Min	Max	Default	Optimized
Tabu	tabu size	1	1000	10	563
Tabu	scaling probability strength	0	1	0.25	0.1875
Tabu	feature selection probability strength	0	1	0.25	0.3125
Tabu	est probability strength	0	1	0.25	0.3125
Tabu	hyperparams probability strength	0	1	0.25	0.6875
ILS	local iters	50	10000	100	5025
ILS	no improve limit	10	50	25	30
ILS	perturbation size	0	1	0.25	0.5
ILS	acceptance method	default	annealing or always	default	annealing
SA	Temperature	10	10000	100	2507

algorithm parameter optimization of the proposed framework brought any measurable improvements over the default selected parameters.

Moreover, the following graph in Figure 5.2 illustrates the comparison of the evaluated algorithms. Each algorithm is indicated with its own color, where the lighter shade of the color is the not optimized variant and the darker one is the optimized value. While the best performing algorithm of each dataset is highlighted in bold in Table 5.6. In addition, the precision of the frameworks algorithms can be viewed in Table 5.7.

From Figure 5.2 we can clearly observe that there is a obvious gain from the optimization process for the algorithm parameters in performance, while there sometimes seem to be some outliers where the non optimized variant performed better than its optimized counterpart which can be addressed to the wide range of differences between datasets. The overall gained improvement in F1-Score on average is 7%.

From Table 5.6 we can observe that most of the times the better performing algorithms

are the Tabu Search and ILS variants. Where Tabu Search is the best performing algorithm 15 out of 31 times and ILS 13 out of 31 times, leaving Simulated Annealing winning only on 3 of total 31 instances.

In Table 5.7 we can observe the precision of each algorithm separately, where the error values are normalized with respect to the total error of the overall best model obtained by our framework in its first hour of training. The precision values range between 1 and 0, where 0 represents an worst-case scenario model with only errors and 1 represents a perfect model without any errors. From Table 5.7 we can clearly observe that there is a significant increase in precision in almost all optimized algorithms gaining 3% in precision overall.

Table 5.6: Metaheuristic Only F1-Score Performance Evaluation on Testing Benchmark

Dataset	SA Not Optimized	SA Optimized	Tabu Not Optimized	Tabu Optimized	ILS Not Optimized	ILS Optimized
Bioresponse	0.64	0.71	0.66	0.71	0.68	**0.73**
Click_prediction_small	0.54	0.54	0.54	**0.56**	0.48	0.52
Diabetes130US	0.14	0.19	0.15	**0.21**	0.19	0.19
GesturePhaseSegmentation-Processed	0.23	0.30	0.22	**0.32**	0.25	0.32
Higgs	0.41	**0.42**	0.41	0.41	0.37	0.41
Internet-Advertisements	0.92	0.93	0.93	0.92	**0.96**	0.94
KDDCup99	0.29	0.36	0.30	0.34	0.30	**0.38**
PhishingWebsites	0.90	0.94	0.89	**0.96**	0.93	0.95
Satellite	**0.87**	0.84	0.84	0.83	0.76	0.82
ada	0.55	0.62	0.56	0.63	0.76	**0.77**
amazon-commerce-reviews	0.82	0.82	0.80	0.79	0.77	**0.83**
arcene	0.74	0.80	0.75	0.78	**0.93**	0.92
churn	0.50	0.54	0.49	**0.55**	0.54	0.54
cmc	0.43	0.46	0.44	**0.48**	0.42	0.46
dna	0.85	0.91	0.86	**0.94**	0.90	0.92
eucalyptus	0.46	0.44	**0.46**	0.46	0.38	0.42

Table 5.6: Metaheuristic Only F1-Score Performance Evaluation on Testing Benchmark - continued

first-order-theorem-proving	0.27	0.32	0.27	0.31	0.31	**0.34**
gina	0.91	0.97	0.89	**0.99**	0.93	0.95
kick	0.41	0.48	0.39	0.48	0.42	**0.49**
madeline	0.48	0.53	0.49	0.53	0.86	**0.87**
micro-mass	0.74	0.80	0.74	0.82	0.81	**0.82**
okcupid-stem	0.42	0.43	**0.45**	0.41	0.44	0.41
ozone-level-8hr	0.61	0.66	0.62	0.65	**0.71**	0.70
pc4	0.68	0.75	0.67	**0.76**	0.72	0.75
philippine	0.74	0.72	0.72	0.71	**0.79**	0.77
porto-seguro	0.50	0.49	**0.50**	0.50	0.46	0.49
qsar-biodeg	0.86	0.84	0.86	0.82	**0.88**	0.85
steel-plates-fault	0.76	**0.79**	0.76	0.77	0.68	0.66
wilt	0.93	0.94	**0.95**	0.95	0.90	0.92
wine-quality-white	0.07	0.07	0.08	**0.09**	0.05	0.05
yeast	0.55	0.53	**0.56**	0.54	0.47	0.51

Table 5.7: Metaheuristic Only Precision Performance Evaluation on Testing Benchmark

Dataset	**SA Not Optimized**	**SA Optimized**	**Tabu Not Optimized**	**Tabu Optimized**	**ILS Not Optimized**	**ILS Optimized**
Bioresponse	**0.71**	0.71	0.70	0.70	0.68	0.71
Click_prediction_small	0.51	**0.56**	0.52	0.55	0.51	0.56
Diabetes130US	0.43	0.44	0.43	0.42	**0.48**	0.46
GesturePhaseSegmentation-Processed	0.45	0.46	0.47	0.47	**0.50**	0.47
Higgs	0.63	0.67	0.63	0.65	**0.67**	0.65
Internet-Advertisements	0.91	0.97	0.93	0.95	**0.98**	0.97
KDDCup99	0.43	0.44	0.41	**0.45**	0.36	0.43

Table 5.7: Metaheuristic Only Precision Performance Evaluation on Testing Benchmark - continued

PhishingWebsites	0.92	0.94	0.93	**0.95**	0.94	0.93
Satellite	0.90	0.92	0.90	0.90	0.93	**0.94**
ada	0.73	0.75	0.73	0.73	**0.81**	0.79
amazon-commerce-reviews	0.84	0.83	0.83	0.82	**0.86**	0.85
arcene	0.82	0.81	0.81	0.82	0.88	**0.92**
churn	0.76	0.80	0.77	0.81	**0.82**	0.81
cmc	0.47	0.47	0.48	**0.49**	0.49	0.48
dna	0.84	**0.92**	0.83	0.91	0.87	0.91
eucalyptus	**0.47**	0.46	0.47	0.46	0.46	0.47
first-order-theorem-proving	0.36	0.41	0.37	0.40	0.39	**0.43**
gina	0.99	0.97	**1.00**	0.95	0.95	0.95
kick	**0.62**	0.59	0.59	0.62	0.55	0.59
madeline	0.51	0.56	0.53	0.56	0.81	**0.87**
micro-mass	0.74	0.81	0.74	0.82	0.79	**0.82**
okcupid-stem	0.50	0.48	**0.50**	0.46	0.50	0.49
ozone-level-8hr	0.67	0.66	0.65	**0.68**	0.65	0.69
pc4	0.78	0.80	0.77	**0.82**	0.71	0.78
philippine	0.67	0.72	0.68	0.72	0.75	0.77
porto-seguro	0.42	0.48	0.40	0.47	0.42	**0.49**
qsar-biodeg	0.84	0.83	0.86	0.82	0.78	**0.83**
steel-plates-fault	0.75	**0.83**	0.74	0.80	0.69	0.76
wilt	0.91	**0.95**	0.92	0.94	0.91	0.94
wine-quality-white	0.14	0.13	0.13	0.13	0.12	**0.15**
yeast	**0.61**	0.60	0.60	0.60	0.52	0.55

5.2.2 Comparison of proposed framework with state-of-the-art

This section evaluates the proposed framework as a whole against the most recent state-of-the-art frameworks. Just as in the inter-algorithm comparison, the test benchmark collection was used for evaluation of framework performance. The performance metrics are the same as in the previous section. Also for this part of the evaluation a Logistic Regression classifier was evaluated as well to serve as a baseline comparison for the other frameworks.

As can be observed in the graph in Figure 5.3 and Table 5.8, there are a couple of frameworks which consistently outperform the rest; this mostly consists of the AutoSklearn

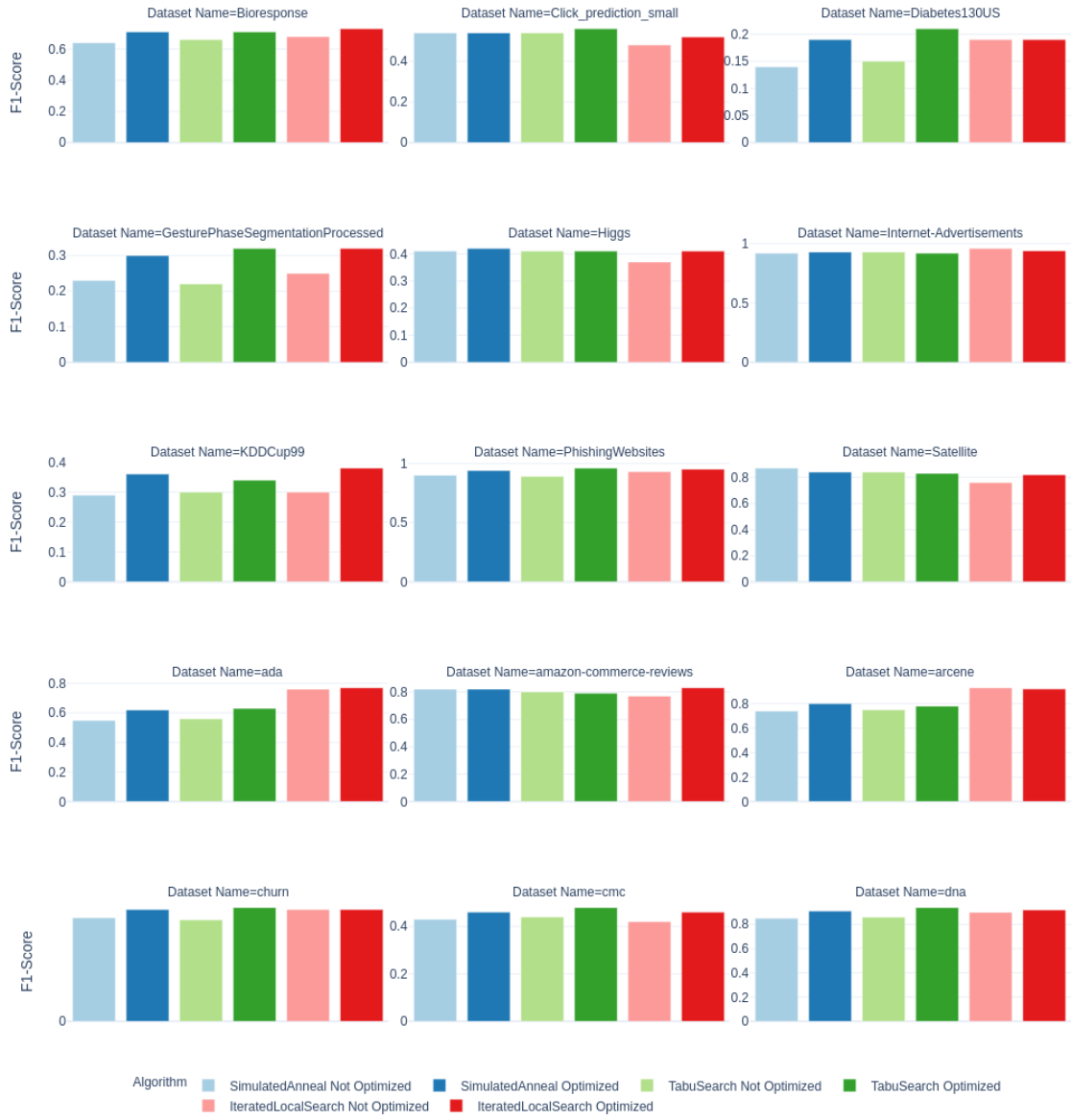

Figure 5.2: F1-Score Evaluation of Metaheuristic Algorithms on the Testing Benchmark for 15 datasets

framework as well as AutoWeka. However, even with these high performing algorithms there are still significant numbers of times where the MetaheuristicSklearn framework outperforms this set of frameworks. The proposed framework managed to find a better performing pipeline 8 out of 31 times. Consistent with previous results, this mainly consists of the ILS and Tabu Search algorithms, which seem to be working on a more feature simple optimization datasets compared to other more complex datasets inside this benchmark.

Whereas Table 5.9 illustrates the precision performance of the metaheuristic framework in comparison with state-of-the-art frameworks for the Testing Benchmark. Compared to the other frameworks, our framework was only 2% less precise than the best performing framework for all of the datasets.

Also part of the evaluation is the resulting classifier found after training has finished. Tables 5.10 and 5.11, for the best performing dataset Arcene and the worst one Higgs, display the best performing classifiers found for each framework after one hour of training.

Due to their more complex structure and wide range of parameters to optimize, AutoML problems will often require a much longer training period than standard classification problems. Therefore, as part of the evaluation, the top ten worst performing datasets for the MetaheuristicSklearn framework were put into a separate experiment with the goal of increasing the running time for allowing a greater and better search of the solution space.

The graph in Figure 5.4 and Table 5.12 display the F1-Score performances of the done experiment. The longer running time resulted in an improvement of 52% in F1-Score performance for the MetaheuristicSklearn framework and an state-of-the-art frameworks improvement of 16%.

Table 5.8: Overall Framework F1-Score Performance Evaluation on Testing Benchmark

Dataset Name	**TPOT**	**AutoWeka**	**AutoSklearn**	**HyperoptSklearn**	**AutoML Tables**	**MetaheuristicSklearn**	**SimulatedAnneal**	**Tabu**	**ILS**	**LogisticRegression**
Bioresponse	0.77	0.77	0.77	**0.80**	0.44	0.73	0.71	0.68	0.73	0.72
Click_prediction_small	0.60	0.56	0.55	0.52	0.55	**0.55**	0.54	0.55	0.54	0.54
Diabetes130US	0.40	0.40	**0.41**	0.35	0.30	0.20	0.19	0.18	0.20	0.36
GesturePhaseSegmentation-Processed	**0.67**	0.65	0.65	0.57	0.42	0.30	0.30	0.28	0.29	0.33

Table 5.8: Overall Framework F1-Score Performance Evaluation on Testing Benchmark - continued

Higgs	0.63	**0.64**	0.62	0.35	0.46	0.42	0.42	0.42	0.42	0.63
Internet-Advertisements	0.91	0.92	0.92	0.89	0.66	**0.94**	0.93	0.93	0.94	0.94
KDDCup99	0.28	0.30	0.27	0.08	0.09	**0.37**	0.36	0.34	0.37	0.14
PhishingWebsites	0.97	0.95	**0.98**	0.93	0.80	0.94	0.94	0.93	0.93	0.94
Satellite	**0.88**	0.88	0.86	0.84	0.53	0.87	0.84	0.87	0.82	0.84
ada	0.79	0.80	0.80	**0.81**	0.09	0.77	0.62	0.60	0.77	0.76
amazon-commerce-reviews	0.49	0.50	0.48	0.50	0.38	**0.83**	0.82	0.83	0.83	0.82
arcene	0.88	0.81	0.80	0.88	0.75	**0.92**	0.80	0.78	0.92	0.76
churn	0.67	**0.69**	0.68	0.64	0.51	0.55	0.54	0.54	0.55	0.57
cmc	0.45	0.43	0.47	0.46	**0.48**	0.46	0.46	0.46	0.45	0.47
dna	0.96	0.95	0.97	**0.98**	0.91	0.93	0.91	0.89	0.93	0.91
eucalyptus	0.49	0.48	**0.51**	0.51	0.28	0.46	0.44	0.46	0.43	0.47
first-order-theorem-proving	0.51	**0.52**	0.49	0.46	0.52	0.33	0.32	0.31	0.33	0.34
gina	0.93	0.97	0.97	0.89	0.93	**0.97**	0.97	0.95	0.95	0.83
kick	**0.50**	0.48	0.50	0.40	0.28	0.49	0.48	0.46	0.49	0.50
madeline	0.87	0.90	**0.91**	0.89	0.04	0.87	0.53	0.52	0.87	0.55
micro-mass	0.87	0.87	0.89	0.80	**0.90**	0.80	0.80	0.79	0.80	0.80
okcupid-stem	0.58	**0.61**	0.58	0.57	0.59	0.44	0.43	0.44	0.43	0.53
ozone-level-8hr	0.70	**0.75**	0.74	0.60	0.29	0.70	0.66	0.65	0.70	0.67
pc4	0.75	0.73	**0.77**	0.67	0.61	0.75	0.75	0.73	0.73	0.74
philippine	0.76	0.82	**0.83**	0.74	0.08	0.77	0.72	0.74	0.77	0.73
porto-seguro	0.49	0.49	0.51	0.45	0.41	**0.51**	0.49	0.51	0.47	0.49
qsar-biodeg	0.83	0.81	0.84	0.85	0.57	**0.86**	0.84	0.86	0.84	0.85
steel-plates-fault	0.80	**0.82**	0.82	0.82	0.64	0.79	0.79	0.79	0.66	0.72
wilt	0.95	0.93	0.94	**0.96**	0.34	0.95	0.94	0.95	0.92	0.69
wine-quality-white	0.40	**0.41**	0.38	0.34	0.33	0.08	0.07	0.08	0.07	0.22
yeast	0.51	0.51	0.51	0.52	0.27	**0.55**	0.53	0.55	0.51	0.49

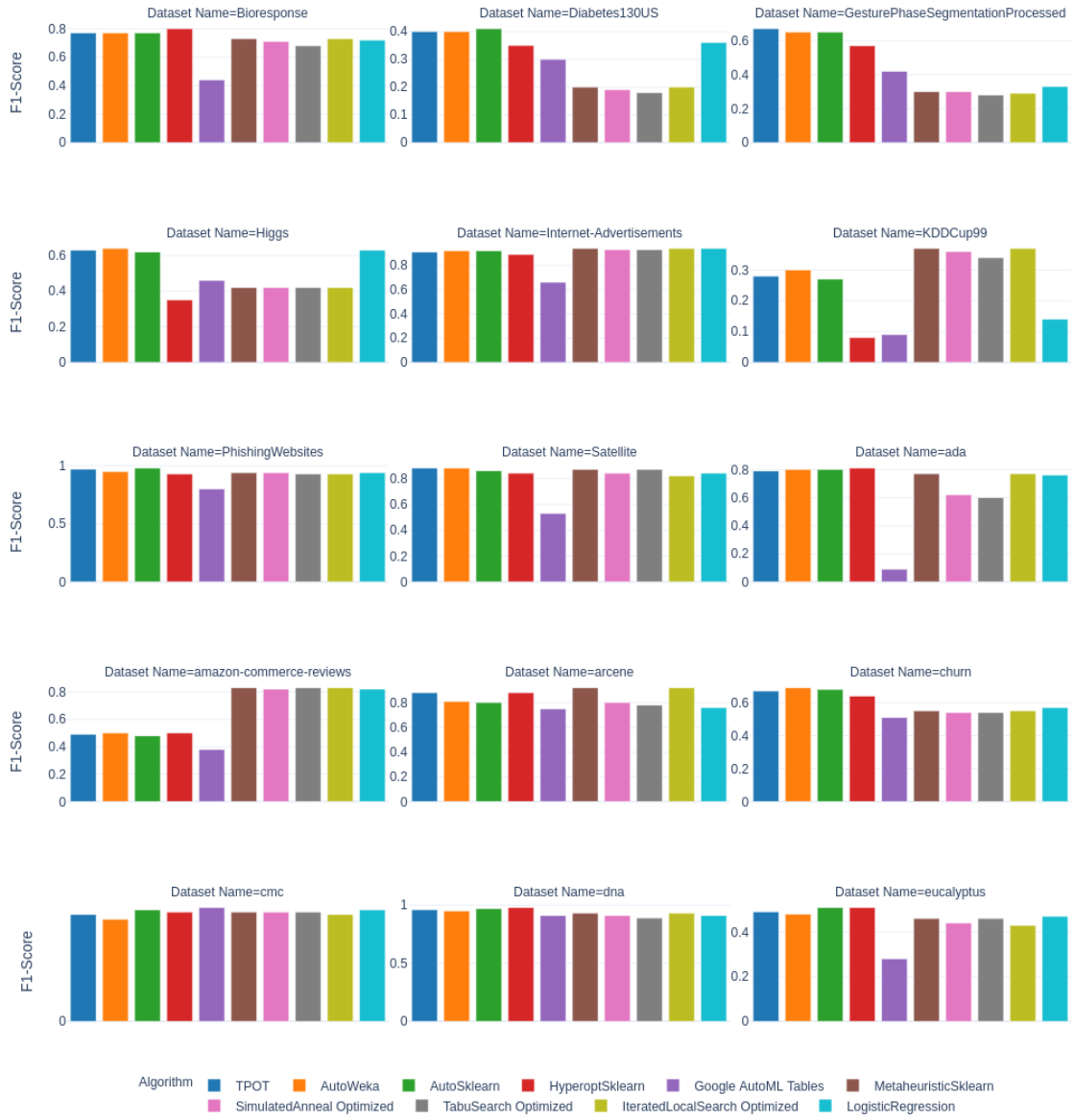

Figure 5.3: Overall Framework F1-Score Evaluation on the Testing Benchmark for 15 datasets

Table 5.9: Overall Framework Precision Performance Evaluation on Testing Benchmark

Dataset Name	TPOT	AutoWeka	AutoSklearn	HyperoptSklearn	AutoML Tables	MetaheuristicSklearn	SimulatedAnneal	Tabu	ILS	LogisticRegression
Bioresponse	**0.78**	0.77	0.77	0.69	0.61	0.72	0.71	0.72	0.70	0.72
Click_prediction_small	0.70	0.65	0.68	0.54	0.58	**0.58**	0.56	0.57	0.58	0.56
Diabetes130US	0.41	0.41	0.41	0.42	0.40	**0.44**	0.44	0.42	0.44	0.43
GesturePhaseSegmentation-Processed	0.67	**0.69**	0.65	0.61	0.56	0.48	0.46	0.48	0.45	0.46
Higgs	0.63	0.61	0.63	0.26	0.60	**0.69**	0.67	0.69	0.69	0.64
Internet-Advertisements	0.96	0.95	0.98	0.86	0.86	**0.99**	0.97	0.97	0.99	0.97
KDDCup99	0.34	0.35	0.34	0.07	0.04	**0.46**	0.44	0.42	0.46	0.13
PhishingWebsites	0.97	0.97	**0.99**	0.94	0.92	0.96	0.94	0.92	0.96	0.94
Satellite	0.93	0.93	0.92	0.84	0.54	0.92	0.92	0.90	0.90	**1.00**
ada	0.82	**0.82**	0.82	0.76	0.82	0.79	0.75	0.74	0.79	0.78
amazon-commerce-reviews	0.65	0.63	0.63	0.56	0.65	0.85	0.83	0.84	0.85	**0.85**
arcene	0.89	0.80	0.80	0.85	0.80	**0.92**	0.81	0.81	0.92	0.76
churn	0.67	0.69	0.65	0.62	0.44	**0.82**	0.80	0.80	0.82	0.81
cmc	0.46	0.46	0.47	0.41	0.21	**0.48**	0.47	0.47	0.48	0.47
dna	0.96	0.97	**0.97**	0.92	0.89	0.94	0.92	0.94	0.94	0.91
eucalyptus	0.50	**0.52**	0.49	0.48	0.23	0.48	0.46	0.48	0.48	0.48
first-order-theorem-proving	**0.52**	0.49	0.50	0.50	0.13	0.41	0.41	0.39	0.40	0.41
gina	0.93	0.97	0.97	0.92	0.97	**0.97**	0.97	0.97	0.95	0.83
kick	0.55	0.55	0.57	0.56	0.21	**0.61**	0.59	0.59	0.61	0.60
madeline	0.87	**0.91**	0.91	0.86	0.91	0.87	0.56	0.54	0.87	0.55
micro-mass	0.88	**0.90**	0.86	0.80	0.67	0.82	0.81	0.82	0.82	0.81
okcupid-stem	0.56	0.55	0.58	0.57	0.22	0.48	0.48	0.48	0.47	**0.61**
ozone-level-8hr	0.70	**0.80**	0.80	0.61	0.80	0.69	0.66	0.68	0.69	0.78

Table 5.9: Overall Framework Precision Performance Evaluation on Testing Benchmark - continued

pc4	0.76	0.74	0.75	0.66	0.46	**0.81**	0.80	0.80	0.81	0.79
philippine	0.76	**0.83**	0.83	0.67	0.83	0.77	0.72	0.73	0.77	0.73
porto-seguro	0.48	0.49	0.49	**0.51**	0.23	0.50	0.48	0.50	0.48	0.48
qsar-biodeg	0.85	**0.87**	0.86	0.81	0.81	0.84	0.83	0.82	0.84	0.85
steel-plates-fault	0.83	**0.85**	0.85	0.75	0.85	0.84	0.83	0.84	0.76	0.74
wilt	0.93	0.95	0.95	0.85	0.95	**0.96**	0.95	0.96	0.94	0.86
wine-quality-white	0.42	**0.43**	0.41	0.33	0.42	0.13	0.13	0.11	0.12	0.27
yeast	0.53	0.52	0.52	0.51	0.52	**0.61**	0.60	0.61	0.55	0.52

Table 5.10: Best performing classifier for each AutoML framework for the arcene dataset

AutoML Framework	**Best Performing Classifier**
TPOT	StackingEstimator
AutoWeka	lazy.LWL
AutoSklearn	Ensemble
Hyperopt-Sklearn	ExtraTreesClassifier
AutoML Tables	Neural Network Ensemble
MetaheuristicSklearn	RandomForestClassifier

Table 5.11: Best performing classifier for each AutoML framework for the Higgs dataset

AutoML Framework	**Best Performing Classifier**
TPOT	SGDClassifier
Auto-Weka	meta.RandomSubSpace
Auto-Sklearn	MyDummyClassifier
Hyperopt-Sklearn	SGDClassifier
AutoML Tables	Neural Network Ensemble
MetaheuristicSklearn	SGDClassifier

Figure 5.4: Overall Framework F1-Score Evaluation on the Testing Benchmark for Top 10 worst performing datasets of MetaheuristicSKlear with an extended experiment running time of 10 hours

Table 5.12: Overall Framework F1-Score Evaluation on the Testing Benchmark for Top 10 worst performing datasets of MetaheuristicSKlear with an extended experiment running time of 10 hours

Dataset Name	TPOT	AutoWeka	AutoSklearn	HyperoptSklearn	AutoML Tables	MetaheuristicSklearn	SimulatedAnneal	Tabu	ILS
GesturePhaseSegmentation-Processed	0.70	0.75	**0.76**	0.62	0.48	0.42	0.42	0.36	0.32
wine-quality-white	0.54	0.49	**0.56**	0.49	0.40	0.28	0.28	0.20	0.16
Higgs	0.71	**0.74**	0.68	0.42	0.58	0.54	0.42	0.49	0.54
Diabetes130US	0.47	0.41	0.46	**0.55**	0.44	0.40	0.26	0.21	0.40
first-order-theorem-proving	0.53	**0.65**	0.52	0.55	0.56	0.52	0.45	0.45	0.52
okcupid-stem	0.68	0.64	**0.69**	0.69	0.65	0.51	0.43	0.45	0.51
churn	0.73	**0.78**	0.69	0.66	0.63	0.67	0.67	0.56	0.61
micro-mass	0.89	0.91	0.90	0.84	**0.92**	0.84	0.81	0.84	0.83
Bioresponse	0.81	0.79	**0.84**	0.83	0.46	0.77	0.77	0.75	0.74
dna	0.97	0.95	0.97	**0.98**	0.94	0.95	0.92	0.90	0.95

CHAPTER 6

Conclusion

In this thesis, we proposed a novel metaheuristics framework that is built on the Simulated Annealing, Tabu Search, and Iterated Local Search algorithms for solving the difficult Automated Machine Learning problem.

A new framework, named *MetaheuristicSklearn*, was developed that enables the development and reproduction of multi-step supervised classification pipelines in a unified way. The MetaheuristicSklearn framework offers a common manner in which pipeline steps and parameters are implemented and integrated through different methods. In the context of solver algorithms for optimal solutions the metaheuristic algorithms Simulated Annealing, Tabu Search and Iterated Local Search were introduced.

To finally fine-adjust the framework, an extensive data set benchmark collection containing over 70 datasets was used to optimize the algorithms by SMAC as part of the framework optimization process.

In order to evaluate the framework, the three-solver algorithms have been applied in various dataset benchmarks. We used well-known literature benchmark collections, OpenML-CC18 Benchmarking Suite, where the framework was applicable to a wide variety of datasets. Inter-algorithm performance was evaluated and compared while also comparing the framework to the most state-of-the-art frameworks for solving AutoML problems.

The Tabu Search and ILS have been found to perform better than their Simulated Annealing equivalent. Where Tabu Search was the most successful algorithm top performing 15 out of 31 cases, ILS in 13 out of 31, and Simulated Annealing was only the best performing algorithm in 3 out of 31 cases. The algorithm parameter tuning process was also proven to be quite effective, where the overall improvement in F1-Score was 7% on average.

Furthermore, a better performance of pipeline 9 out of 31 times has been found through the proposed framework. The precision of the metaheuristic framework was two percent less precise than the best performing framework for all datasets compared to state of the art systems for the test benchmark.

Finally, we want to emphasize that this is a working framework that has already been applied to real-world Machine Learning problems. The implementation is very easy and can be used by anyone who knows and understands the fundamentals of AutoML. To conclude, take into account that we have developed an easy way to automate machine learning problems with state-of-the-art metaheuristics algorithms like Simulated Annealing, Tabu Search, and Iterated Local Search.

This research presents a number of opportunities in this field for further research. The future works remain to improve the precision of the MetaheuristicSklearn framework and apply the framework to more complicated issues and evaluate supervised problems of regression. We also suggest that the search space methods should be parallelized in order to enhance performance and to evaluate alternative algorithms such as population based algorithms.

List of Figures

List of Tables

Bibliography

[AL97] Emile Aarts and Jan K. Lenstra. *Local Search in Combinatorial Optimization*. John Wiley amp; Sons, Inc., USA, 1st edition, 1997.

[BB12] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *The Journal of Machine Learning Research*, 13(1):281–305, 2012.

[BBBK11] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554, 2011.

[BCdF10a] Eric Brochu, Vlad M. Cora, and Nando de Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *CoRR*, abs/1012.2599, 2010. URL: `http://arxiv.org/abs/1012.2599`, `arXiv:1012.2599`.

[BCDF10b] Eric Brochu, Vlad M Cora, and Nando De Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*, 2010.

[BCF+19] Bernd Bischl, Giuseppe Casalicchio, Matthias Feurer, Frank Hutter, Michel Lang, Rafael G. Mantovani, Jan N. van Rijn, and Joaquin Vanschoren. Openml benchmarking suites, 2019. `arXiv:1708.03731`.

[Bis19] Ekaba Bisong. *Building Machine Learning and Deep Learning Models on Google Cloud Platform: A Comprehensive Guide for Beginners*. Apress, Berkeley, CA, 2019. URL: `http://link.springer.com/10.1007/978-1-4842-4470-8`, `doi:10.1007/978-1-4842-4470-8`.

[BJP20] Víctor Blanco, Alberto Japón, and Justo Puerto. Optimal arrangements of hyperplanes for SVM-based multiclass classification. *Advances in Data Analysis and Classification*, 14(1):175–199, March 2020. URL: `http://link.springer.com/10.1007/s11634-019-00367-6`, `doi:10.1007/s11634-019-00367-6`.

[BLS13] Ilhem Boussaïd, Julien Lepagnot, and Patrick Siarry. A survey on optimization metaheuristics. *Information Sciences*, 237:82–117, 2013.

[BS91] Bruno Betrò and Fabio Schoen. A stochastic technique for global optimization. *Computers Mathematics with Applications*, 21(6):127–133, 1991. URL: https://www.sciencedirect.com/science/article/pii/0898122191901673, doi:https://doi.org/10.1016/0898-1221(91)90167-3.

[D+02] Thomas G Dietterich et al. Ensemble learning. *The Handbook of Brain Theory and Neural Networks*, 2:110–125, 2002.

[Den14] Li Deng. A tutorial survey of architectures, algorithms, and applications for deep learning. *APSIPA Transactions on Signal and Information Processing*, 3:e2, 2014. doi:10.1017/atsip.2013.9.

[DPI+20] Piali Das, Valerio Perrone, Nikita Ivkin, Tanya Bansal, Zohar S. Karnin, Huibin Shen, Iaroslav Shcherbatyi, Yotam Elor, Wilton Wu, Aida Zolic, Thibaut Lienart, Alex Tang, Amr Ahmed, Jean Baptiste Faddoul, Rodolphe Jenatton, Fela Winkelmolen, Philip Gautier, Leo Dirac, Andre Perunicic, Miroslav Miladinovic, Giovanni Zappella, Cédric Archambeau, Matthias W. Seeger, Bhaskar Dutt, and Laurence Rouesnel. Amazon sagemaker autopilot: a white box automl solution at scale. *CoRR*, abs/2012.08483, 2020. URL: https://arxiv.org/abs/2012.08483, arXiv:2012.08483.

[EFH+13] Katharina Eggensperger, Matthias Feurer, Frank Hutter, James Bergstra, Jasper Snoek, Holger Hoos, and Kevin Leyton-Brown. Towards an empirical foundation for assessing bayesian optimization of hyperparameters. In *NIPS workshop on Bayesian Optimization in Theory and Practice*, volume 10, page 3, 2013.

[EHH18] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural Architecture Search: A Survey. *arXiv e-prints*, page arXiv:1808.05377, August 2018. arXiv:1808.05377.

[EMS19] Radwa Elshawi, Mohamed Maher, and Sherif Sakr. Automated machine learning: State-of-the-art and open challenges. *arXiv preprint arXiv:1906.02287*, 2019.

[FD02] Dimitris Fouskakis and David Draper. Stochastic optimization: a review. *International Statistical Review*, 70(3):315–349, 2002.

[FH19] Matthias Feurer and Frank Hutter. Hyperparameter optimization. In *Automated Machine Learning*, pages 3–33. Springer, Cham, 2019.

[Fil19] Peter Filzmoser. Lecture notes in advanced methods for regression and classification, October 2019.

[FKE+15] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Advances in neural information processing systems*, pages 2962–2970, 2015.

[GJ05] Michael J. Grimble and Michael A. Johnson, editors. *Introduction to Genetic Algorithms*, pages 325–336. Springer London, London, 2005. doi:10.1007/1-84628-121-0_14.

[Glo90] Fred Glover. Tabu search: A tutorial. *Interfaces*, 20(4):74–94, 1990.

[Glo95] Fred Glover. *Tabu search fundamentals and uses*. Graduate School of Business, University of Colorado Boulder, 1995.

[GLP+19] P. Gijsbers, E. LeDell, S. Poirier, J. Thomas, B. Bischl, and J. Vanschoren. An open source automl benchmark. *arXiv preprint arXiv:1907.00909 [cs.LG]*, 2019. Accepted at AutoML Workshop at ICML 2019. URL: https://arxiv.org/abs/1907.00909.

[HHLB11] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization*, pages 507–523. Springer, 2011.

[HS04] Daniel P Heyman and Matthew J Sobel. *Stochastic models in operations research: stochastic optimization*, volume 2. Courier Corporation, 2004.

[HSCS19] Kashif Hussain, Mohd Najib Mohd Salleh, Shi Cheng, and Yuhui Shi. Metaheuristic research: a comprehensive survey. *Artificial Intelligence Review*, 52(4):2191–2233, 2019.

[KBE14] Brent Komer, James Bergstra, and Chris Eliasmith. Hyperopt-sklearn: automatic hyperparameter configuration for scikit-learn. In *ICML workshop on AutoML*, volume 9, page 50. Citeseer, 2014.

[KGV83] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[KH00] Alexandros Kalousis and Melanie Hilario. Model selection via meta-learning: A comparative study. pages 406 – 413, 02 2000. doi:10.1109/TAI.2000.889901.

[LLG12] Hongtao Lei, Gilbert Laporte, and Bo Guo. A generalized variable neighborhood search heuristic for the capacitated vehicle routing problem with stochastic service times. *Top*, 20(1):99–118, 2012.

[LMS03] Helena R. Lourenço, Olivier C. Martin, and Thomas Stützle. *Iterated Local Search*, pages 320–353. Springer US, Boston, MA, 2003. doi:10.1007/0-306-48056-5_11.

[MF] Arlind Kadra Pieter Gijsbers Neeratyoy Mallik Sahithya Ravi Andreas Mueller Joaquin Vanschoren Frank Hutter Matthias Feurer, Jan N. van Rijn. Openml-python: an extensible python api for openml. *arXiv*, 1911.02490. URL: https://arxiv.org/pdf/1911.02490.pdf.

[Moc94] Jonas Mockus. Application of bayesian approach to numerical methods of global and stochastic optimization. *Journal of Global Optimization*, 4(4):347–365, 1994.

[noaa] auto-sklearn — AutoSklearn 0.8.0 documentation. URL: https://automl.github.io/auto-sklearn/master/.

[noab] AutoML Tables features and capabilities. URL: https://cloud.google.com/automl-tables/docs/features.

[noac] Examples - TPOT. URL: http://epistasislab.github.io/tpot/examples/.

[noa20] hyperopt/hyperopt-sklearn, July 2020. original-date: 2013-02-19T16:09:53Z. URL: https://github.com/hyperopt/hyperopt-sklearn.

[OL96] Ibrahim H Osman and Gilbert Laporte. Metaheuristics: A bibliography, 1996.

[OM19] Randal S. Olson and Jason H. Moore. TPOT: A Tree-Based Pipeline Optimization Tool for Automating Machine Learning. In Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren, editors, *Automated Machine Learning*, pages 151–160. Springer International Publishing, Cham, 2019. URL: http://link.springer.com/10.1007/978-3-030-05318-5_8, doi:10.1007/978-3-030-05318-5_8.

[OUA+16a] Randal S. Olson, Ryan J. Urbanowicz, Peter C. Andrews, Nicole A. Lavender, La Creis Kidd, and Jason H. Moore. *Applications of Evolutionary Computation: 19th European Conference, EvoApplications 2016, Porto, Portugal, March 30 – April 1, 2016, Proceedings, Part I*, chapter Automating Biomedical Data Science Through Tree-Based Pipeline Optimization, pages 123–137. Springer International Publishing, 2016. URL: http://dx.doi.org/10.1007/978-3-319-31204-0_9, doi:10.1007/978-3-319-31204-0_9.

[OUA+16b] Randal S. Olson, Ryan J. Urbanowicz, Peter C. Andrews, Nicole A. Lavender, La Creis Kidd, and Jason H. Moore. Automating biomedical data science through tree-based pipeline optimization. *arXiv:1601.07925 [cs]*, January 2016. arXiv: 1601.07925. URL: http://arxiv.org/abs/1601.07925.

[Pol12] Robi Polikar. Ensemble learning. In *Ensemble machine learning*, pages 1–34. Springer, 2012.

[RLMS00] Helena Ramalhinho-Lourenço, Olivier C Martin, and Thomas Stützle. Iterated local search. 2000.

[RM51] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, pages 400–407, 1951.

[Rut89] Rob A Rutenbar. Simulated annealing algorithms: An overview. *IEEE Circuits and Devices Magazine*, 5(1):19–26, 1989.

[Sch13] Robert E Schapire. Explaining adaboost. In *Empirical inference*, pages 37–52. Springer, 2013.

[Spa03] James C. Spall. *Introduction to Stochastic Search and Optimization: Estimation, Simulation, and Control*. John Wiley & Sons, Inc., Hoboken, NJ, USA, March 2003. URL: http://doi.wiley.com/10.1002/0471722138, doi:10.1002/0471722138.

[THHLB13] Chris Thornton, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Auto-weka: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, page 847–855, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2487575.2487629.

[TyHCPP] Jack Smyth Tin-yun Ho, Enrique Ibarra Anaya Chris Pocock, and Carlos S. Perez. Tackling high-value business problems using automl on structured data (cloud next '19). URL: https://www.youtube.com/watch?v=MqO_L9nIOWM.

[Vap95] Vladimir N. Vapnik. *The Nature of Statistical Learning Theory*. Springer New York, New York, NY, 1995. URL: http://link.springer.com/10.1007/978-1-4757-2440-0, doi:10.1007/978-1-4757-2440-0.

[VLA87] Peter JM Van Laarhoven and Emile HL Aarts. Simulated annealing. In *Simulated annealing: Theory and applications*, pages 7–15. Springer, 1987.

[VvRBT13] Joaquin Vanschoren, Jan N. van Rijn, Bernd Bischl, and Luis Torgo. Openml: Networked science in machine learning. *SIGKDD Explorations*, 15(2):49–60, 2013. URL: http://doi.acm.org/10.1145/2641190.2641198, doi:10.1145/2641190.2641198.

[WM97] D.H. Wolpert and W.G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, April 1997. URL: http://ieeexplore.ieee.org/document/585893/, doi:10.1109/4235.585893.

[Yan10] Xin-She Yang. *Nature-inspired metaheuristic algorithms*. Luniver press, 2010.

[YL99] Yiming Yang and Xin Liu. A re-examination of text categorization methods. In *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 42–49, 1999.

[ZM12] Cha Zhang and Yunqian Ma. *Ensemble machine learning: methods and applications*. Springer, 2012.