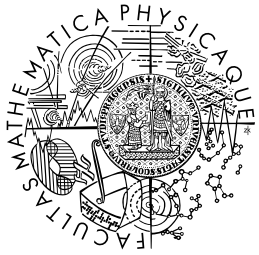

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Rajat Sharma

Evolutionary techniques in AutoML

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis: Mgr. Roman Neruda
Study programme: Computer Science

Prague 2024

I declare that I carried out this master thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

First and foremost, I would like to thank Professor Roman Neruda for his incredible guidance and amazing leadership, without which this would have been impossible. I would also like thank everyone who supported me during the writing of this master's thesis.

Title: Evolutionary techniques in AutoML

Author: Rajat Sharma

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Roman Neruda, CSc.

Abstract: AutoML methods search for a suitable pipeline of preprocessing and ML components given a data set representing an ML task. The goal of the thesis is to design an evolutionary optimization algorithm which will search the space of pipelines and propose an optimized solution. Several search approaches, such as hill climbing, simulated annealing and evolutionary search are tested. Implementation of developed algorithms using standard machine learning libraries such as scikit-learn, and their experimental evaluation on benchmark data as also a part of the work.

Keywords: Machine learning Evolutionary computing AutoML

Contents

Introduction

In the past recent years, artificial intelligence has reached new heights. It has officially became a household name. It has influenced everything we see or encounter on our daily basis. They range from fields like finance to optimising trading strategies and fraud detection, transportation, where it powers autonomous vehicles, automated analyses of documents, through image recognition all the way to music recommendation. This all became possible because of the advances in machine learning that we were able to witness a great variety of applications. Hence, artificial intelligence has played a key role in initiating Industry 4.0.

When we talk about these advances in machine learning, what we really mean is that we are able to solve complex problems and complete tasks more efficiently than ever before. These complex problems are tackled by a large team of machine learning experts and data scientists. With their careful and smart approach, they are able to choose what is best for their problem, be it selecting the best parameters for their learning process or choosing the best optimal model according to their dataset size.

But then there are teams, who do not have enough budget, time or even correct knowledge about which parameters or models to use for their research. Many times they end up using a configuration for their system, which either takes too long to design or ends up with a system which does not produce promising results.

This is where AutoML comes into the picture. AutoML stands for automated machine learning. This is a recently developed area whose main purpose is to solve these kinds of issues in the first place. It helps in atomizing the entire workflow for a user, providing them with some decent results to begin with. Afterwards, they can improve these results by updating the configuration further as per their requirement. In many cases, it can even suggest a list of models based on the input dataset.

This brings to our topic. Our main goal is to design such a system that creates pipelines for all the possible combinations of preprocessors and classifiers. Then use multiple search techniques to find the best possible pipelines for the given input dataset. For our experimentations, we are using the datasets of different sizes (as seen in section 4.1 and 4.3). As for the results of these experiments, we have a dedication section, i.e. section 4.2 for that purpose.

Regarding the overall structure of the thesis, we have divided it into four chapters and a conclusion. In chapter 1, we have three subsections, mainly machine learning (section 1.1), AutoML (section 1.2), and our dedicated search algorithms (section 1.3).

In machine learning, we provide a basic overview of the subject followed by the performance and ensembles of models. In AutoML, we have took a deep insight about what is the workflow in machine learning, followed by hyperparameter optimization and lastly, model selection and it's criteria. At the end of this section comes the most promising part, where we discuss all the kinds of searches we have used in our computations, both exhaustive and smart searches.

Coming to chapter 2, it is called related work. It contains a brief introductions about the existing AutoML systems in the present world.

Chapter 3 forms the base of how our machine learning pipelines fits into these

local (hill climbing and simulated annealing) and global (evolutionary algorithm) searches. We describe how we were able to optimise the pipelines we have in the experiments (section 3.1). This is followed by the evaluation of the different kinds of searches used during the experiments (section 3.2). In the last section, i.e. section 3.3, we have written some implementation regarding the project. It contains the details about where this entire project is hosted and while libraries were used.

Finally, we have the last chapter, i.e. chapter 4, where we have discussed all the aspects of our experiments in great detail. We start by defining all the datasets we used throughout our experimentation (section 4.1). This is followed by the results and plots in section 4.2, where we used the datasets mentioned in section 4.1. At the last section. i.e. section 4.3, we ran our experiments on various dataset of OpenML-CC18 benchmark as well.

1 Preliminaries

In this chapter, we will be discussing all the components of our experimentation from the theoretical point of view. We start with the brief overview of machine learning, followed by the performance and ensembles of models. Then we move on to the section 1.2, i.e. AutoML. Here, we take a deep insight about what is the workflow in machine learning, followed by hyperparameter optimization and lastly, model selection and it's criteria. Finally, we discuss all the different search algorithms we have utilized during our experimentation.

1.1 Machine Learning

In simple terms, machine learning is a field of study that gives computers a capability to learn without being explicitly programmed. Machine learning in itself covers a broad range of statistical methods and a wide variety of algorithms for data processing. In his book, Flach defines machine learning as the systematic study of algorithms and systems that improve their knowledge or performance with experience [1]. For example, personalized recommendation. This uses a user's past behavior to suggest items they might like, or have purchased in the past, making their shopping experience more tailored to their preferences. So this field of machine learning can easily be divided into three separate subdomains - supervised learning, unsupervised learning and reinforcement learning.

In the case of supervised learning, we train our models on training data which is correctly labelled. So the primary task here is to predict the correct labels, also sometimes called the target. There are two types of supervised learning — classification and regression. Classification refers to problems where the labels are from a finite set of distinct categories. And regression contains the problems where the labels can take on any real number.

In the case of unsupervised learning, the main goal is to discover patters in our dataset. This is because the data we have is not labelled, meaning, it does not has any target labels. So we use clustering techniques and association rules to find meaningful patters in our input dataset.

And lastly, the reinforcement learning. Here, an agent and an environment continuously interact with each other. The goal for the agent is to gather as many rewards from the state as possible in order to maximize the reward function.

Out of all the types of machine learning, our work focused on the supervised learning aspect of it.

1.1.1 Performance

In this section, we discuss in detail regarding evaluating the performance of machine learning models. Let's start with a formal definition of supervised learning. According to Abhishek, performance metrics play a crucial role in evaluating the effectiveness and accuracy of machine learning models. They provide insights into a model's predictive capabilities and help measure its performance across various tasks [2].

- **Evaluation Metrics**: Various metrics like accuracy, precision, recall, and F1-score are used to measure how well a model performs.
- **Overfitting and Underfitting**: Overfitting and underfitting occur when a model is too complex or too simple respective, thereby affecting its performance.
- **Cross-Validation**: Techniques such as k-fold cross-validation are used to assess model performance by splitting data into training and validation sets.
- **Bias-Variance Tradeoff**: This is when models face a tradeoff between bias (underfitting) and variance (overfitting), thereby impacting their performance. The general idea is to find a balance between both of them.
- **Learning Curves**: Learning curves depict how a model's performance changes with training set size or iterations.

1.1.2 Model Ensembles

The word ensemble means a group of musicians, actors etc. In the machine learning domain, when we combine a large number of models, it is called ensemble learning. And it's a very powerful technique to produce some great predictions. As mentioned in Pytorch's official website, the idea of model ensembling is to combine the predictions taken from multiple models together. This solution is achieved by running each model on some inputs separately and at the end, we combine their predictions [3].

As Evan mentioned in this article, it is very clear that the final goal of any machine learning problem is to find that one single model which will produce the best results. So the idea of ensembling is that rather than making one machine learning model and hoping that model will produce the best outcomes, we can instead make multiple models and average those models to produce one final model [4].

According to Derrick Mwiti, a famous data scientist and writer, ensemble learning can help improve the performance of machine learning models by a good proportion. By that, he means the overall accuracy increment or reduction of error. Besides, he also believes that ensembling results in a more stable model. [5].

There are many ensembling techniques using in the machine learning domain. The most used ones are:

- **Bagging (Bootstrap Aggregating):** it uses n bootstrapped samples to train n models, aggregating predictions by voting or averaging for stability and diversity.
- **Boosting:** this technique weights training examples to focus on misclassified instances, iterating until a set threshold, with final predictions weighted by model accuracy.

1.2 AutoML

There is a phenomena in machine learning called 'non free lunch' theorem [6], which states that we cannot create just one machine learning algorithm model and it will outperform all the other algorithms. So we cannot say for certain which model is suitable for which dataset and problem in hand. This brings us to the topics of model selection (discussed in detail in section 1.3) and hyperparameter selection (discussed in section 1.2.2). Both these parameters are very essential and must be used in order to gather optimal solutions for the problem set. But doing all of this selectively requires great setup, time and processing power to yield promising results in the first place.

This is where AutoML comes into the picture. Houssam thinks that AutoML can be used to anticipate behaviour (probability that a customer will leave the company, abandon a purchase, cancel a reservation, etc.), segment populations, detect fraud, predict the imminence of an event (predictive maintenance, etc.) or establish sales forecasts [7].

1.2.1 Workflows in ML

When it comes to the idea of workflow in a machine learning model, Regan has narrowed it down to the most essential steps. According to him, we should start with the project preparation. This is followed by data collection or data preparation. And then comes the most crucial step, i.e. machine learning modelling. This is where all the feature engineering, model selection and evaluation are done. And then, we have the deployment of the model. From time to time, we need to keep a track of all the activities happening in the entire workflows. And for that purpose, we have the final step of monitoring the entire pipeline [8].

Speaking of pipeline, it is nothing but a series of configurations put together and evaluated. We have an entire section 3.1 dedicated to the creation of these pipelines, specifically made in the manner so that it can integrate with our experimentation perfectly.

Coming back to the ml workflow, Figure 1.1 captures the overview of how a typical machine learning workflow looks like. The most crucial step is to figure out the problem definition and the dataset that can be used to find such an optimal solution for this problem.

One of the leading software AI company, RunAi says the ml workflows are really necessary part of the machine learning ecosystem. They forms the basic definition of that steps we need to follow, in which exact manner and sequence, in order to fetch the best optimal solutions and smooth working of the pipeline. So, in short, these workflows follows a sequence of systemic tasks from problem formulation all the way to the deployment of the model [9].

1.2.2 Hyperparameter Optimization

When we run the machine learning models with some predefined parameters, we end up getting some results. But we have seen that if we tweak those parameters, we may end up getting much better optimal solutions to our machine learning problem. This is the underlying concept of Hyperparameter optimization.

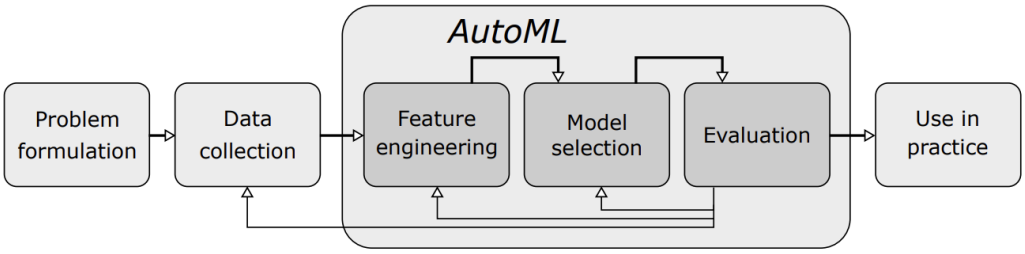

Figure 1.1 A typical machine learning workflow.

That is why, hyperparameter optimization deals with finding the best hyperparameters of a model. This will help the model performs the best and eventually, yields best optimal results. Thornton believed this as one of the most critical components of AutoML. The motive is to find the best set of hyperparameters for our model [10].

1.2.3 Model Selection

In this section we will take a look at the concept of model selection. If well look from the view point of machine learning and artificial intelligence, model selection could be very different. It can even have different meanings depending upon different levels of layers. So, in short, we do model selectionin order to select the best performing model architecture for our needs or the size of the dataset, depending upon problem definition [11].

There are so many important thing we have to keep in mind before selecting a model for our machine learning problem. That's why choosing an effective machine learning model is crucial and involves considering several key factors.

Firstly, we need to assess the complexity of the problem to determine if we would need a simple or more complex model. We can look at the dataset size, feature complexity, and nonlinear relationships to determine this factor. Next, we can evaluate data availability and quality in order to avoid overfitting with models that match the dataset's scale and quality. Finally, this is the time to employ our domain expertise and consider ensemble methods for enhanced model performance through diversified predictions. Let's see some of the frequently use methods for selecting models:

- **Train-Test Split:** Data is split into training and testing sets. Model performance is evaluated on the testing set after training set.
- **Cross-Validation:** Data is split into folds. This way, models are trained and evaluated on each fold to reduce evaluation variance.
- **Grid Search:** Exhaustive search can be used over predefined hyperparameter values to find the best model configuration. This usually provides promising results, but the time it takes to reach those results is very high.
- **Random Search:** Random sampling of hyperparameter values to explore a subset of the search space efficiently. This technique can sometimes prove beneficial as well.

- **Model Averaging:** We can combines predictions from multiple models to improve overall accuracy, reducing bias and variance.
- **Domain Expertise:** Sometimes, we have prior knowledge about the configuration or dataset that impacts model selection.

1.3 Search Algorithms

In this section, we will discuss about the search algorithms in detail. Moreover, we will look more closely into all the searches we used during our experimentation. To start with, search algorithms fall into the category of optimization methods that focus on iteratively moving from one solution to a neighboring solution. This is done by evaluating solutions based on a heuristic function, which provides the value or quality of the solution. Basic purpose of the search algorithms is to find the best solution either in the vicinity of the current state, or exploring the entire solution space, depending upon the kind of searching we are performing.

Selecting an appropriate optimization method is crucial. While local search algorithms are effective for many problems, they are not universally applicable. The choice of method should be driven by problem characteristics, such as the nature of the solution space, the presence of local optima, the scalability of the problem, and available computational resources. In some cases, global optimization methods or hybrid approaches may be better suited to the problem's requirements. It's essential to understand the problem landscape and tailor the optimization approach accordingly to achieve the desired results. [12]

1.3.1 Exhaustive Search

Exhaustive search, also known as brute-force search, is a straightforward optimization method that systematically evaluates every possible solution to find the optimal one. Unlike heuristic methods, exhaustive search does not rely on heuristics or domain-specific knowledge but instead checks every possible solution within a defined search space.

A general schema of exhaustive search is presented in Algorithm 1. The process begins by systematically generating and evaluating each possible solution within the search space (lines 2-6). The algorithm continues until all solutions have been evaluated (line 7). Given its exhaustive nature, this method guarantees finding the optimal solution, but it can be computationally expensive, especially for large search spaces.

- **Search Space:** All possible solutions that the algorithm evaluates, dictated by problem complexity and decision variables.
- **Objective Function:** Scores solutions numerically, guiding the algorithm towards optimal or near-optimal solutions.
- **Stopping Criteria:** Terminates when all solutions in the search space are evaluated, directly impacting computational feasibility for larger problems.

Algorithm 1 Exhaustive Search

Require: Search space S, objective function f
Ensure: Optimized solution

```
1: x_best ← None
2: f_best ← −∞                ▷ Initialize with a very low value for minimization
3: for x in S do
4:     f_x ← f(x)
5:     if f_x > f_best then
6:         x_best ← x
7:         f_best ← f_x
8:     end if
9: end for
10: return x_best
```

In summary, exhaustive search is a reliable optimization technique that guarantees finding the optimal solution within the constraints of the defined search space. Its effectiveness lies in its systematic evaluation of all possible solutions, making it suitable for problems where the search space is manageable and computational resources allow for exhaustive evaluation.

1.3.2 Hill Climbing

Hill climbing is a heuristic optimization algorithm that iteratively improves a candidate solution by adjusting it in iterative manner to obtain a higher quality solution. It's analogous to climbing a hill where each step moves towards a higher point on the landscape. So basically, you keep on moving in a direction of increasing value, something like a hill. And we stop this process when we notice there are no more neighbours which can have a higher value [13].

In the case of hill climbing, we start by randomly generating a solution. Then, the algorithm evaluates the neighboring solutions, and moves to the best neighbor if it offers an improvement. This process is continued until a local maximum is reached where no better neighbors exist.

A general schema of hill climbing is presented in Algorithm 2. The process begins with initializing a solution randomly or heuristically (line 1 of Algorithm 2). At each iteration, the algorithm evaluates neighboring solutions (line 3). If a neighboring solution offers an improvement, the current solution is updated (line 6). The algorithm terminates when no better solution can be found in the neighborhood (line 7).

Let's discuss the components of hill climbing in more depth.

- **Objective function:** it evaluates the quality of a solution. Its primary role is to guide the search towards optimal (or near-optimal) solutions by assigning a numerical score to each candidate solution.

- **Neighborhood search:** it explores nearby solutions of the current solution. This step is crucial as it checks if a neighboring solution offers an improvement, updating the current optimal solution accordingly.

Algorithm 2 Hill Climbing

Require: Initial solution x, stopping condition c
Ensure: Optimized solution

1: $x_{\text{current}} \leftarrow x$ ▷ Initialization
2: **while** c is not met **do**
3: Find neighboring solutions of x_{current}
4: $x_{\text{neighbor}} \leftarrow$ select the best neighboring solution
5: **if** $f(x_{\text{neighbor}}) > f(x_{\text{current}})$ **then**
6: $x_{\text{current}} \leftarrow x_{\text{neighbor}}$ ▷ Update current solution
7: **end if**
8: **end while**
9: **return** x_{current} ▷ Return optimized solution

- **Stopping criteria:** it determines when the hill climbing process terminates. Common criteria include reaching a certain number of iterations or when no better solution is found in the neighborhood.

So to summarize this, hill climbing is a very straightforward yet effective optimization technique. It is primarily suitable for problems where a good initial solution can lead to satisfactory results, something like how we have implemented in our experiments. We did this step by calling one of our functions to generate a random pipeline and initialize the search for the optimal solution from there. But again, hill climbing is prone to getting stuck in local optima. So we have to be careful while making the parameter tuning to achieve desired results.

1.3.3 Simulated Annealing

Simulated annealing is an optimization technique, which is inspired by the annealing process in metallurgy. The basic idea behind is that it mimics the annealing process where a material is heated and then gradually cooled to decrease defects, which ultimately leads to a more stable crystalline structure.

Just like in hill climbing, in SA, we again start with an initial solution which is randomly generated. In our case, it was a pipeline, as discuss in section 3. The algorithm start with iteratively adjusting its solution, exploring nearby solutions with the hope of finding an optimal or near-optimal solution.

A general schema of simulated annealing (SA) is presented in Algorithm 3. The process begins with initializing a solution randomly or heuristically (line 1 of Algorithm 3). At each iteration, a candidate solution is randomly generated around the current solution (line 5). The quality of this candidate solution is evaluated using an objective function (line 6). The algorithm then decides whether to move to this candidate solution based on its quality and the current temperature (line 8). The temperature decreases over time according to a cooling schedule (line 3). This cooling process controls the probability of accepting worse solutions early in the search and encourages exploration of the solution space. The algorithm terminates when a stopping condition is met (line 9), typically after a predetermined number of iterations or when the temperature drops below a certain threshold.

Moving on, let's discuss the components of simulated annealing in detail.

Algorithm 3 Simulated Annealing

Require: Initial solution x, stopping condition c, initial temperature T_{init}, and cooling rate α

Ensure: Optimized solution

$x_{\text{current}} \leftarrow x$ ▷ Initialization
$T \leftarrow T_{\text{init}}$ ▷ Initial temperature
while c is not met **do**
    Generate a candidate solution $x_{\text{candidate}}$ near x_{current}
    Compute objective value $f(x_{\text{candidate}})$
    if $f(x_{\text{candidate}}) < f(x_{\text{current}})$ or $\text{random}(0,1) < e^{-\frac{f(x_{\text{candidate}})-f(x_{\text{current}})}{T}}$ **then**
        $x_{\text{current}} \leftarrow x_{\text{candidate}}$ ▷ Move to candidate solution
    end if
    $T \leftarrow \alpha \cdot T$ ▷ Cooling schedule
end while
return x_{current} ▷ Return optimized solution

- **Objective function:** It evaluates the quality of a solution by assigning a numerical score to each candidate solution, guiding the search towards optimal or near-optimal solutions.
- **Temperature schedule:** This schedule determines how the algorithm explores the solution space. Initially, the high temperature allows the algorithm to accept worse solutions more readily. As temperature decreases, acceptance of worse solutions decreases, balancing exploration and exploitation crucial for simulated annealing success.
- **Cooling schedule:** It controls the rate at which temperature T decreases over time, shifting the algorithm from exploring the solution space (at higher temperatures) to exploiting promising solutions (at lower temperatures).
- **Stopping criteria:** These criteria determine when the simulated annealing process should be terminated. We have several ways to figure this out, i.e. we can set a fixed number of iterations and we can terminate once that number is reached. Another technique is to stop the algorithm when the temperature drops below a predefined threshold.

In summary, simulated annealing is a versatile optimization technique suitable for a wide range of problems. By balancing exploration and exploitation through temperature control, it efficiently searches for optimal or near-optimal solutions in complex solution spaces.

1.3.4 Evolutionary Algorithm

Charles Darwin's theory of natural selection has a great impact on the world of machine learning ecosystem. One such impact was the introduction of evolutionary algorithm, which is entirely inspired by biological evolution and natural selection [14]. More precisely, evolutionary algorithm is a specific application of evolutionary computing techniques within the area of optimization and search problems. These

algorithms iterate through populations, where individuals compete randomly and the fittest individuals are selected over time. This closely mimics the idea behind *survival of the fittest*, which states that the beings with the best survival traits will live and end up creating new offspring and passing their traits to them. Overtime, the evolution will continue its course and in a matter of some generations, we shall notice that the most strongest traits predominate.

Moving onto Algorithm 4, we can see the working of EA. It begins by initializing a population of individuals (line 1) and selecting the parents for reproduction (line 7). Here, the process of reproduction creates new individuals until the population reaches its original size. Afterwards, environmental selection combines these individuals from the entire group.

So keep on running this algorithm until a stopping condition is met (line 3). This stopping conditions could be set as a number of iterations or generations are achieved by the algorithm. The main idea behind all this is to have the final population, which will consist of individuals with the highest fitness values.

Algorithm 4 Evolutionary Algorithm

Require: Initial population, crossover probability, mutation probability
Ensure: Evolved individuals
1: Initialize population
2: **while** Stopping condition is not met **do**
3: **for** each individual in current population **do**
4: Compute fitness
5: **end for** ▷ Reproduction
6: **for** each pair of individuals **do** ▷ Select parents and create offspring
7: Select parents
8: Perform crossover and mutation
9: Add offspring to new population
10: **end for** ▷ Environmental selection
11: Select individuals for next generation
12: **end while**
13: **return** Final population

Here's few essential terms related to the concept of evolutionary algorithm.

- **Parent Selection:** This step selects the parents for mating to maintain diversity. It also balances better and worse individuals, in order to maintain diversity.
- **Crossover and Mutation:** These are reproduction operators that alter genetic structures to create offspring. In simple terms, crossover combines traits of parents, while the mutation introduces diversity.
- **Fitness Evaluation:** It assesses the quality of individuals based on objective functions, which guides the selection and evolutionary process.
- **Environmental Selection:** It creates a new population deterministically by favoring individuals with higher fitness.

- **Stopping Criteria:** We can either set the certain number of generations as the limit. Once we reach that limit, we can terminate the algorithm.

2 Related work

In recent years, the field of automated machine learning (AutoML) has seen significant advancements, mainly aimed at automating the model selection, hyperparameter tuning, and pipeline optimization processes. AutoML systems helps across various domains, starting from traditional machine learning all the way to deep learning approaches.

Let's start with discussing one such prominent AutoML system, called TPOT. TPOT utilises genetic programming to automatically explore and optimise machine learning pipelines composed of preprocessing steps and estimators. Its main focus is on generating efficient pipelines through evolutionary algorithms. As of now, its scope primarily covers simple model architectures and ensemble methods within the scikit-learn ecosystem [15].

Then, we have deep learning-oriented AutoML approaches, such as those discussed by Elsken et al. (2018) [16], which delve into Neural Architecture Search (NAS) to automate the design of deep neural network architectures. These methods primarily use strategies like bayesian optimization and reinforcement learning to efficiently explore the huge state space. This helps them achieve state-of-the-art results in tasks like image recognition and object detection.

Even classical machine learning AutoML tools, such as Auto-WEKA [10] and its successor Auto-sklearn, employ Bayesian optimization techniques for hyperparameter tuning and model selection. These systems typically focus on optimising simpler model configurations and ensemble structures. Ultimately, their technique limits the complexity of the pipelines they can generate.

Our work contributes to this landscape by introducing a pipeline configured with a random number of numerical and categorical preprocessors followed by a classifier. This approach helps in finding the optimal solution for any incoming dataset. It starts by segregating the columns of the input dataset into numerical columns and categorical columns. This is followed by applying numerical and categorical preprocessors respectively to these columns. In some cases, we have even applied common preprocessors to the pipeline. Then we select a classifier from the wide range of classifiers we have in our system. At last, we run this pipeline across the dataset and find the accuracy. We do this for all the combinations of numerical preprocessors, categorical preprocessors and classifiers we used in our experiments. Unlike existing systems, our approach aims to find the best optimal solutions for a given dataset. In the end, we evaluated our searches, both brute force and smart searches on benchmarks like OpenML-CC18.

3 Our Search Solution

In this chapter, we will go deep about studying our pipelines and integration of those pipelines with our local and global search to find the best optimal solutions for our input dataset (mentioned in section 4.1). So the entire base of our experimentation lies in creating pipelines of different combinations of preprocessors, classifiers, and leveraging sophisticated algorithms. This system ultimately streamlines the journey from raw data to a finely-tuned model, thereby enhancing both efficiency and performance

We have mentioned Algorithm 5, which shows the pipeline optimization steps clearly. As we can see, the input is the dataset and the main objective is to find that pipeline or set of pipelines that produces the best optimal solution for our dataset. The configuration related to construction of pipelines are discussed thoroughly in section 3.1. And for its quick overview, you can refer to the Tables 3.1 and 3.2. The dataset provided to the algorithm does not need to be preprocessed, as we have plenty of preprocessors in the inventory to perform imputation of missing values.

Algorithm 5 Pipeline Optimization — Main

1: **Data:** dataset d
2: **Result:** optimised pipelines
3: split dataset d into numerical and categorical columns
4: generate pipelines using preprocessors and classifiers
5: run pipelines on d
6: select pipelines with the best accuracy
7: return optimised pipelines

3.1 Optimization of pipelines

In this section, we describe the necessary components and optimization of the pipelines. Initially, we started with exhaustive search, and as we gained more insights, we shifted towards adding smart search which consisted of hill climbing, simulated annealing and evolutionary algorithm.

3.1.1 Preprocessors and classifiers

In this section, we have mentioned different types of preprocessors and classifiers used during the experiments. Table 3.1 and 3.2 shows the list of numerical and categorical preprocessors respectively. And Table 3.4 shows the classifiers used during the experimentation.

On receiving the data, we divide its features into numerical columns where numerical preprocessors are applied and categorical columns, where categorical preprocessors are applied. And as mentioned in Table 3.3, we have TruncatedSVD and SelectKBest preprocessors, which acts as common preprocessors for the pipelines.

Preprocessor	Description
StandardScaler	Scales numerical features to have zero mean and unit variance.
SimpleImputer	Handles missing data by filling in missing values using various strategies.
MaxAbsScaler	Scales each feature by its maximum absolute value.
QuantileTransformer	Transforms numerical features to follow a Gaussian distribution using quantiles.
Normalizer	Scales each sample to have unit norm (L2 norm).
KBinsDiscretizer	Discretizes numerical features into bins based on specified strategies.

Table 3.1 List of Numerical Preprocessors

Preprocessor	Description
SimpleImputer	Handles missing data by filling in missing values using various strategies.
OneHotEncoder	Converts categorical features into binary vectors (one-hot encoding).
OrdinalEncoder	Encodes categorical features as ordinal integers.

Table 3.2 List of Categorical Preprocessors

Preprocessor	Description
TruncatedSVD	Dimensionality reduction using TruncatedSVD.
SelectKBest	Selects top k features based on statistical tests.

Table 3.3 List of Common Preprocessors

Classifier	Description
DecisionTreeClassifier	A tree-based classifier
RandomForestClassifier	An ensemble of decision trees
LogisticRegression	A linear model for binary classification
SVC	Support Vector Classifier
Perceptron	A linear classifier
SGDClassifier	Stochastic Gradient Descent classifier
PassiveAggressiveClassifier	A linear model suitable for large-scale learning
MLPClassifier	Multi-layer Perceptron classifier

Table 3.4 List of Classifiers

3.1.2 Individual pipeline

In the early stages, we focused on creating individual pipelines. Each pipeline consisted of specific preprocessors and classifiers (as mentioned in section 3.1.1) tailored to handle either numerical or categorical data.

This division was important as it allowed us to apply the right techniques to the right kind of data. Once the pipelines were set up, we ran them across the

dataset to see which ones performed the best. Regarding the output, we ended up with a set of high-performing pipelines, each representing a potential best-fit pipeline for our dataset.

Algorithm 6 Create Pipeline

```
function CREATE_PIPELINE
    if preprocess_flag then
        Set up preprocessor for numerical and categorical columns
        Combine preprocessors into a single preprocessor
        Create pipeline with preprocessor and classifier
    else
        Create pipeline with only the classifier
    end if
    return pipeline
end function
```

3.2 Different search techniques and their performance estimation

Let's have a look at the different search processes and their performances in this section. This will give a better way to understand our evaluation process and how we implemented different algorithms in our workflow.

3.2.1 Exhaustive search

We started our experimentation with the most straightforward approach, i.e. exhaustive search. For this, we created a large number of pipelines using different combinations of numerical preprocessor, categorical preprocessor and classifiers (as seen in the table 3.1 and 3.2) using the `create_pipeline()` (Algorithm 6). This newly created pipeline then underwent `evaluate_pipeline()` (Algorithm 7) which returns the accuracy and execution time of the pipeline. And we repeat this for all the pipelines that were generated.

Algorithm 7 Evaluate Pipeline

```
function EVALUATE_PIPELINE
    Record start time
    Fit the pipeline with training data
    Record end time
    Calculate execution time
    Compute accuracy on test data
    return accuracy, execution time
end function
```

With this brute force approach, we ran every pipeline across our input dataset and selected the individual pipelines that performed the best. So this straightforward approach basically resulted in finding the best pipelines which achieved

highest accuracy among all the searches during our experimentation. And this way, we were able to figure out what combinations of preprocessors and classifiers achieves best accuracy against the input dataset as shown in Algorithm 8.

But this approach is very costly in terms of time. Since we are doing the brute force, the computation time is very high. For small datasets like iris dataset (as mentioned in Section 4.1) it took like 5 minutes, but medium datasets like digits dataset, it took 1.5 hours to produce the best performing pipelines. If the input dataset size increases, so does the computation time massively.

Algorithm 8 Pipeline Selection Algorithm

1: Dataset, Pipelines → Best Pipeline
2: Generate all possible pipeline combinations from the configurations.
3: Test each pipeline thoroughly on the dataset.
4: Select the pipeline with the highest accuracy.
5: **return** Best Pipeline

3.2.2 Hill climbing

After brute force, we wanted to implement smart search in our experiments. And so, we started with implementing hill climbing. Since we have already discussed in section 1.5.2 about hill climbing, we had to implement this algorithm in our in our pipleline to find the best possible optimal solution for the incoming dataset. We initialise the algorithm by generating a random configuration using `generate_random_configuration()` as seen in Algorithm 9. This function returns a pipeline composed of randomly generated preprocessors and a classifier. Then we pass this pipeline and our dataset to the `perform_hill_climbing()`, as seen in Algorithm 10.

Algorithm 9 Generate Random Configuration

1: **function** GENERATE_RANDOM_CONFIGURATION
2: Get available preprocessors and classifiers
3: Choose a random number of numerical preprocessors
4: Choose a random number of categorical preprocessors
5: Select configurations for numerical and categorical preprocessors randomly
6: Select a random classifier configuration
7: Format and return selected configurations
8: **end function**

In this function, we calculate the accuracy of this pipeline, called initial accuracy. Then we tweak this pipeline bit by bit to see if it improves. We do this by flipping the preprocessors (numerical or categorical) or classifier. If the tweaking produces better accuracy as stated in the line 6 of Algorithm 10, we stick with it and keep refining until all the iterations are over.

As a result, we get the pipeline which scored the best accuracy during the whole process. With this, we know which set of preprocessors and the classifier was used in building this pipeline.

Algorithm 10 Perform Hill Climbing

```
function PERFORM_HILL_CLIMBING
    Initialize current configurations for preprocessors and classifier
    Initialize variables for tracking best accuracy and pipeline
    Evaluate initial pipeline's accuracy
    Store initial accuracy and update best if it's the highest so far
    Set number of iterations for hill climbing
    for _ in iterations do
        if random decision to modify preprocessors then
            if random decision to change numerical preprocessor then
                Update numerical preprocessor configuration randomly
            else
                Update categorical preprocessor configuration randomly
            end if
        else
            Update classifier configuration randomly
        end if
        Evaluate the new pipeline's accuracy
        if new accuracy is higher than current best then
            Update best accuracy and pipeline
        end if
        Record best accuracy achieved in each iteration
    end for
    return best pipeline, best accuracy, list of best accuracies during iterations
end function
```

3.2.3 Simulated annealing

After discussing the theoretical aspect of simulated annealing in section 1.5.3, this section deals with the practical implementation of this. Here, we use the logic to integrate with our pipelines to find the best possible optimal solution for the incoming dataset.

Implementation for simulated annealing is very similar to the hill climbing, as seen in the Algorithm 11. The only difference is that it introduces the occasional steps backward to explore potentially better solutions. Just as we discussed in theory, it uses the **temperature** parameter to control randomness in accepting worse solutions and a **cooling rate** to reduce this randomness over iterations. But as seen by the results (in section 4.2.3) it is clear that its performance very much resembles hill climbing.

Algorithm 11 Perform Simulated Annealing

```
function PERFORM_SIMULATED_ANNEALING
    Initialize current configurations for preprocessors and classifier
    Initialize variables for tracking best accuracy and pipeline
    Evaluate initial pipeline's accuracy
    Store initial accuracy and update best if it's the highest so far
    Set number of iterations and initial temperature for simulated annealing
    Define cooling rate for temperature decrease
    for _ in iterations do
        if random decision to modify preprocessors then
            if random decision to change numerical preprocessor then
                Update numerical preprocessor configuration randomly
            else
                Update categorical preprocessor configuration randomly
            end if
        else
            Update classifier configuration randomly
        end if
        Evaluate the new pipeline's accuracy
        if new accuracy is higher than current best then
            Update best accuracy and pipeline
        else
            Calculate acceptance probability based on temperature and accuracy difference
            if probability condition met then
                Accept the new configuration despite lower accuracy
                Update best accuracy and pipeline
            end if
        end if
        Cool down the temperature
        Record best accuracy achieved in each iteration
    end for
    return best pipeline, best accuracy, list of best accuracies during iterations
end function
```

3.2.4 Evolutionary algorithm

Moving on to the last smart search, i.e. evolutionary algorithm. In order to implement it, we had to mimic natural selection by breeding the best pipelines to create even better ones. So the idea is to start with the population size of n individual pipelines and run the evolution process over g generations. For this, we use `initialize_population(size)` (refer to Algorithm 12. Then, we evaluate each pipeline's performance using accuracy as a fitness metric. This involved training and testing the pipelines on our dataset, storing the results along with their configurations. Algorithm 13 clearly explains how we utilised evolutionary algorithm to search for the best optimal solutions.

Algorithm 12 Initialize Population

```
function INITIALIZE_POPULATION(size)
    Initialize an empty population list
    for _ in range(size) do
        Generate random configurations for numerical preprocessors, categorical preprocessors, and classifier
        Append the generated configuration tuple to the population list
    end for
    return population
end function
```

The top-performing pipelines, determined by their accuracy scores, were selected as parents for the next generation. To create new pipelines for the next generation, we applied crossover and mutation on pairs of selected parents' configurations. After running it for g generations, we get the pipeline which scored the best accuracy during the whole process.

Algorithm 13 Evolutionary Algorithm

```
function PERFORM_EVOLUTIONARY_ALGORITHM
    Initialize a random population of pipeline configurations.
    Evaluate each pipeline's accuracy.
    Select top-performing pipelines to create the next generation.
    Combine features of selected pipelines through crossover.
    Introduce random mutations to maintain diversity.
    Form a new population from offspring and a subset of parents.
    Repeat for multiple generations or until a termination condition.
    return the best pipeline configuration found and its accuracy score.
end function
```

3.3 Code Implementation

We have hosted the entire code base on my GitHub repository [17]. For the machine-learning components, we utilised various tools from scikit-learn [18]. And lastly, the project repository includes all the instructions in order to run the experiments for testing and validation purposes smoothly.

4 Experiments

In this chapter, we start with defining all the datasets used during the experimentation (section 4.1). The first experiment was aimed at Exhaustive Search, whose main purpose was to discover the best optimal solutions in the entire search field using brute force (section 4.2.1). This was followed by experimentation done using hill climbing (section 4.2.2), simulated annealing (section 4.2.3) and evolutionary algorithm (section 4.2.4); we tried to determine which search was able to perform better and whether the results were related to the dataset in any manner. The goal of the last experiment was to run the top four most used datasets from OpenML (section 4.3).

4.1 Datasets

Throughout the experimentation, we got the opportunity to run our set of pipelines across multiple datasets numerous times. The main idea behind selecting a wide range of datasets was to notice if the performance of the experiments change on updating the size of the datasets. So for the purpose of our experimentation, we used the datasets mentioned in Table 4.1.

Dataset Name	**Size (Instances, Features)**	**Source**
iris dataset	Small (150, 4)	[19]
wine dataset	Small (178, 13)	[20]
blood transfusion dataset	Small (748, 4)	[21]
kc2 dataset	Small (522, 21)	[22]
breast cancer dataset	Medium (569, 30)	[23]
credit-g dataset	Medium (1000, 21)	[24]
kc1 dataset	Medium (2109, 21)	[25]
wilt dataset	Medium (4839, 6)	[26]
wine-quality-white	Medium (4898, 12)	[27]
digits dataset	Large (1797, 64)	[28]
MagicTelescope (magic)	Large (19020, 12)	[29]

Table 4.1 Dataset Summary

4.2 Experimentation using different searches

In this section, we will explore the results achieved by different searches. For a fair comparison, we used the *wine quality dataset* mentioned in section 4.1 for all the searches, i.e. exhaustive search, hill climbing, simulated annealing and evolutionary algorithm.

In the case of smart search techniques, we plotted the results as a comparison between a randomly generated pipeline and smart search techniques, just like we mentioned in section 3.2.

4.2.1 Exhaustive search

As previously discussed in section 3.2.1, this is the brute force search we started our experimentation with. This approach generated all possible combinations of numerical preprocessor, categorical preprocessor and classifiers listed in Table 3.1 and 3.2.

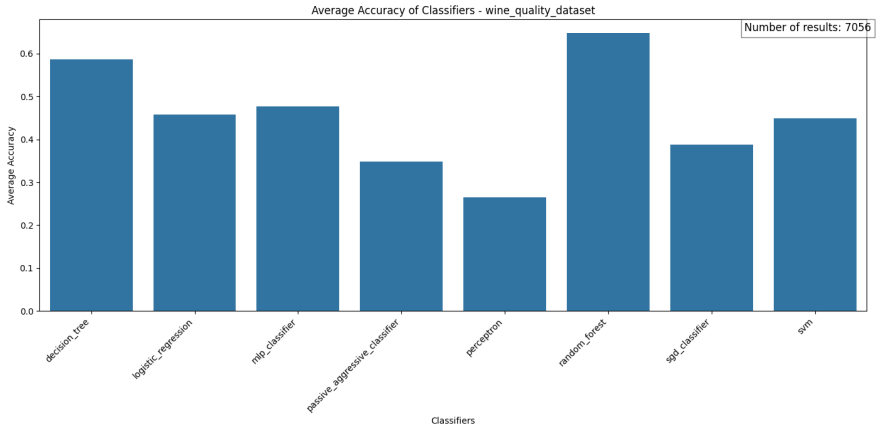

Figure 4.1 Average accuracy of classifiers achieved on wine quality dataset.

For example, during the exhaustive search, a total of **7056** pipelines were created using all the existing configurations and we ran it across the *wine quality dataset* (as mentioned in Section 4.1).

Figures 4.1 and 4.2 show the classifier's average accuracy and execution time were during experimentation. And figure 4.3 shows accuracy and execution time using a scatter plot with Kernel Density Estimate (KDE) for both the x-axis (Execution Time) and y-axis (Accuracy).

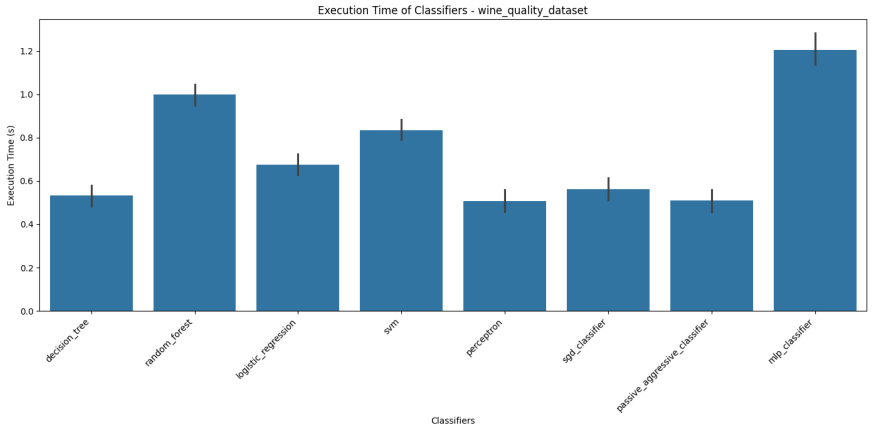

Figure 4.2 Execution time of classifiers achieved on wine quality dataset.

Next, we chose two small and two medium sized datasets and ran exhaustive search on them. We selected top 200 pipelines based to their accuracy. Then, we plotted a bar graph showing which classifiers appeared the most number of times. Random forest classifier performed the best in all the four scenarios. And quite interestingly, in both the medium datasets, random forest was the only classifier present in the top 200 best performing pipelines. Figure 4.8 provides a visual representation of these four scenarios.

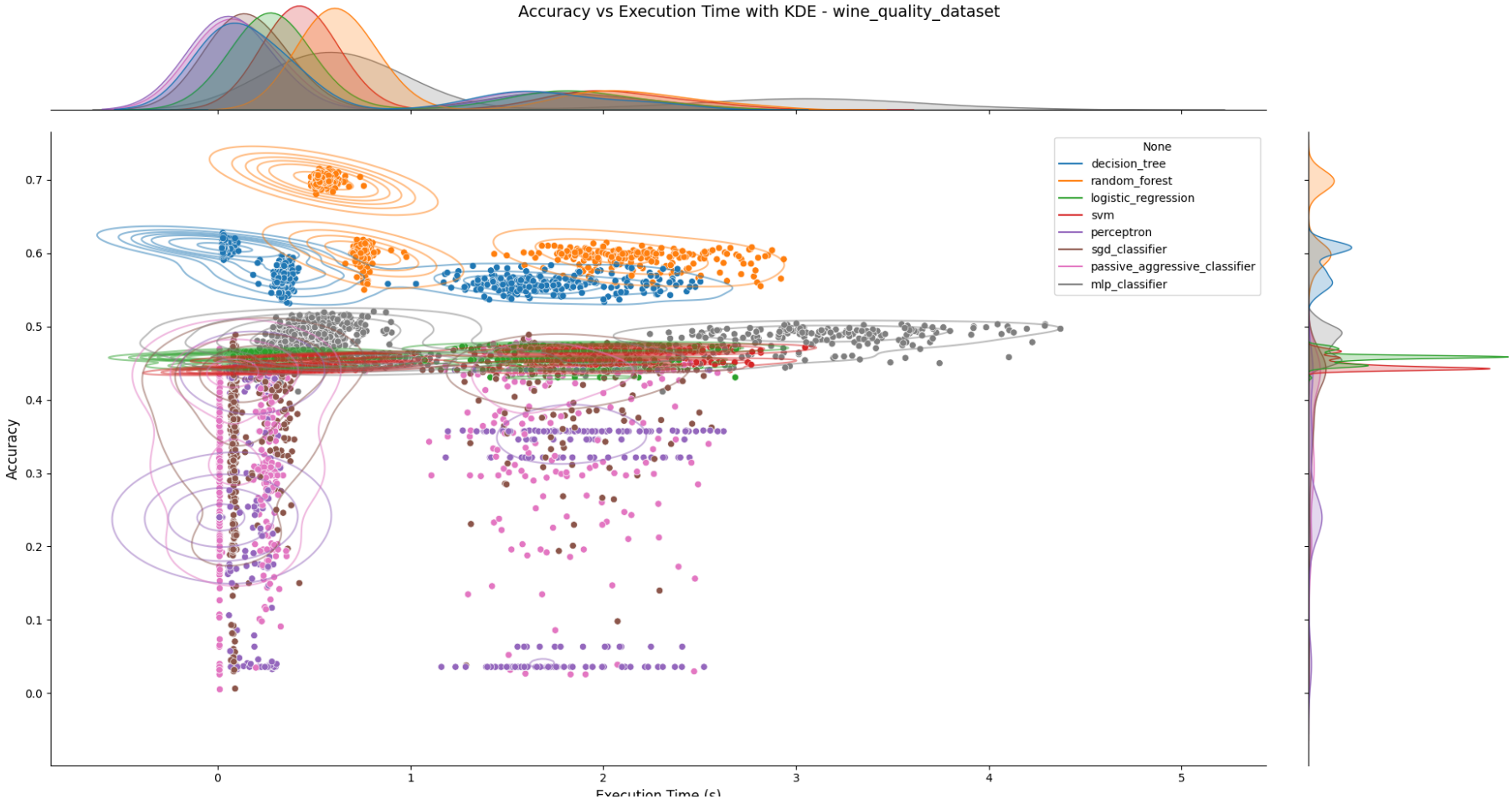

Figure 4.3 X-Axis represents the execution time of different models and Y-Axis represents the accuracy of the models. Each point on the scatter plot represents a model or method with its corresponding execution time and accuracy. The distribution of points helps to visualize the trade-off between execution time and accuracy. KDE on X-Axis shows the density of execution times and peaks in this plot indicate execution times where there are many models concentrated. Similarly, KDE on Y-Axis shows the density of accuracies and peaks here indicate accuracy levels where many models are concentrated.

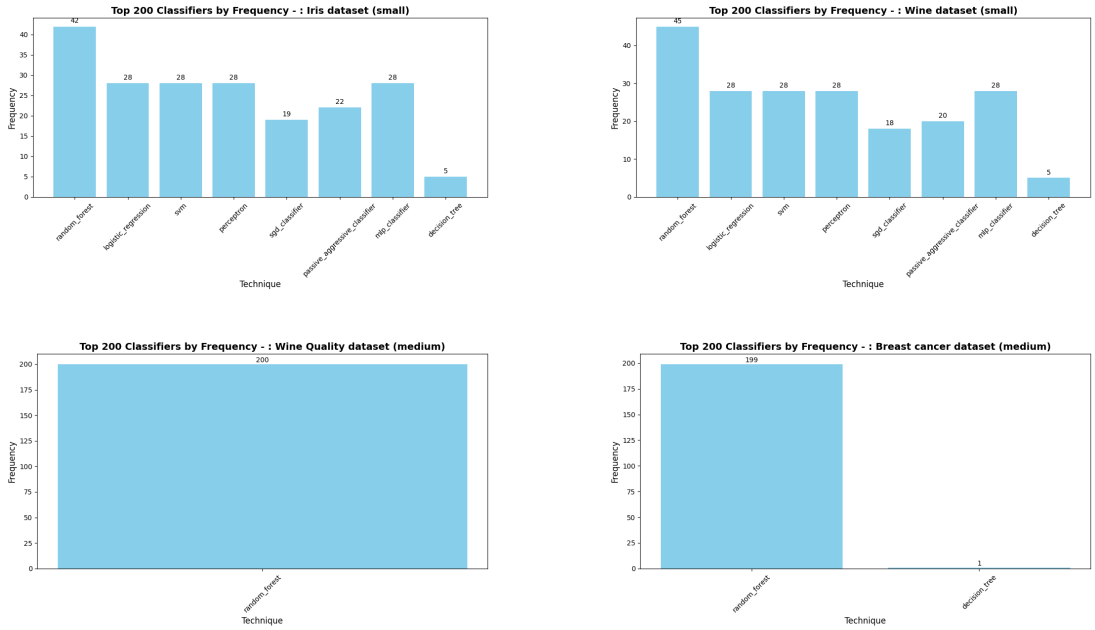

Figure 4.8 Results of top 200 best performing pipelines on four different datasets.

4.2.2 Hill climbing

As mentioned in section 3.2.2, the next approach we used was hill climbing. After generating a random pipeline to start with using Algorithm 9, we ran the Algorithm 10. This ran for 1000 iterations and gave the pipeline with the best accuracy as the output. Figure 4.9 shows the learning curve of hill climbing for these 1000 iterations.

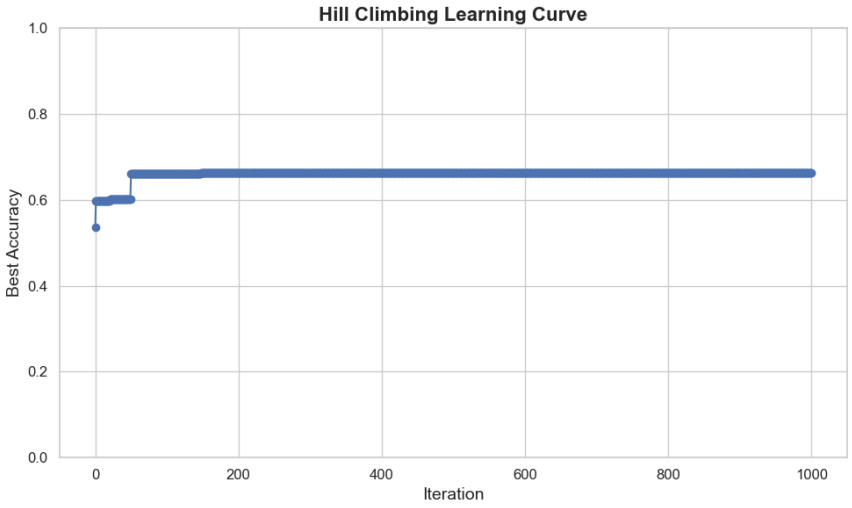

Figure 4.9 Learning curve of hill climbing search algorithm on 1000 iterations.

4.2.3 Simulated annealing

Very similar to hill climbing in the section 4.2.2, the next approach we used was simulated annealing. After initialising the experiment with a random pipeline using Algorithm 9, we ran Algorithm 11.

Just like hill climbing, we ran this for 1000 iterations as well, but we set the temperature to 1 and cooling rate to 0.95. And as a result, we received a pipeline with the best accuracy as the final output. Figure 4.10 shows the learning curve of simulated annealing for these 1000 iterations.

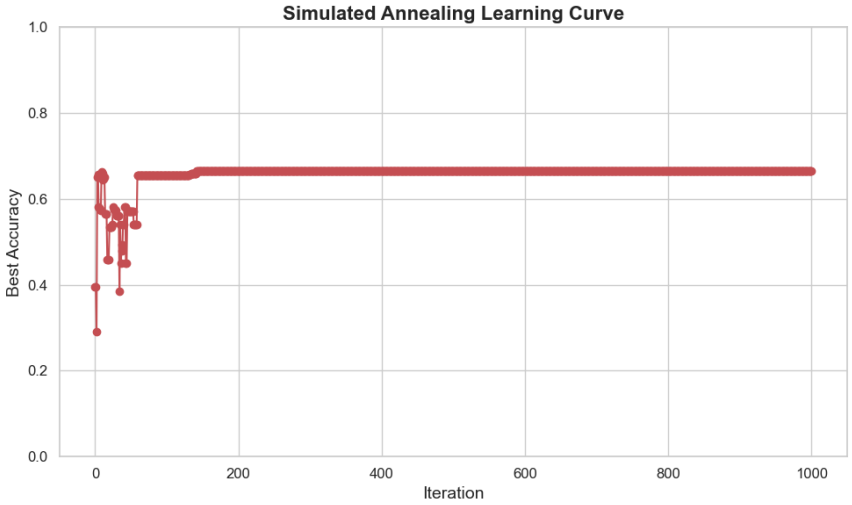

Figure 4.10 Learning curve of simulated annealing search algorithm on 1000 iterations.

4.2.4 Evolutionary algorithm

After this, we moved on to the last approach, which is searching for the best optimization solution using the Evolutionary algorithm. Here, we started with a population size of 20 pipelines (called individuals) and ran the evolution process for over 50 generations. 50 generations of 20 individuals corresponds to 1000 iterations of fitness. Thus, it is comparable to hill climbing and simulated annealing experiments. Figure 4.11 shows the learning curve of hill climbing for these 1000 iterations.

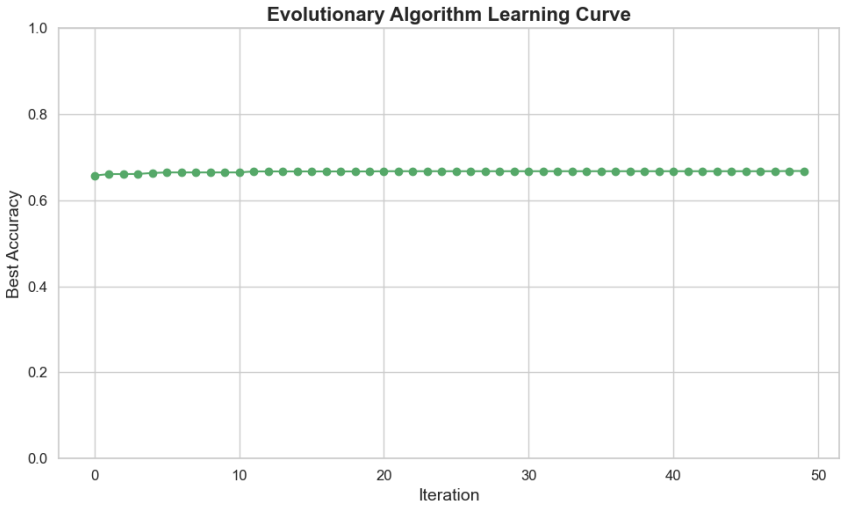

Figure 4.11 Learning curve of evolutionary algorithm search algorithm on 50 generations.

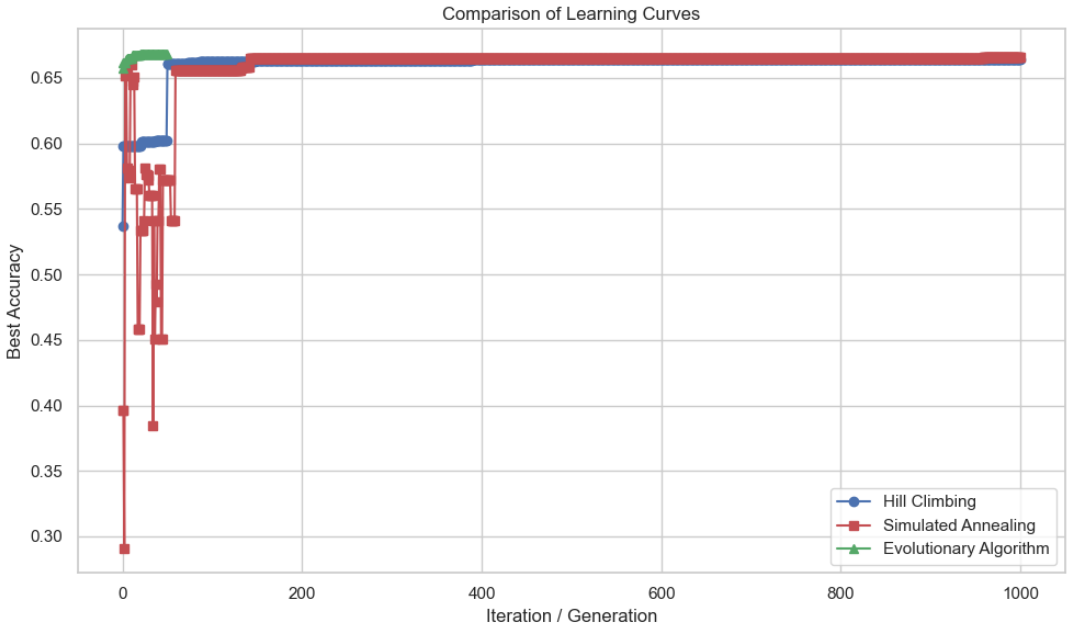

Figure 4.12 Learning curve of all search algorithms on 1000 iterations. Here, 50 generations of evolutionary algorithm corresponds to 1000 iterations of hill climbing and simulated annealing in terms of number of fitness evaluations.

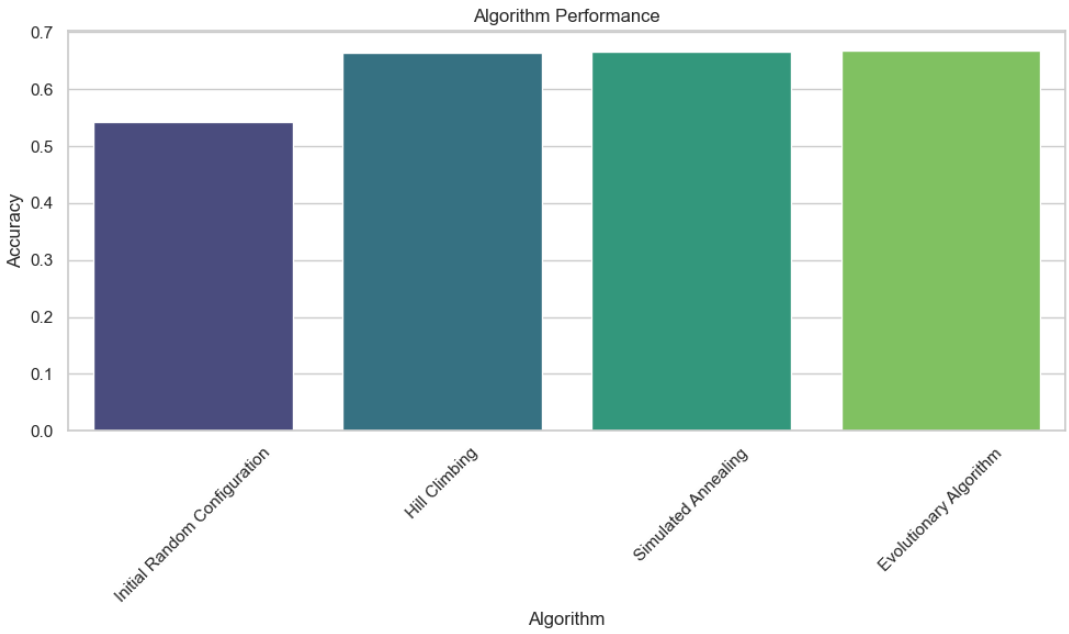

Figure 4.13 Accuracy of the resulting pipeline for each algorithm.

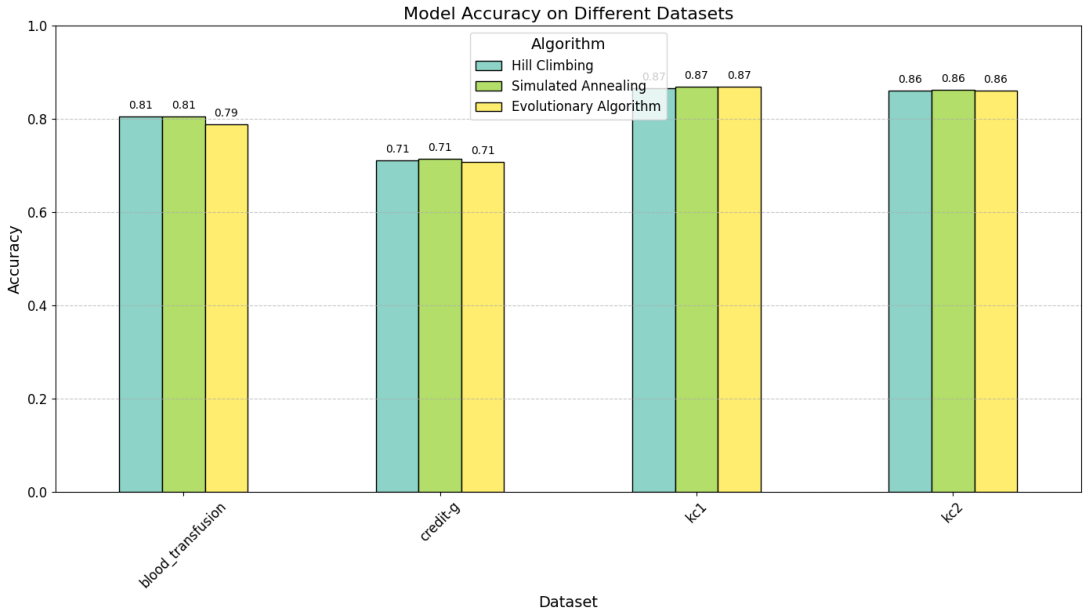

Figure 4.14 Accuracy achieved by all smart searches on four OpenML-CC18 benchmark datasets.

Figure 4.12 shows how the learning curves of hill climbing, simulated annealing and evolutionary algorithm looks together after 1000 iterations.

And Figure 4.13 shows the accuracy of the best optimal solution from each of the smart search algorithm. We added a randomly generated pipeline to compete with the other three algorithms. As we can see in the plot, all the three smart searches achieved similar results. On the other hand, the randomly generated pipeline didn't perform well.

4.3 Experimentation using OpenML's datasets

After going though the other datasets (as mentioned in section 4.1), we came across OpenML's public dataset OpenML-CC18. It contains a wide variety of 72 datasets. Although the datasets can be accessed after a free registration and using your own API key, we downloaded the four most used datasets among all the 72 mentioned over there [30].

In the figure 4.14, we can see the performance of all the three smart searches (as mentioned in section 3.2) when we ran it across the top four most used dataset from OpenML-CC18.

Conclusion

The main idea behind our experimentation was to design an AutoML system that takes in the dataset as an input and performs supervised learning optimization. It achieves this by forming a set of pipelines whose configuration is composed of any length of numerical preprocessors, categorical preprocessors and a classifier. It could be a single randomly generated pipeline, or a set of pipelines generated using all the possible combinations of preprocessors and classifiers. Basically, this depended upon the kind of search we were performing in order to find the best optimal solution.

For the exhaustive search, the results were phenomenal and we were receiving the best set of pipelines for a given dataset. But on the downside, this process took a lot of time because of its brute force approach. If we talk about smart searches, the hill climbing and simulated annealing always received almost similar accuracy in their best performing pipeline configuration, be it a small or large dataset. And in the case of evolutionary algorithms, it always used to converge the fastest and provided the best result out of all the smart searches.

Regarding the OpenML's top four most used datasets, we saw that all the smart searches received almost similar accuracy after running for 1000 iterations.

Regarding the future scope of this experimentation, we can integrate many new ideas. One such promising concept was hyperparameter tuning. The mutations of hyperparameters can significantly improve the results. The idea was to input different values of the parameters for all the preprocessors and classifiers. This could uplift the performance of the best pipelines even further. This is because we would be using the best parameter settings for the particular configuration and the input dataset. Another extension is to implement the crossover rate of the evolutionary algorithm between 0.5 to 0.8. This change would enhance the resulting pipelines by better mimicking the process of natural selection, making the overall evolutionary algorithm more realistic.

Bibliography

1. Flach, Peter. *Machine Learning: The Art and Science of Algorithms That Make Sense of Data* [Cambridge University Press, New York, NY, USA, 2012]. [N.d.]. ISBN 1107422221, 9781107422223.
2. Jain, Abhishek. *A Comprehensive Guide to Performance Metrics in Machine Learning* [Medium, https://medium.com/@abhishekjainindore24/a-comprehensive-guide-to-performance-metrics-in-machine-learning-4ae5bd8208ce]. 2024. Accessed: 2024-04-13.
3. PyTorch. *Ensembling multiple models together* [https://pytorch.org/tutorials/intermediate/ensembling.html]. Accessed: 2024-06-04.
4. Science, Towards Data. *Ensemble Methods in Machine Learning: What Are They and Why Use Them?* [https://towardsdatascience.com/ensemble-methods-in-machine-learning-what-are-they-and-why-use-them-68ec3f9fef5f]. Accessed: 2023-09-12.
5. Neptune.ai. *Ensemble Learning Guide* [https://neptune.ai/blog/ensemble-learning-guide]. Accessed: 2024-05-22.
6. Wolpert, David H.; Macready, William G. No Free Lunch Theorems for Optimization. *IEEE Transactions on Evolutionary Computation*. 1997, vol. 1, no. 1, pp. 67–82. Available from doi: 10.1109/4235.585893.
7. Devoteam. *Is AutoML the Future of Data Science?* [https://www.devoteam.com/expert-view/is-automl-the-future-of-data-science/]. Accessed: 2024-07-02.
8. Somi, Regan. *A Guide to Machine Learning Workflow* [Medium, https://medium.com/@regansomi/a-guide-to-machine-learning-workflow-9a4302b8787e]. 2021. Accessed: 2024-07-18.
9. Run.AI. *Machine Learning Workflows* [https://www.run.ai/guides/machine-learning-engineering/machine-learning-workflow]. Accessed: 2024-05-03.
10. Thornton, Chris; Hutter, Frank; Hoos, Holger H.; Leyton-Brown, Kevin. AutoWEKA: Automated Selection and Hyper-Parameter Optimization of Classification Algorithms. *CoRR*. 2012, vol. abs/1208.3719. Available also from: http://arxiv.org/abs/1208.3719.
11. ScholarHat. *Model Selection for Machine Learning* [https://www.scholarhat.com/tutorial/machinelearning/model-selection-for-machine-learning]. Accessed: 2024-07-02.
12. AlmaBetter. *Local Search Algorithm in Artificial Intelligence* [https://www.almabetter.com/bytes/tutorials/artificial-intelligence/local-search-algorithm-in-artificial-intelligence]. Accessed: 2024-07-11.
13. Russell, Stuart; Norvig, Peter. *Artificial Intelligence: A Modern Approach*. 3rd. Upper Saddle River, NJ, USA: Prentice Hall Press, 2009. isbn 0136042597, isbn 9780136042594.

14. Darwin, Charles. *On the Origin of Species by Means of Natural Selection, or, The Preservation of Favoured Races in the Struggle for Life*. London: Murray, 1859.

15. Le, Trang T.; Fu, Weixuan; Moore, Jason H. Scaling Tree-based Automated Machine Learning to Biomedical Big Data with a Dataset Selector. *bioRxiv*. 2018. Available from doi: 10.1101/502484.

16. Elsken, Thomas; Metzen, Jan Hendrik; Hutter, Frank. Neural Architecture Search: A Survey. *arXiv e-prints*. 2018. Available from arXiv: 1808.05377 [cs.LG].

17. Sharma, Rajat. *Evolutionary techniques in AutoML* [https://github.com/silver-times/autoML_pipeline]. 2024. Accessed 2024-07-18.

18. Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V.; Vanderplas, J.; Passos, A.; Cournapeau, D.; Brucher, M.; Perrot, M.; Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*. 2011, vol. 12, pp. 2825–2830.

19. Fisher, R.A. *Iris Dataset* [https://archive.ics.uci.edu/ml/datasets/iris]. 1936. Small dataset used for classification of iris species into setosa, versicolor, and virginica classes based on four features. Cite: R.A. Fisher (1936) "The use of multiple measurements in taxonomic problems" Annals of Eugenics.

20. Forina, M. et al. *Wine Dataset* [https://archive.ics.uci.edu/ml/datasets/wine]. UCI Machine Learning Repository. Small dataset used for classifying wines into three classes based on chemical analysis of 13 features. Cite: Forina, M. et al. (1991) "PARVUS: An Extendible Package for Data Exploration, Classification and Correlation." Institute of Pharmaceutical and Food Analysis and Technologies.

21. Yeh, Prof. I-Cheng. *Blood Transfusion Service Center Dataset* [https://archive.ics.uci.edu/ml/datasets/Blood+Transfusion+Service+Center]. UCI. Dataset from the Blood Transfusion Service Center in Hsin-Chu City, Taiwan, used for predicting blood donation behavior. Attributes include recency, frequency, monetary value, and time since first donation. Cite: Yeh, I-Cheng, Yang, King-Jang, and Ting, Tao-Ming, "Knowledge discovery on RFM model using Bernoulli sequence", Expert Systems with Applications, 2008.

22. Mike Chapman, NASA. *KC2 Dataset* [https://openscience.us/repo/defect/mccabehalsted/kc2.html]. 2004. Dataset from the NASA Metrics Data Program, used for software defect prediction based on McCabe and Halstead features of source code.

23. Wolberg, Dr. William H.; Street, W. Nick; Mangasarian, Olvi L. *Breast Cancer Dataset* [https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)]. UCI Machine Learning Repository. Medium-sized dataset used for classifying tumors as malignant or benign based on 30 features extracted from breast images. Cite: Dua, D. and Karra Taniskidou, E. (2017) "UCI Machine Learning Repository.".

24. Hofmann, Dr. Hans. *German Credit Dataset* [https://archive.ics.uci.edu/ml/datasets/Statlog+(German+Credit+Data)]. 1994. UCI Machine Learning Repository.

25. Mike Chapman, NASA. *KC1 Dataset* [https://openscience.us/repo/defect/mccabehalsted/kc1.html]. 2004. Dataset from the NASA Metrics Data Program, used for software defect prediction in storage management software. Includes McCabe and Halstead features to assess software quality.

26. Johnson, Brian Alan; Tateishi, Ryutaro; Hoan, Nguyen Thanh. A Hybrid Pansharpening Approach and Multiscale Object-Based Image Analysis for Mapping Diseased Pine and Oak Trees. *International Journal of Remote Sensing*. 2013, vol. 34, no. 20, pp. 6969–6982. Available from doi: 10.1080/01431161.2013.810825.

27. Cortez, Paulo; Cerdeira, António; Almeida, Fernando; Matos, Telmo; Reis, José. Modeling Wine Preferences by Data Mining from Physicochemical Properties. *Decision Support Systems*. 2009, vol. 47, no. 4, pp. 547–553. issn 0167-9236. Available from doi: 10.1016/j.dss.2009.05.016.

28. Alpaydin, E.; Kaynak, C. *Digits Dataset* [https://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits]. UCI Machine Learning Repository. Large dataset used for recognizing handwritten digits (0-9) based on 64 features representing pixel values. Cite: E. Alpaydin, C. Kaynak (1998) "Cascading Classifiers, Kybernetika.".

29. Bock, R.K.; Chilingarian, A.; Gaug, M.; Hakl, F.; Hengstebeck, T.; Jiřina, M.; Klaschka, J.; Kotrč, E.; Savický, P.; Towers, S.; Vaiciulis, A.; Wittek, W. Methods for Multidimensional Event Classification: A Case Study Using Images from a Cherenkov Gamma-ray Telescope. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*. 2004, vol. 516, no. 2, pp. 511–528. issn 0168-9002. Available from doi: 10.1016/j.nima.2003.08.157.

30. OpenML. *OpenML-CC18 Documentation* [https://docs.openml.org/benchmark/#openml-cc18]. 2019. Accessed 2024-03-05.

List of Figures

List of Tables

List of Algorithms