FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Gabriela Suchopárová

Evolutionary optimization of machine learning workflows

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis:	Mgr. Roman Neruda, CSc.
Study programme:	Computer Science
Study branch:	General Computer Science

Prague 2019

My sincere thanks to Mgr. Roman Neruda, CSc. for his guidance and patience. I would also like to thank my family and friends for supporting me throughout my studies.

Title: Evolutionary optimization of machine learning workflows

Author: Gabriela Suchopárová

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Roman Neruda, CSc., Department of Theoretical Computer Science and Mathematical Logic

Abstract: This work deals with automated machine learning (AutoML), which is a field that aims to automatize the process of model selection for a given machine learning problem. We have developed a system that, for a given supervised learning task represented by a dataset, finds a suitable pipeline — combination of machine learning, ensembles and preprocessing methods. For the search we designed a special instance of the developmental genetic programming which enables us to encode directed acyclic graph pipelines into a tree representation. The system is implemented in the Python programming language and operates on top of the scikit-learn library. The performance of our solution was tested on 72 datasets of the OpenML-CC18 benchmark with very good results.

Keywords: Machine learning Evolutionary computing Meta-learning Workflows

Contents

Introduction

Over the last few years we have witnessed an enormous success of artificial intelligence. Thanks to advances in machine learning, a great variety of applications emerged, ranging from automatized analyses of documents, through image recognition to music recommendation. The artificial intelligence is also present in medical research and has initiated the Industry 4.0. As so, it already influences many parts of our everyday lives.

Most of these examples are however complex systems developed by large teams of data scientists and machine learning experts. Every machine learning algorithm depends on a set of hyperparameters which influence the learning process. Because of that, to obtain good results they must be carefully chosen, otherwise the algorithm fails to capture important relations in data. The design of a machine learning model is therefore a time-consuming process which requires lots of fine-tuning often done by trial and error.

The character of the input data matters as well, sometimes a series of preprocessing must be done before it is possible to analyse it. Moreover, powerful machine learning methods, like neural networks, are usually very computationally demanding, which further complicates the process.

As such, for small teams and non-experts it may be very difficult (or even impossible) to develop a working machine learning system. Due to limitations in time, budget or knowledge, instead of choosing the model best suited for their purpose, they resort to simple methods with default settings. In some cases, this approach may not even produce any satisfactory results.

The automated machine learning (AutoML) is a recent area of research that aims to overcome these problems. It is a field that opens up the world of machine learning to more people and facilitates the work of machine learning experts as well. It encompasses systems that automatize a part of a machine learning workflow, which is the iterative process of solving the given problem. Some methods optimize the whole workflow, thus enabling the user to obtain decent results even without any adjustments or data preprocessing. Other are aimed at facilitating the model selection by proposing some models which can be further optimized by experts.

Nevertheless, the results of AutoML systems are not always better than hand-designed models. The reason is that in order to limit the running time without impairing the result, compromises must be made. Existing systems usually limit the architecture of the model or operate only with a small number of methods. Although this approach works relatively well, it is not possible to discover novel structures.

The goal of this work is to solve some of these limitations. We want to design an extensive system that enables to create more complex model architectures while also taking into account simpler structures with equally good results. To do so, we explore existing systems and propose a a more general representation. A machine learning model has the structure of a pipeline — a directed acyclic graph with a logical order of methods. As it is hard to optimize this kind of structures directly, we provide an encoding that converts an arbitrary pipeline to a tree representation. This enables us to use the developmental genetic programming, which is a heuristic optimization method that operates on tree structures.

We will propose an evolutionary algorithm based on the developmental genetic programming. The algorithm should find the most suitable model architecture for a given problem as well as optimize the hyperparameters of every method contained in the structure. As we design more complex pipelines which may induce some overhead in evaluation time, we also introduce a performance estimation method based on sampling which should reduce the overall running time of the algorithm.

The whole system will be evaluated on a suite of publicly available datasets — namely the OpenML-CC18 benchmark. This benchmark suite is tied with the OpenML machine learning environment, which is a project that provides data of results of previous runs on machine learning tasks. As such, we will be able to compare our system with reference results.

This thesis has the following structure. Chapter 1 provides the necessary theoretical background of this work. We first define the machine learning in general (section 1.1), along with the process of model evaluation. We will also describe the ensembles — combinations of models — in detail. Then we present an overview of problems which are solved by AutoML systems (chapter 1.2). In section 1.3, we define the evolutionary algorithms in a general way, as well as some concepts relevant to our work, namely the multi-objective optimization. Next, we continue with the description of a subfield of evolutionary algorithms, which is the genetic programming. Specifically, we present the developmental genetic programming which we will use in our work (section 1.4). Chapter 2 is dedicated to existing AutoML systems, which are also compared with our approach. In chapter 3 we present our solution in detail. The original evolutionary algorithm for the search of ML pipelines is presented. We describe in what way we use the developmental genetic programming and the character of the tree encoding (section 3.1). Moreover, we discuss the performance estimation strategy used in our work (section3.2). Finally, section 3.3 lists some implementation details of our system along with utilized libraries. In chapter 4 we will examine the general properties of our system and assess its performance. We first carry out an experiment which compares the different sampling strategies proposed in previous chapters (section 4.1). Next, we test the influence of genetic operators on the quality of the solution (section 4.2). Finally, we evaluate our system on the OpenmML-CC18 benchmark and present the results in section 4.3.

1. Preliminaries

In this section we present the theoretical background of this work. We first define the general concept of machine learning, then describe a modern area of machine learning research — the automatic machine learning, or AutoML. Finally, we dedicate two sections to the heuristic optimization method — evolutionary computation and its subfield, genetic programming.

1.1 Machine learning

The field of machine learning (ML) encompasses a broad range of algorithms and statistical methods for data processing. In his book on machine learning, Flach provides the following general definition:

> Machine learning is the systematic study of algorithms and systems that improve their knowledge or performance with experience [Flach, 2012].

The knowledge of a system is gained through learning from *experience*. This procedure is referred to as the *training* phase. In this process, the ML system adjusts its parameters according to the nature of training data. The result of the process is a prediction function which depends on learned parameters. By applying this function on previously unseen data, denominated as the *testing* data, we obtain the output of the algorithm. The result is then evaluated to determine the performance of the method [Bishop, 2006, p. 2]. The character of the learning process varies with different machine learning problems. There are three main classes of ML tasks: *supervised*, *unsupervised* and *reinforcement* learning.

In case of the supervised learning, the training data is a set of labelled examples (also called instances) and the task is to predict labels (also called the target) of previously unseen data. The field is further subdivided into two groups. If the labels are elements of a finite number of discrete categories, the problem is called *classification*. Otherwise, if a label may be any real number, the problem is called *regression*. In contrast to supervised learning, in unsupervised learning the training data is unlabelled. The key task is then to divide the data into groups of similar examples. The task of reinforcement learning is to find suitable actions as to maximize a reward. The training data is in this case a history of previous actions and corresponding rewards [Bishop, 2006, p. 3].

In all ML tasks, the actual learning process is influenced by the *hyperparameters* of the algorithm. Not to be confused with the internal parameters which are set during the learning process, the hyperparameters are the actual settings of the algorithm. Often, the hyperparameters largely influence the success of a particular method on a specific machine learning task.

In the following sections we elaborate on some important concepts of machine learning. We will focus on supervised learning, as it is the subject of our work.

1.1.1 Performance of machine learning models

In this section we describe the general terms related to performance evaluation of machine learning models. We first present a formal definition of supervised learning [Russell and Norvig, 2009, Mitchell, 1997].

Definition 1 (Supervised learning)**.** *Let $X \subset \widetilde{X}$ be a set of examples (or also* feature vector*) from an unknown space and $y \subset \widetilde{y}$ a set of corresponding labels. Moreover, let $f : \widetilde{X} \to \widetilde{y}$ be an unknown function that satisfies $f(X) = y$. Then the task of* supervised learning *is to find a hypothesis function h which approximates the function f by* learning *from the training set (X, y).*

Suppose we have a model which learned a particular hypothesis h and we want to determine whether it approximates f well. In other words, we want to measure the difference between the output of the model and the real value of the target — the *error* of the model. There are several types of metrics that can be used as error functions. In case of classification, the simplest metric is the *error rate* which is defined as the proportion of incorrectly classified instances — examples x for which $h(x) \neq y$. A related term is the *predictive accuracy*, which is defined as the proportion of correctly classified examples [Flach, 2012, p. 54].

The performance of a model is thus tied with a small error rate. However, it is important to note that good performance on training data does not ensure just as good performance on new data. Sometimes the model performs exceptionally well on training data, but fares much worse on testing data. This behaviour is called *overfitting*, and it usually occurs when unnecessarily subtle details of the data are learned. The opposite concept is called *generalisation*, which is the ability to perform well on different types of testing data.

A related term is the so-called *bias-variance dilemma*, which is a phenomenon closely related to model selection. The bias and variance of a model are defined as follows [Casella and Berger, 2001, p.59, 356].

Definition 2 (Bias and variance)**.** *Bias of a model is the value $E(h_t(X)) - y$, where h_t is a hypothesis learned from a particular training set t, and X and y is the set used for evaluation. Variance of a model is defined as $E[(h_t(X) - y)^2]$.*

In practice, a model with few parameters will make some errors even with sufficient training data, thus introducing a bias from the correct output. On the other hand, by increasing the number of model parameters, it will highly depend on training data. Then, with small changes in data there will be a high variance in the output. A model with both small bias and small variance can rarely be found, hence it is often necessary to choose the side that is less harmful to the particular task. Also, a choice of a particular model can introduce an implicit bias. For example, some models are linear, while other models consider that the data is of a special distribution. As so, if the input data does not match the assumptions, errors be made.

Another option how to approach the bias-variance dilemma is to use some of the ensemble methods described in section 1.1.2. For example, bagging is a method of variance reduction, while boosting noticeably reduces the bias [Flach, 2012, p. 93–94, 338].

The task of finding a model which performs well on a particular problem instance is by no means straightforward. In practice, it is usually solved by iterative selection of model settings. As described in section 1.2, there are systems which automatize a part of the process. Nevertheless, even these methods must take into account the problems presented in this section.

1.1.2 Model ensembles

Model ensembles are powerful learning techniques that combine simpler models to achieve better results. They are widely used in practice, specific examples can be found in the review of Rokach [2009].

The rationale behind ensembles is also of theoretical character, namely from statistics and from computational learning theory. In statistics, a general idea is to average measurements to get more stable and reliable results. Here, the models are trained on data samples or feature subsets and the results are then combined into a final hypothesis [Flach, 2012, p. 330]. The computational learning theory defines the term *learnability*, which describes whether a model outputs a correct hypothesis by learning on random sets of instances from a unknown distribution. A *strong learner* is a model which outputs most of the time a correct hypothesis. It is not required that it always produces a hypothesis with error equal to zero, as the set of chosen instances might be atypical or not representative enough. Similarly, a *weak learner* is then a model which outputs most of the time a hypothesis, which is slightly better than random guessing (i.e. has a success rate over 0.5). A more detailed elaboration of the learnability theory is beyond the scope of this work and can be found in books of [Flach, 2012] and [Mitchell, 1997].

The assumption of strong learnability may appear to be quite strict when compared to the weak learnability, as the model must output a correct hypothesis on almost all example sets. However, Schapire [1990] proved that a model is weakly learnable if and only if it is strongly learnable. This was proven in a constructive manner by iteratively correcting the errors of the hypotheses, thus *boosting* the model. The boosting method has directly inspired one of the most successful ensemble methods — the award winning AdaBoost [Freund and Schapire, 1996, 1997].

In the following paragraphs, we will present some of the most used ensemble methods.

Bagging *Bootstrap aggregating*, usually abbreviated to bagging, is a highly effective ensemble method. First, n samples are independently taken from the original dataset with replacement. This is referred to as *bootstrapping*. Then, we use the samples to train an ensemble of n different models. It can be used to generate predictions, which are then *aggregated* by voting or averaging into a final prediction.

This method takes advantage of the statistical stability described at the beginning of this section. As the examples are drawn with replacement, there will be some instances missing in every sample. Thus, we introduce diversity between the ensemble models [Flach, 2012, 331].

Boosting The above-mentioned boosting technique uses a different approach to model combining. Before the learning process starts, we add weights to the training examples (the base-learner must support weighting). Then, we learn the model on the modified training set, which produces a set of misclassified instances along with the (weighted) training error. We then adjust the weights in such a way that weights of the correctly classified examples decrease and those of the incorrectly classified examples increase. Therefore, when we continue and learn a new model on the data with changed weights, it will concentrate more on the problematic instances.

The algorithm stops after a fixed number of iterations or when the weighted training error increases over 0.5 — which is when the algorithm stops improving. The resulting prediction is again an average of all model predictions, but with putting more weight on models with a lower training error [Flach, 2012, 335].

1.2 AutoML

When using machine learning in practice, it is not always evident which model is suitable for a particular problem. As a consequence of the 'no free lunch' theorem [Wolpert and Macready, 1997], it is not possible for a single ML algorithm to outperform all other methods for an arbitrary problem instance. As such, model selection and hyperparameter choice must often be performed in order to obtain a satisfactory result. Along with this, it is necessary to use more complex models for some ML problems. For example, we may use some of the ensemble methods which reduce bias or variance (section 1.1.2). Also, the data typically requires some preprocessing before it can be passed to a ML algorithm, hence the model becomes even more complex. All these aforementioned tasks require human expertise, and they are usually time-consuming. The automated machine learning — AutoML — aims to automatize the process and make machine learning available even to non-experts [Zöller and Huber, 2019].

There seems to be no generally accepted definition of AutoML as for now. According to Yao et al. [2018], "*AutoML attempts to construct machine learning programs . . . without human assistance and within limited computational budgets.*" Informally, AutoML encompasses methods that automatize a part of a machine learning workflow. Examples of existing AutoML systems are presented in Chapter 2.

1.2.1 Machine learning workflows

A machine learning workflow is the process of solving a particular machine learning problem. In literature, it is also labelled as a 'machine learning pipeline' Yao et al. [2018]. However, in the context of AutoML it may not be a suitable term, as a pipeline is a set of methods logically arranged into a directed acyclic graph (DAG), but the workflow itself is an iterative process, as can be seen in Figure 1.1. The term *pipeline* generally denotes complex model structures which comprise several feature preprocessing methods and/or model ensembles.

A workflow can be decomposed into well-defined steps, which are depicted in Figure 1.1. The formulation of the problem and data collection is usually performed by an expert from an unrelated field, whereas the rest is the task of a machine learning scientist. The feature engineering comprises of initial data cleaning and feature extraction. After that, a suitable machine learning model must be selected along with suitable hyperparameters. Finally, the resulting model is evaluated, most often on a set of previously unseen *validation data*.

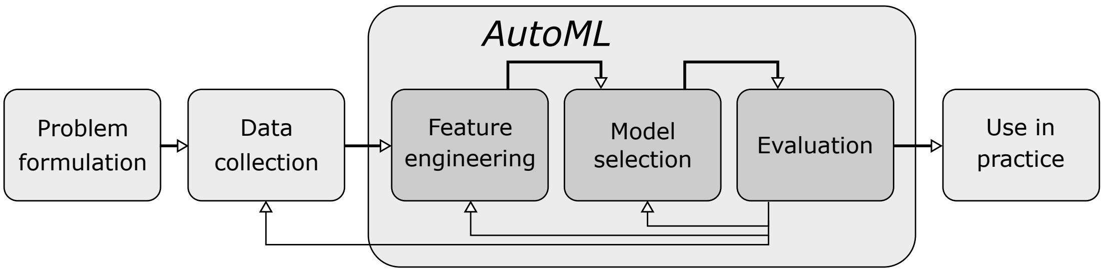


Figure 1.1: A typical machine learning workflow

The process of solving the problem is iterative; it is usually necessary to try out many different settings, as there are no algorithms that would perform well on all types of problems. There are many possibilities in what to optimize. Some models concentrate only on one part of the workflow, for example on automatic feature engineering, other handle multiple steps at once. As mentioned above, there is no general categorization of AutoML frameworks. Thus, in the following sections we present only some AutoML approaches relevant to this work. For more examples and a proposition of a general AutoML framework refer to Yao et al. [2018].

1.2.2 Hyperparameter optimization

Automated hyperparameter optimization (HPO) is the most basic, but nevertheless very important task of AutoML. In the simple case, HPO is the task of finding a hyperparameter setting of a machine learning algorithm that performs the best on a given dataset. If we optimize multiple models at once, we obtain an extended version of this problem — combined algorithm selection and hyperparameter optimization problem, or CASH for short. The task is then to select the best combination of model and hyperparameter setting [Thornton et al., 2012].

By combining multiple hyperparameter spaces, we optain a very large search space of possible models. There are several limitations which complicate the search through this space. For large models (for example in deep learning) and large datasets, the learning process and evaluation take a lot of time. Moreover, the configuration space may be quite complex, with many continuous hyperparameters or some which depend on each other. It is also necessary to avoid overfitting, which is not always obvious as the training set may not be large enough.

Some examples of frameworks which try to solve the HPO are mentioned in section 2. Various approaches are presented in the book written by Hutter et al. [2018].

1.2.3 Creating model architecture

The creation of model architecture is a problem related to the HPO. It is an important part of the design of neural networks, which is currently a very popular area of research — neural architecture search (NAS). It also needs to be considered in traditional machine learning when employing model ensembles or several feature preprocessing methos. The problem may be solved at the same time as HPO, which however introduces even more complexity in the search space.

There are many difficulties that have to be solved when designing automated architecture search methods. With increasing size of the search space the optimization time may become very long. Moreover, some complex models may score only slightly better but at the cost of a considerably longer running time. Also, evaluation cost of a single model may be very high, preventing the AutoML system from being usable in practice.

The metalearning, which uses *metaknowledge* about previous results of machine learning algorithms (as described in the following section), can be very useful in the process. It enables us to select models which performed well on similar datasets, thus starting with promising models instead of spending time on poor ones. This approach significantly speeds up the optimization process and possibly improves the result [Vanschoren, 2018]. Another option is to estimate the performance of the architecture. There are various approaches. For example, we can evaluate the model on smaller subsets of data, which however introduces some bias in the estimate. Another option is to decrease the training time, if it is possible, and extrapolate the learning curve. More on this topic can be found in a recent survey by Elsken et al. [2018].

1.2.4 Metalearning

The metalearning, also known as 'learning to learn' is a field closely related to AutoML. The subjects of study of this approach are universal properties of data and the learning process itself. Just as in case of the traditional learning — called *base-learning* in this context — metalearning improves with experience. The difference lies in the learning process. While base-learning comprises of a single run on a specific task, metalearning may include several runs on many different tasks.

One of the applications of metalearning is model recommendation. The input data is most frequently a history of previous runs. First, we need to accumulate some *metadata* to learn from. Mostly, the metadata contains algorithm performance from past runs and *metafeatures*, which are artificial features based on characteristics of input datasets. Using the *metaknowledge* learned from the metadata, we can choose an algorithm that performed well on a dataset whose metafeatures are similar to those of our dataset.

There are many other concepts used in metalearning; some are presented in more detail in following chapters, others can be found in the book by Brazdil et al. [2008] or in a recent survey by Vanschoren [2018].

1.3 Evolutionary computing

Evolutionary computing is a heuristic population-based optimization method inspired by Charles Darwin's theory of *natural selection* Darwin [1859]. In a population, individuals with the best characteristics are most likely to reproduce, thus passing the traits to the offspring. As the evolution is repeated over several generations, the most advantageous traits predominate. This phenomenon is also called 'survival of the fittest'.

In nature, every individual is defined by its *genome* — a long string of information which determines all of its characteristics. The genome of an offspring is created by combining the genomes of its parents. Therefore, it inherits some features of both of the parents.

In evolutionary computing, we maintain a population of individuals. A genome of an individual represents a particular solution to the problem which we optimize. A new individual can be created by randomly combining or altering genomes of existing individuals. The 'natural selection' is realized according to the *fitness* of the individual, which is determined by the quality of the corresponding solution. The whole process can be understood as a stochastic search through the space of all possible genomes [Engelbrecht, 2007].

1.3.1 Evolutionary algorithms in detail

A general schema of an evolutionary algorithm (EA) is presented in Algorithm 1. Every evolutionary algorithm algorithm defines a suitable *selection* method, *mutation* and/or *crossover* operators and a *fitness function*.

First, a population of individuals is initialized (line 2 of Algorithm 1). Then, using the parent selection we choose two individuals for reproduction (line 8). Using the reproduction operators, we create two new individuals, which are added to the offspring population (lines 10–14). The reproduction is repeated with other individuals until the offspring population is of the same size as the population of parents. Afterwards, we create a new population via environmental selection from the combined populations (line 16).

We continue with the process until a certain *stopping condition* is met (line 3). The result of the algorithm is the last population, which should contain individuals with a high fitness value.

In the following paragraphs, the components of the evolutionary algorithm will be described in more detail.

Fitness and objective functions The goal of an optimization process is to maximize or minimize a certain objective function. In other words, we try to find a solution for which the objective value is extremal.

In case of evolutionary algorithms, we optimize a *fitness* function, which determines the quality of an individual. It is an essential component of evolutionary algorithms, as it also determines the chance of an individual surviving and reproducing. To compute the fitness, we decode the genome of the individual to obtain the actual solution. The result is then computed by applying the objective function on the solution. As such, the fitness function and the objective function are optimized simultaneously. Most frequently the value of both of the functions is the same.

In the simple case, the fitness is tied only with one objective. To operate with multi-objective fitness, some adjustments must be made, which is presented in section 1.3.2.

Algorithm 1: Evolutionary algorithm

```
Data: population size $k$, stopping condition $c$, crossover probability $p_{cx}$
      and mutation probability $p_{mut}$
Result: evolved individuals

$P(0) \longleftarrow$ population of size $k$
while $c$ is not met:
    for individual $ind$:
        $f(ind) \longleftarrow$ compute fitness

    /* reproduction */
    for i in range($k/2$):
        /* parent selection */
        $i_1, i_2 \longleftarrow$ select two individuals from $P(n)$
        if $p_{cx}$:
            crossover($i_1, i_2$)
        if $p_{mut}$ for $k = 1, 2$:
            mutation($i_k$)

        add $i_1, i_2$ to offspring population $P_o(n)$

    /* environmental selection */
    $P(n+1) \longleftarrow$ select $k$ individuals from $P_o(n) \cup P(n)$

return $P(c)$
```

Selection The selection may be divided into two steps. The first is the *parent selection*, also called *mating selection* (line 8 of Algorithm 1), and the second is called *environmental selection* or *recombination* (line 16). Sometimes the latter is omitted, as the selection can be limited to copying all offspring to the new population.

The purpose of the parent selection is to select individuals for the mating process. Usually, it is a probabilistic process with 'better' individuals being more likely to be selected. The worse individuals have some chance to be selected as well for the sake of maintaining diversity in the population. An example of the mating selection is the *tournament selection*, where individuals 'compete' in rounds and the overall winner is selected. A round is won by an individual, if it has a greater fitness value.

The environmental selection is used to create a new population. Unlike the mating selection which is a stochastic process, replacement is usually deterministic. Individuals with a higher fitness are usually preferred as in the first type of selection, but the decision may take into account the age of the individuals. As such, it is possible to include or not to include the parents along with the offspring. A popular option is to directly choose a small number of the most successful individuals. This method is called *elitism* [Eiben and Smith, 2015]. A widely used elitist selection algorithm is the NSGA-II, which is used for multiobjective fitness functions and is described in more detail in section 1.3.2.

Crossover and mutation The crossover and mutation (also called reproduction operators [Engelbrecht, 2007]) are genetic operators that modify the structure of individuals to create new ones. Both operations are highly dependent on problem encoding, as they directly alter the structure of individuals. An example of the operators can be found in section 1.4.1. In the schema of the evolutionary algorithm, reproduction operators are applied on parents selected by the mating selection, as can be seen on lines 10 and 12.

During the crossover, the genetic information of the parents is combined into one or more children. Often, two parents are used to produce two offspring, though the counts may differ in some special types of evolutionary algorithms. Ideally, if we have two parents with different but nevertheless 'good' features, the offspring receives both of them [Eiben and Smith, 2015].

The mutation is a stochastic operator which is used to introduce diversity into the population. It randomly alters the genome in hope of creating a more fit individual. Thus, it must be applied with care, as it is also possible that some 'good' parts of the genome may become distorted in the process. This can be avoided by setting a low mutation probability or by elitism.

Stopping criteria Some commonly used stopping criteria, as listed by Engelbrecht, are for example a limit on the number of generations, an objective function threshold, or termination after no improvement is observed [Engelbrecht, 2007].

The advantage of evolutionary algorithms is such that there are potentially many different solutions present in every population. With well defined selection and fitness, the algorithm performs a multi-directional search. In comparison with other directed search methods, this proves to be a more robust approach [Michalewicz, 1996], Mitchell [1997].

1.3.2 Multi-objective optimization

In many problems, the quality of the solution depends on more than one objective function. With this, it is much harder to say whether one solution is strictly better than another. Multi-objective optimization (MOO) formally describes this class of problems. We first define the general terminology and then present MOO in evolutionary computation.

In an optimization problem, the task is to maximize or minimize the objective function $f(x)$, where x is a vector from the search space. The problem may also be restricted by constraints in the form of equalities and inequalities. In multi-objective optimization, the setting remains the same, but the objective function changes to an *objective vector* — for objective functions $f_i(x), i = 1, \dots, k$, the objective vector is defined as $f(x) = (f_1(x), f_2(x), \dots, f_k(x))$.

Although the problem setting is similar, the meaning of optimality needs to be redefined. For once, given a pair of solutions, one solution may be better than the other solution in one objective function and worse in another. For this purpose, we need the following definition.

Definition 3 (Domination)**.** *In a minimization problem, a vector x_1 dominates a vector x_2 if and only if*

- $\forall i \in \{1, \dots, k\} : f_i(x_1) \leq f_i(x_2)$, *and*
- $\exists j \in \{1, \dots, k\} : f_j(x_1) < f_j(x_2)$

As it is possible to have solutions where neither one dominates the other, it is impossible to determine one optimal solution. Hence we define the *Pareto-optimality* [Engelbrecht, 2007, p. 551-561, 569–573].

Definition 4 (Pareto-optimality)**.** *A vector x_1 is said to be* Pareto-optimal, *if there is no other vector x_2 that dominates it. The* Pareto-optimal set P^* *is the set of all non-dominated solutions. Finally, the* Pareto-optimal front *(Pareto front) is defined as* $PF^* = \{ f = (f_1(x^*), \dots f_k(x^*)) \mid x^* \in P^*) \}$.

If we use the evolutionary computation to solve multi-objective problems, the algorithm needs to be modified. As not all individuals are directly comparable, we cannot use the selection operators defined in 1.3. As such, there are various approaches on how to solve this problem, which can be divided into three groups:

- Weighted aggregation — define a single objective function as a weighted sum of sub-objectives and proceed with standard evolutionary algorithm
- Population-based non-Pareto solutions — works with the sub-objectives, but does not use the dominance
- Pareto-based solutions — tries to approximate the Pareto front

From these three groups, we describe in more detail a Pareto-based algorithm. More examples are presented in [Engelbrecht, 2007, p. 170-173]. The algorithm is called Nondominated sorting genetic algorithm (NSGA). It is a *ranking* selection, which means that individuals are sorted by their fitness values and the selection is performed with regard to the ordering.

To compute the fitness, the individuals are divided into non-dominated fronts. This is done by finding a Pareto front of a subpopulation, assigning a front number to its individuals and removing the from the subpopulation. The process is repeated with front numbers increasing until no unassigned individuals remain. Every front then obtains a dummy fitness value, where the fitness of a front $F(n)$ is better than the fitness of $F(n+1)$. Moreover, for every individual, the value is divided by a *niching* factor (while keeping the fitness inequality). For a distance metric d and a parameter σ_{share}, the niching factor is defined as

$$N(i) = \sum_{j \neq i} S(d(i,j)),$$

where S is a sharing function computed as

$$S(d(i,j)) = \begin{cases} 1 - \left(\dfrac{d(i,j)}{\sigma_{share}}\right)^2, & \text{if } d(i,j) < \sigma_{share} \\ 0, & \text{otherwise.} \end{cases} \tag{1.1}$$

This is the definition from the original article of Srinivas and Deb [1994], but it can be computed in a different manner, for example as the count of individuals closer than σ_{share} [Engelbrecht, 2007].

As the algorithm has some drawbacks, like dependence on σ_{share}, a very high computational complexity and lack of elitism, the authors of NSGA have proposed an improved variant called NSGA-II. This algorithm not only adresses the above-mentioned problems, it also outperforms other elitist algorithms [Deb et al., 2002]. An extended version — the NSGA-III — designed for the many-objective optimization (with four or more objectives) has also been developed [Deb and Jain, 2014].

1.4 Genetic programming

In this section, we present a subfield of evolutionary computing — the genetic programming (GP) — where the population is a set of computer programs. The aim of this technique is to evolve programs that provide a good solution to the given problem. There are various approaches in means of how to represent the individuals and what kind of genetic operators to use.

1.4.1 Tree-based genetic programming

The individuals are most frequently represented in the form of *syntax trees.* Inner nodes of the tree are *functions*, whereas leaves are constants and variables (*terminals*). Together, all possible functions and terminals form the *primitive set.* Every function of the set must have a well-defined arity value.

The fitness is usually computed by running the program and assessing its performance. Sometimes, the fitness is impaired by a phenomenon called the *bloat.* A bloat occurs when the GP tree grows in size without significant improvement in fitness. The result may be even worsened by parts of the tree which do not contribute in any way to the solution. As such, some countermeasures need to be adopted, for example it is possible to add a second objective to the function that minimizes the number of nodes [Poli et al., 2008, p. 24–26]. More about multi-objective optimization can be read in section 1.3.2. We also mention other bloat-reducing approaches in Chapter 3.

The initialization step of the genetic programming is very important, as there are many different ways how to design trees. Also, specialized genetic operators need to be designed. On the other hand, the selection step remains largely the same [Poli et al., 2008].

Now we describe the components of evolutionary algorithm which are specific for the genetic programming along with some important general characteristics of the GP.

Type consistency An important property of GP is the *closure*. That is, a function must be able to accept any subtree as its argument and produce a valid output if it is evaluated. This enables the reproduction operators to work properly, as otherwise a lot of invalid individuals would be generated. As a consequence, in simple GP all functions have arguments of a single type and produce output of the same type. A more complex case is the *strongly typed GP*. It constraints the primitive set in such a way that every primitive has a well-defined output type and in addition, every function defines input types of its arguments. With this we lose the closure property, so the genetic operators need to be designed in a more careful way [Poli et al., 2008, p. 21].

Initialization During the initialization, the nodes of the tree are selected from the primitive set which is provided as input to the algorithm [Koza, 1992]. As was mentioned, there are various methods of initialization. We will present two methods that are among the simplest and most used ones — *grow* and *full*.

In both cases, nodes are selected at random and inserted to the tree up to a certain height limit. The two methods differ only in the way how nodes are selected. The grow method allows to select both functions and terminals before the limit is reached; afterwards, only terminals can be inserted. The full method restricts the selection only to functions on all levels but the last one, thus generating a full tree. Leaves are then chosen from the terminal set like in the previous approach.

The drawback of the full method is that all trees are very similar. On the contrary, the grow method generates a wide range of sizes and shapes, but the number of nodes in a tree might be too small. Because of that, a method called *ramped half-and-half* is often used in practice. It combines both of the presented methods; half of the population is generated using the full method, the other one via the grow method. Also, instead of one height limit, a range of values is used to introduce more diversity. [Poli et al., 2008]

In strongly typed GP, the initialization must not violate the typing. As such, when inserting a node into the tree, we randomly choose node whose output type matches the argument type of its parent.

Genetic operators The most common type of crossover is the *subtree crossover* of two individuals. A random node — the crossover point — is selected in each individual independently. Then, subtrees corresponding to the points are exchanged between them.

Similarly, the most used mutation technique is the *subtree mutation*. Just like in subtree crossover, a mutation point is chosen randomly. Afterwards, the corresponding subtree is entirely replaced by a new randomly generated tree. Another possibility is to swap a node with a different one from the primitive set — this is called the *point mutation*.

In case of the strongly typed GP, the result of all genetic operations must be a valid individual. This means that during the crossover we can swap only these subtrees whose roots have a common output type and in subtree mutation the new tree has a root of the same output type as the previous one. In point mutation we must generate a node with the same input types and output type as the old node has. [Poli et al., 2008]

1.4.2 Developmental genetic programming

In simple GP, it is not possible to directly evolve other graph structures than trees. There are other types of GP that enable this, but individuals and genetic operators may become fairly complex. However, there is also a subfield of tree-based GP — developmental GP — which allows to indirectly evolve not only arbitrary graphs, but also much more complex real-world structures.

The key concept of the developmental genetic programming is the specific *cellular encoding* of individuals. It was first presented by Gruau [1994] as a form of evolution of neural network architecture. In cellular encoding, an *embryo* is a basic structure from which all individuals are created. The root of the GP tree corresponds to this cell, and every subtree corresponds to operations which modify specific parts of the cell. In the case of neural networks, these operation are for example node insertion or parallel/serial duplication of a part of the network. John Koza has used the developmental GP to evolve analogue circuits [Koza et al., 1998]. With this, it was even possible to achieve human-competitive result, that is, reinventing a circuit that has been previously designed for a specific purpose by hand.

In our work we are using developmental GP to evolve ML workflows, which have the structure of a directed acyclic graph. We present this in more detail in Chapter 3 along with specific examples of the encoding and of modifying operations.

2. Related work

The goal of this work is to design an AutoML system for pipeline optimization. As such, we present existing AutoML systems in this chapter to provide insight into this recently emerged field. There are several approaches, with every system focusing on a different aspect of the problem. The examples are divided into two groups, where the first one comprises of methods focusing mainly on simple models, and the second one of systems which enable more complex architectures. Finally, we compare the systems, which used a similar approach in pipeline structure representation, with the system designed by us.

Hyperparameter optimization methods The first attempt to solve the CASH problem (defined in section 2) was the AutoML system *Auto-WEKA* [Thornton et al., 2012]. It employs a Bayesian optimization method to find the optimal set of hyperparameters. To solve the model selection at the same time, an artificial hyperparameter which represents the selected model is added to the system. Other specific hyperparameters are induced from its value — for example the hyperparameters of the model itself or subestimators of ensembles. The same principle holds also for feature preprocessing methods.

The successor of Auto-WEKA, *Auto-sklearn*, improved existing systems by adding a metalearning step to the algorithm. The optimization step is the same as in Auto-WEKA, but the algorithm is *warm-started* via a history of most successful models on similar datasets. In the end, the resulting models are combined together to a specific ensemble. This is done to take advantage of all potentially good methods instead of choosing only one (not necessarily the best) and discarding the rest [Feurer et al., 2015].

Although these systems work quite well in practice, only a small number of model architectures is supported. The algorithms used in Auto-WEKA are general enough, but the number of subestimators is limited to 5, and only baselearners can be used. Auto-sklearn abandoned ensembles altogether in the optimization process, and it does not support an arbitrary ensemble in the final step of the algorithm. The reason behind this limitation is that by increasing the model complexity the configuration space becomes too large.

A similar system has been developed for the purpose of automatic tuning of neural networks — Auto-Net [Mendoza et al., 2016].

Architecture search The search for an ideal model architecture can be understood as a part of the CASH problem. This subfield of AutoML has recently gained in popularity with the demand for automated *neural architecture search* (NAS) [Elsken et al., 2018]. It also encompasses the task of pipeline design.

NAS is the process of optimization of deep neural network architectures. It may be seen as a subfield of AutoML, as along with the architecture search it may perform a hyperparameter optimization. It has already been successful in some problem classes like image recognition or object detection, where its architectures outperformed those designed by hand. In the paper by Elsken et al. [2018], the NAS methods are classified according to three dimensions — search space, search strategy and performance estimation strategy. The search space is understood as all possible neural network architectures that can be created with the automatic method. The search strategy classifies the method used for architecture optimization — some examples include the already mentioned evolutionary algorithms and Bayesian optimization, as well as the reinforcement learning or random search. Finally, the performance estimation strategies are aimed at reduction of running time. These include ‘lower fidelities estimates’, for example training on subsets of data or for fewer epochs. Another option is the ‘learning curve extrapolation’. Finally, a mate-learning like approach is to warm-start the new networks by setting the weights according to previous networks, while focusing on developing new architectures.

There are only few systems that allow unlimited pipeline sizes. The system *RECIPE* uses grammar-based GP (another subfield of GP which utilizes grammars to constraint the tree structure) to evolve pipelines, but still limits the size of the pipeline [de Sá et al., 2017]. There are two systems which support arbitrary-sized pipelines. One of them is *TPOT*, which uses genetic programming to evolve tree-based models. The root of the tree represents a single estimator whereas the branches are feature preprocessing methods [Olson et al., 2016]. The other system — *ML-Plan* — is based on hierarchical planning [Mohr et al., 2018]. Both of the methods support a chain of feature preprocessors, including feature union and stacking (ensemble methods which construct new features from subestimators).

Nevertheless, none of these methods support more complex ensemble structures. The system GP-ML developed by Křen et al. [2017] uses the strongly typed functional programming to develop an arbitrary pipeline structure. This type of GP is partly similar to the developmental GP, as every node represents a function. However, the constraints on the structure are specified in the language of functional programming, which is very different from our approach.

Comparison of our system with TPOT Our system relates most to TPOT, as it uses genetic programming to optimize the pipelines. However, instead of using the simple GP like TPOT does, we use the developmental GP which enables us to create more complex ensemble structures. For instance, TPOT uses only the stacking ensemble, which combines the output of its base-estimator with the input data, whereas in our work we use for example the voting ensemble, which combines the output of several base-estimators. However, the main difference is that in our system, the pipelines are composed from a feature preprocessor chain and of an estimator, which may be in fact another pipeline with the same structure. In TPOT, there is also a feature preprocessor chain, but at the end of the pipeline, there must be a simple base-estimator. As such, it is impossible to create nested structures as in our case.

TPOT also focuses more on the feature preprocessing step of the workflow, introducing additional built-in transformers beyond the scikit-learn provided algorithms, for example a more scalable OneHotEncoder or the stacking ensemble. TPOT uses either a standard tournament selection or NSGA-II and multiobjective fitness. The latter is however composed from the score and the tree size, whereas our system uses the score and evaluation time.

The three dimensions of the optimization, which were mentioned in the context of NAS, may be partly applied to pipeline optimization as well. For example, the search space of our method are all pipelines, whereas the TPOT operates on the search space of tree based pipelines. In both cases, the search strategy is the genetic programming, more precisely the developmental GP in our case. Lastly, both performance estimation strategies of TPOT and of our system are based on 'lower fidelities' — our model proposes sampling strategies and reduces the instance count, whereas in TPOT there has been recently introduced a special biologically inspired feature selection that significantly reduced the running time [Le et al., 2018].

Comparison of our system with GP-ML The idea behind GP-ML is to separate the actual ensemble methods from the splitters and combinators. A splitter is a method that takes input data and splits or copies them into new data; the result is then passed to base-estimators. The combiner in turn combines the output of the estimators into a final output.

Compared with our system, this approach is more general, as in our case both splitting and combining is implicitly defined by a specific ensemble, which acts as a black box. Thus, if we want to combine a specific splitting strategy with a combining function, we must define a new ensemble. On the other hand, as the developmental GP has a simpler constraint interface, the extensibility is probably easier. Moreover, it is more easily convertible to scikit-learn pipeline API, which opens up the optimization space for new methods. Lastly, to represent a pipeline as an individual of GP-ML, it is necessary to utilize more nodes than in our case.

3. Our solution

In our solution we design an AutoML system for workflow optimization based on developmental genetic programming. Compared to most of the existing systems it supports arbitrary-sized pipelines as well as complex ensemble structures. An overview of the process is shown in Algorithm 2.

The input of the algorithm is the dataset for which we want to find an optimized pipeline, and configuration of the system. The latter comprises of settings of the underlying evolutionary algorithm, configuration related to encoding (Tables 3.1 and 3.2.1, or a user-defined alternative), and an evaluation method along with the metric used for scoring.

The dataset provided to the algorithm must be already preprocessed to some extent, as no imputation of missing values or string feature encoding is performed. On the other hand, feature selection and scaling is a part of the algorithm.

When everything is set up, we run the developmental GP optimization (line 2 of Algorithm 2), which operates with pipelines in a tree-based representation. The output of this algorithm are individuals which performed the best on the given dataset. The individuals are decoded into pipelines (line 3) and returned as the final output of the algorithm.

The process of the developmental GP along with the specific tree encoding of individuals is described in more detail in section 3.1. Evaluation of pipelines and implementation details as well as relation to scikit-learn (Pedregosa et al. [2011]) are presented in section 3.2.

Algorithm 2: Pipeline optimization — main

```
Data: dataset $d$, configuration $c$
Result: optimized pipelines
1
2 $individuals \longleftarrow$ run developmental GP on $d$ with $c$
3 $pipelines \longleftarrow$ compile($individuals$)
4
5 return $pipelines$
```

3.1 Evolutionary optimization of pipelines

In this section, we describe the necessary components of the evolutionary algorithm. The process corresponds to the scheme of a general evolutionary algorithm (Algorithm 1). A more detailed scheme is presented in Algorithm 3.

The input of the algorithm is the population size, maximum number of generations (which defines the stopping condition), probabilities of a genetic operator being applied, node arity and tree height limits.

At the beginning, the initial population of individuals, that is, encoded pipelines, is generated and evaluated (lines 12 and 14). If an error occurred during the evaluation of a pipeline, the corresponding individual is removed from the population. New individuals are then generated and evaluated until a valid one is found.

We then run the actual evolutionary optimization (line 19). In every generation, we first perform the parent selection of individuals for reproduction (line 22). Then, the genetic operators are applied with well-defined probabilities one after another (lines 23–31). The newly created offspring are evaluated and added to population of offspring; in case of invalid fitness, we generate a valid individual instead (lines 33–36). Finally, the next generation is created by performing the NSGA-II selection on population of offspring and parents combined (line 38).

In the following sections we describe the components of this particular evolutionary algorithm in detail. The encoding of pipelines is summarized in section 3.1.1, and the initialization procedure is explained in section 3.1.2. All reproduction operators are presented in section 3.1.3. Finally, the selection and fitness are specified in section 3.1.4.

3.1.1 Individual encoding

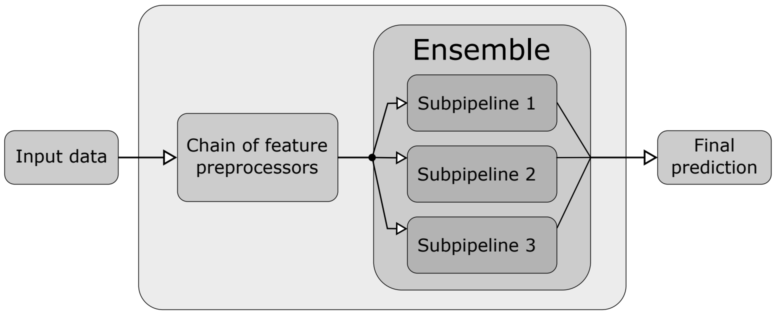


Figure 3.1: Scheme of an example pipeline

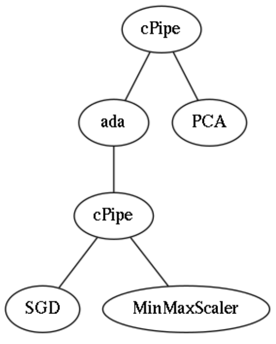
The encoding is one of the most important parts of this system. The individual represents a particular ML pipeline, which is composed of a chain of feature preprocessing methods and of a final estimator. Additionally, we may allow more complex pipeline steps like ensembles and complex feature preprocessing methods like stacking or feature union (some of these methods are described in the book by Brazdil et al. [2008]). In such case, most of the pipelines become in fact directed acyclic graphs (Figure 3.1). Therefore, we cannot directly use the simple tree-based encoding. Instead we use the developmental GP with cellular encoding described in section 1.4.2.

In our case, the embryo is an empty pipeline. To create a complex pipeline, we modify it by inserting steps into it.

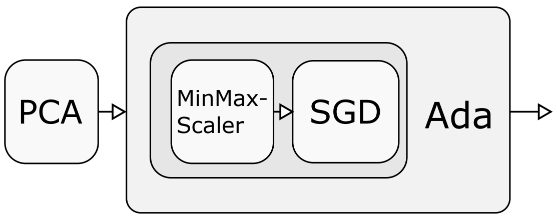
The process can be demonstrated on Figure 3.2. The root of the tree represents the embryo which will be modified by subsequent operations. In this case, the left subtree modifies the ensemble structure whereas the right subtree modifies the feature preprocessor chain. The pipeline contains only one preprocessor, hence the right subtree is terminated by the corresponding node, PCA in this case. The left child can be either an ensemble or a simple method. Here it is the AdaBoost ensemble which has one base classifier. The subestimator is again a pipeline, which is composed of a MinMaxScaler and Stochastic gradient descent classifier. The specific hyperparameters of every pipeline step are stored aside the nodes and are not depicted in the figures.

Algorithm 3: Pipeline optimization — developmental GP

```
Data: population size $k$, maximum number of generations $max\_gen$, crossover
      probability $p_{cx}$, mutation probabilities $p_{mut}$, $p_{mut\_node}$, $p_{mut\_args}$, height
      and arity limits $max\_height$ and $max\_arity$
Result: evolved tree individuals

def evaluate($ind$):
    $pipe \longleftarrow$ compile($ind$)
    $score, time \longleftarrow$ cross-validate $pipe$ on a sample
    $ind.fitness \longleftarrow (score, \log(time))$

def generate_valid():
    while $score$ is not valid:
        $ind \longleftarrow$ initialize a new individual
        $score \longleftarrow$ evaluate($ind$)

/* run developmental GP */
$P(0) \longleftarrow$ initialize population of GP trees

/* compute fitness of initial population */
for $ind$ in $P(n)$:
    evaluate($ind$)
    if fitness is not valid:
        generate_valid($ind$)

/* the process of evolution */
while $n < max\_gen$:

    /* reproduction */
    for $i$ in range($k/2$):
        $i_1, i_2 \longleftarrow$ tournament selection from $P(n)$
        if $p_{cx}$:
            crossover($i_1$, $i_2$)                 /* subtree crossover */

        if $p_{mut}$ for $k = 1, 2$:
            mutation($i_k$)                          /* subtree mutation */
        if $p_{mut\_node}$ for $k = 1, 2$:
            node_mutation($i_k$)                        /* node mutation */
        if $p_{mut\_args}$ for $k = 1, 2$:
            arg_mutation($i_k$)               /* hyperparameter mutation */

        if evaluate($i_k$) for $k = 1, 2$:
            add $i_k$ to offspring population $P_o(n)$
        else:
            generate_valid($i_k$) and add to $P_o(n)$

    $P(n+1) \longleftarrow$ NSGA-II selection from $P_o(n)$ and $P(n)$

return Pareto front of $P(c)$
```



(a) Tree encoding



(b) Encoded pipeline

Figure 3.2: An example pipeline encoded into a tree individual. The word *cPipe* stands for our pipeline representation, other acronyms refer to corresponding ML methods — the AdaBoostClassifier ensemble method, SGD classifier and preprocessing methods PCA and MinMaxScaler.

Table 3.1 lists all nodes used in the current implementation of our system. There are several nodes that represent structure altering operations. The most common nodes are *cPipe* and *cPred*, which create a pipeline in the model. The pipeline is composed from an estimator and, in case of *cPipe*, of a feature preprocessor chain. The estimator may be another pipeline or a single base estimator.

The nodes *cFeatSelect*, *cScale* and *cData* all create the feature preprocessor chain. The first node creates a chain that contains only a feature selection method, the second creates a chain with a scaling method and the last creates a chain with both types of preprocessors. This way, the methods may be logically ordered one after another. The last node which operates with preprocessing methods is *dUnion*. It inserts a feature union into the preprocessor chain — a method that takes several feature preprocessing methods and combines the outputs into a final output.

Lastly, we define a specific node for every ensemble, preprocessing and classifier method which is present in 3.2.1. The purpose of these methods is to define which particular method should be inserted into a pipeline.

In order to produce reasonable pipelines (i.e. with correct logical ordering of methods), we use the strongly typed genetic programming. This means that a node has a well-defined input and output type. Formally, the input type of a function node is defined as a Cartesian product of types, whereas the output type is a single type; terminals have only the output type specified. Output types of child nodes must match the input type of parent node. The list of nodes is extensible, it is possible to add a definition of similar nodes, e.g. a different ensemble flavour like stacking. The nodes that are specific for a given estimator correspond to the list of methods in section 3.2.1. To decode the pipeline, the tree is traversed from root to leaves while applying the operations associated with the nodes.

3.1.2 Initialization

To initialize an individual, we use a modified grow method (the original algorithm is described in section 1.4.1). For the initialization process, we define two sets

Table 3.1: Nodes representing modifying operations

Node[1]	**In type**[2]	**Out type**[3]	**Operation**
cPipe	ens × data	out	Create pipeline with a preprocessor chain and a predictor
cPred	ens	out	Create pipeline only with a predictor
cData	featsel × scale	data	Create preprocessor chain with feature selector and scaler
cFeatSelect	featsel	data	Create preprocessor chain only with a feature selector
cScale	scale	data	Create preprocessor chain only with a scaler
dUnion	data^n	data	Create feature union in the preprocessor chain
ensemble	out^n	ens	Insert ensemble
classifier	$\emptyset$	out	Insert classifier
selector	$\emptyset$	featsel	Insert feature selector
scaler	$\emptyset$	scale	Insert scaler

[1] *There is one specific node per ensemble, classifier and preprocessor present*

[2] *Variable arity is allowed (i.e.* $n \in [1, max\_arity]$*)*

[3] *In the last level classifier and preprocessing can have output type* ens *and* data *resp.*

of nodes. The first (S_{grow}) is a set of functions and terminals and is used in the growing phase, that is, until the height limit is reached. The second set (S_{term}) contains all terminals contained in the first set, and also some additional terminals — copies of all classifier nodes, but of output type 'ens', and copies of all preprocessors, but instead of type 'data'. The reason of this is that if there is a cPipe, cPred or dUnion node (Table 3.1) in the penultimate level of the tree, there is no terminal to terminate with in the first set. Thus, the second set is defined as to be able to finish the initialization properly.

Algorithm 4 shows the process in detail. Along with the two sets it takes as its input the height and arity limits and the dictionary of method hyperparameters associated with the nodes. First, we determine the height of the tree as a random number between 1 and max_height, which is the hyperparameter of our system (line 2). If the selected height is one, we directly return a random classifier from S_{term}. Otherwise, we insert one of the pipeline creating operations into the tree (line 3).

The tree is then grown from root, choosing nodes from S_{grow} or S_{term} respectively. When inserting a node, we first choose a function which has any argument without a child associated with it. To preserve the type consistency, we randomly select a node whose output type matches the type of this argument. Also, we choose its hyperparameters and, if it is a function, we define its arity (this is described in section 3.1.4). Finally, the node can be inserted as a child of the function into the tree (lines 8–21). This process ends as soon as all functions have a child for every argument. The output of the algorithm is the generated individual.

Algorithm 4: Modified grow method for individual initialization

```
Data: maximum height $max\_height$, maximum arity $max\_arity$, sets of
      nodes $S_{grow}$ and $S_{term}$, hyperparameter dictionary $H$
Result: generated individual

$h \longleftarrow$ randomly select tree height from interval $[1, max\_height]$
if $h > 1$:
    set $cPipe$ or $cPred$ as the root of $ind$
else:
    return a random classifier from $S_{term}$

while $ind$ has a function with a child missing:
    $f \longleftarrow$ a parent node which can accept a child
    $t_f \longleftarrow$ argument input type of $f$
    $h_{node} \longleftarrow$ height at which the node is inserted

    if $h_{node} < h$:
        $S \longleftarrow S_{grow}$
    else:
        $S \longleftarrow S_{term}$
    $node \longleftarrow$ randomly choose a node from $S$ with output type $t_f$
    $node.arity \longleftarrow$ choose_arity($node$, $max\_arity$)
    $node.hyperparams \longleftarrow$ choose_hyperparams($node$, $H$)

    add the child $node$ as an argument of $f$
return $ind$
```

As terminals are inserted in the growing phase as well, the tree may become smaller than the height limit. However, if the tree were built using the full method, it would introduce a lot of feature preprocessing methods for taller trees.

Weighted selection During the node selection, we use weights to manage the probability of a node to be chosen. The motivation is that some nodes represent lightweight methods which have a short execution time, whereas some nodes slow down the evaluation process, especially when present multiple times in the tree. In other words, this is one of the bloat-reducing techniques used in our work.

The process is as follows: every node is assigned to a group and each group has a well defined weight. When selecting a particular node with output type *out*, we first determine all groups G which correspond to any node with output type *out*. Then we select a group g from G by a weighted random choice. Finally, the node is selected by a simple random choice from g.

The default values of weights are shown in Table 3.2. Every node from Table 3.1 is part of exactly one of the groups — cPipe and cPred nodes are in group *pipeline*, the group *union* contains only the dUnion node and *transform* is the group of cData, cFeatSelect and sScale nodes. The rest of the groups corresponds to nodes specific for a given machine learning methods, that is, *predictor* is a group of all classifiers and so on. The *ensemble_l* is a group of 'lightweight' ensembles; it contains the voting ensemble, which is not so computationally demanding as for example the AdaBoostClassifier, and therefore it can be selected more frequently.

Table 3.2: Default weights of node groups

Group name	Weight
pipeline	1.0
union	0.3
transform	1.0
prepro	1.0
ensemble	0.5
ensemble_l[1]	1.0
predictor	1.0

[1]*less demanding ensembles*

Variable arity Some nodes, e.g. ensemble nodes, may have a *variable arity*. For example, a voting may have two base estimators or many more, but the actual model combination method remains the same. Therefore, it is represented by a single node type for convenience.

The input type of a node is a Cartesian product of its argument types. If we allow variable arity of each type, the input type is instead defined as

$$t = \prod_{i=1}^{n} t_i^{k_i},$$

where t_i is a specific type, k_i is its arity and $\forall i = 1 \dots n-1, t_i \neq t_{i+1}$. The representation is defined in a general way as to allow ensembles with groups of arguments. The ensemble methods used in this work (Table 3.1) are all relatively simple, but in future extensions, more complex cases may be added.

The actual arity is determined just when the node is about to be inserted into the tree. The range of possible arities is specified by an interval which may or may not have an upper bound. If the upper bound is not provided, the values are limited by a global arity limit (*max_arity* in Algorithm 4) to avoid bloat of the trees.

Method hyperparameters Every node has a list of possible values per hyperparameter associated with it. During initialization, the actual values are randomly selected from every list. The validity is not verified in this phase, instead it is handled in the evaluation phase (section 3.2).

3.1.3 Genetic operators

In our system, we use one type of crossover and three different types of mutation. The crossover is the standard strongly typed subtree-swap operation, as described in section 1.4.1. The first mutation type is the *subtree mutation*, which is used to replace whole parts of the pipeline with new structures. The other two mutation

Algorithm 5: Input type arity comparison

```
Data: type $t$ with fixed arities, type $t'$ with variable arities
Result: true if $t$ can be equal to $t'$

if the number of subtypes of $t$ and $t'$ differs:
    return false

for i in range($n$):
    if arity of $t_i$ is not in the arity interval of $t'_i$:
        return false
return true
```

types are more conservative: the *point mutation* replaces a random node of the tree with a new one, while the *node argument mutation* randomly changes a hyperparameter of a node. The mutation operators will be described in more detail in the following paragraphs.

Subtree mutation This mutation type is in the most 'aggressive' mutation type in this system, as it greatly alters the structure of a pipeline. As defined in section 1.4.1, in subtree mutation a chosen subtree is replaced with a randomly generated tree. In our implementation we moreover limit the height of the generated tree. For height h of the subtree, the height of the newly generated tree must be between $[1, h + \epsilon]$ for a small value of ϵ. This way we ensure that for small subtrees the new subtree may be slightly higher and for big subtrees the overall height should not increase too much. In other words, we try to minimize the bloat induced by this operation.

Point mutation The point mutation enables to change only one step of the pipeline while preserving the overall structure. In this mutation, a randomly chosen node is replaced with a new node. Both output and input types must match. Moreover, if the new node supports variable arity, we must compare the subtypes one by one, as depicted in 5. For this type of mutation, any lower bound of any arity must be greater than zero.

Node argument mutation This type of mutation changes one hyperparameter of a random node — chooses a new value from the list of possible values. On some problem instances, this mutation is essential (as shown in experiment 4.2), because it directly searches the hyperparameter space of a single pipeline. There are many possibilities in how to improve this operator, which could significantly improve the results of our system. These will be part of future research of our system.

3.1.4 Fitness and selection

To compute the fitness, the individuals are decoded into scikit-learn pipelines as described in section 3.1.1 (individual encoding). The specific machine learning methods which are used in the pipelines along with evaluation details are listed in the following section.

The fitness has two objectives — the evaluation score and logarithmized evaluation time. The enviromental selection is done via the NSGA-II operator (described in section 1.3.2), and the parent selection is a tournament selection based on individual dominance and crowding distance. This approach allows us to prefer simpler yet well-performing pipelines, as complex ensemble methods are typically time-consuming.

It may happen that some pipeline fails to run, for example due to an unsupported hyperparameter combination. In that case, the individual is discarded and a new one is generated. If the next individual is not valid either, the process is repeated until a valid individual is generated.

3.2 Evaluation and performance estimation

In this section we elaborate on the evaluation process. First we present the pipelines and used methods in more detail. Then we present the evaluation and a performance estimation method which was used to decrease the running time.

3.2.1 Used machine learning methods

We use the scikit-learn implementation of pipelines. An arbitrary pipeline consists of multiple transformer steps and one predictor step, which is either an ensemble or a base-learner. Any machine learning method that complies to the scikit-learn API can act as a pipeline step [Buitinck et al., 2013]. Regression is supported by the system as well, but in this work we focus only on classification problems. Tables 3.3, 3.4 and 3.5 show all machine learning methods present in the default configuration. Every method has a list of hyperparameter values associated with it. These are optimized during the process of evolution; if a hyperparameter is not present, it will be always set to its default value. The list has been created by slightly extending the hyperparameter list used by Křen et al. [2017] in the AutoML system mentioned in section 2.

Table 3.3: Used classifiers with hyperparameters

KNeighborsClassifier	
n_neighbors	[1, 2, 5]
algorithm	[’auto’, ’ball_tree’, ’kd_tree’, ’brute’]
LinearSVC	
loss	[hinge,squared_hinge]
penalty	[l1,l2]
C	[0.1,0.5,1.0,2,5,10,15]
tol	[0.0001,0.001,0.01]
SVC	
C	[0.1,0.5,1.0,2,5,10,15]
gamma	[scale,0.0001,0.001,0.01,0.1,0.5]
tol	[0.0001,0.001,0.01]
LogisticRegression	

penalty	[l1,l2]
C	[0.1,0.5,1.0,2,5,10,15]
tol	[0.0001,0.001,0.01]
solver	[newton-cg,lbfgs,liblinear,sag,saga]
Perceptron	
penalty	[None,l2,l1,elasticnet]
n_iter	[1,2,5,10,100]
alpha	[0.0001,0.001,0.01]
SGDClassifier	
penalty	[none,l2,l1,elasticnet]
loss	[hinge,log,modified_huber,squared_hinge,perceptron]
max_iter	[10,100,200]
tol	[0.0001,0.001,0.01]
alpha	[0.0001,0.001,0.01]
l1_ratio	[0,0.15,0.5,1]
epsilon	[0.01,0.05,0.1,0.5]
learning_rate	[constant,optimal]
eta0	[0.01,0.1,0.5]
power_t	[0.1,0.5,1,2]
PassiveAggressiveClassifier	
loss	[hinge,squared_hinge]
C	[0.1,0.5,1.0,2,5,10,15]
LinearDiscriminantAnalysis	
solver	[lsqr,eigen]
shrinkage	[None,auto,0.1,0.5,1.0]
QuadraticDiscriminantAnalysis	
reg_param	[0.0,0.1,0.5,1]
tol	[0.0001,0.001,0.01]
MLPClassifier	
activation	[identity,logistic,relu]
solver	[lbfgs,sgd,adam]
alpha	[0.0001,0.001,0.01]
learning_rate	[constant,invscaling,adaptive]
tol	[0.0001,0.001,0.01]
max_iter	[10,100,200]
learning_rate_init	[0.0001,0.001,0.01]
power_t	[0.1,0.5,1,2]
momentum	[0.1,0.5,0.9]
hidden_layer_sizes	[(100,),(50,),(20,),(10,)]
DecisionTreeClassifier	
criterion	[gini,entropy]
max_features	[0.05,0.1,0.25,0.5,0.75,1]
max_depth	[1,2,5,10,15,25,50,100]
min_samples_split	[2,5,10,20]
min_samples_leaf	[1,2,5,10,20]
GaussianNB	
-	
GradientBoostingClassifier	
loss	[deviance,exponential]

n_estimators	[20,50,100,200]
subsample	[0.3,0.5,0.75,1.0]
RandomForestClassifier	
n_estimators	[10,50,100,150,200]
ExtraTreesClassifier	
n_estimators	[10,50,100,150,200]

Table 3.4: Used preprocessors with hyperparameters

NMF	
feat_frac	[0.01,0.05,0.1,0.25,0.5,0.75,1]
solver	[cd,mu]
FactorAnalysis	
feat_frac	[0.01,0.05,0.1,0.25,0.5,0.75,1]
FastICA	
feat_frac	[0.01,0.05,0.1,0.25,0.5,0.75,1]
PCA	
feat_frac	[0.01,0.05,0.1,0.25,0.5,0.75,1]
whiten	[False,True]
SelectKBest	
feat_frac	[0.01,0.05,0.1,0.25,0.5,0.75,1]
score_func	[feature_selection.chi2,feature_selection.f_classif]
MaxAbsScaler	
-	
MinMaxScaler	
-	
Normalizer	
-	
StandardScaler	
-	

Note: *feat_frac* is an artificial feature which represents a fraction of total feature count; it is converted to *n_components* or *k* respectively

Table 3.5: Used ensembles with hyperparameters

AdaBoostClassifier	
n_estimators	[5, 10, 50, 100, 200]
algorithm	[SAMME, SAMME.R]
BaggingClassifier	
n_estimators	[5, 10, 50, 100, 200]
VotingClassifier	
voting	[hard]

3.2.2 Scoring and sampling

The score is computed by running the pipeline on the dataset using the scikit-learn scorer interface [scikit-learn, 2019]. The default score is predictive accuracy, but other metrics are supported as well. There are multiple evaluation strategies to choose from. The default strategy is the k-fold cross-validation. In this evaluation method, the dataset is split into k parts and in every 'fold', one part is used as the testing set and the rest is used as the training set. Every part is used exactly once, hence we obtain k different scores which are then averaged into a final score. Another evaluation strategy is to provide as separate validation set for scoring purpose, while using the original dataset for training.

If the number of examples is small enough, it is possible to use the whole dataset for evaluation. On larger datasets though, the duration time may be too long even for a small number of fitness evaluations. As such, it is necessary to decrease the evaluation time of a single pipeline. We use one of the performance estimation methods mentioned in section 1.2.3 — evaluation on smaller subsets of data.

The strategy is as follows: we generate stratified random samples of the original dataset. The samples are either generated per generation or per every fitness evaluation. Both approaches were compared in experiment 4.1; although we have not shown any significant difference between the methods, the results had a greater variance if the sample was generated for every individual evaluation. A possible explanation is that if we generate a sample per generation, the fitness of individuals from a single generation is directly comparable, but in the second case all samples are different and one fitness value may have a diametrically different meaning than another. More flexible sampling and evaluation methods should be developed in future extensions of the system as to improve the overall results.

As the samples are typically much smaller than the whole dataset, the score is computed as a (stratified) k-fold cross-validation on a sample.

3.3 Implementation

The source code of our system is available in a public GitHub repository [Suchopárová, 2019]. For the machine-learning side of the implementation we used scikit-learn [Pedregosa et al., 2011].

We extended the Pipeline class in order to enable usage of pipelines as base-learners (some ensembles require `predict_proba` which is not available on standard pipelines). A meta-transformer was also added to provide conversion between `feat_frac` and `n_components` or `k` hyperparameters respectively (as reflected in Table 3.4).

The components of the evolutionary algorithm were managed by the tools provided by the library DEAP [Fortin et al., 2012]. However, although DEAP supports genetic programming, the primitives cannot be created with variable arity, hence we reimplemented the concept. For parallelization of fitness evaluations we used the library Joblib [Joblib, 2019].

3.3.1 Configurability

The system can be customised by following configuration hyperparameters:

- population size
- number of generations
- maximum tree height
- maximum arity (global for all nodes)
- timeout per individual evaluation (results in invalid score)
- evaluation strategy
- recombination probabilities from algorithm 3
- custom scorer (according to scikit-learn API)

A set of scripts designed for testing of the system is also included in the project repository.

4. Experiments

In this chapter, we present results of three experiments which we carried out in order to estimate the performance of our system and to explore different hyperparameter settings. The outcomes of these experiments can determine the direction of future research on our system and enable us to compare it with reference data.

The first two experiments were aimed at discovering relations between particular hyperparameter settings. In the first experiment (section 4.1) we compare the two sampling strategies which were presented in section 3.2.2. The second experiment focuses on genetic operators; we tried to determine whether the result changes if we turn one of these operators off (section 4.2).

The goal of the last experiment was to evaluate our system on benchmark datasets and compare the results with reference data (section 4.3). We used the OpenML-CC18 benchmarking suite for this purpose [van Rijn, 2019].

4.1 Sampling strategies

In section 3.2.2 we presented two different performance estimation strategies aimed at reducing the evaluation time. Both methods are based on sampling of the full dataset, but the generation of the samples differs — either a sample of the original dataset is generated for every individual evaluation (*per-ind*), or only once per generation (*per-gen*), and it is shared by all individuals. The goal of this experiment was to test whether one approach is better and, if possible, to compare it with evaluation on the complete dataset (the *full* strategy).

The strategies were tested on three different datasets:

- wilt — Medium size dataset (4839 instances, 6 features), the task is to detect diseased trees in image segments. There are two target classes: 'w' (diseased trees) and 'n' (all other land cover). The dataset is heavily unbalanced; only 261 instances of the 'w' class are present [Johnson et al., 2013].
- wine-quality-white — Medium size dataset (4898 instances, 12 features), the features represent properties of a particular Portuguese wine. Each of the wines has been graded by three experts according to sensory properties, the target value is a median of these grades. The task is to predict the target grading (1–7), which may be either a classification or a regression task. We solve it as a classification task [Cortez et al., 2009].
- MagicTelescope (magic) — Large dataset (19020 instances, 12 features) the task is to determine whether the data produced by the Cherenkov gamma telescope describes a 'signal' or only 'background data' [Bock et al., 2004].

Setting The sample size was chosen proportionally to the dataset size to sufficiently reduce the running time — $\frac{1}{4}$ of all instances in case of the medium-sized datasets and $\frac{1}{20}$ for the magic dataset. During the evolution, the individual fitness was computed as 5-fold cross-validation on the samples, or on the full dataset respectively. The final evaluation method of resulting optimized pipelines was 10 times 10-fold cross-validation on the full dataset.

Table 4.1: System hyperparameters for the sampling experiment

Hyperparameter	**Value**
population size	200
maximum generation	15
crossover probability	0.5
subtree mutation probability	0.3
node argument mutation probability	0.6
node mutation probability	0.3
maximum tree height	5
timeout per method	7 minutes
group weights	*default*

Hyperparameter settings of the system are presented in Table 4.1, their influence on the system is discussed in Chapter 3. For the wilt dataset we used Cohen's kappa as the scoring method [Cohen, 1960]. Cohen's kappa is a statistic used to measure agreement of the classifier with the 'ground truth', with values ranging from -1.0 to 1.0. Value of -1.0 corresponds to complete disagreement, 1.0 to complete agreement and 0.0 is equivalent to random guessing. The reason why this more specific statistic was used is that due to wilt being an unbalanced dataset, the cross-validation accuracy score is close to 1.0 and differences between run results may not be apparent. With Cohen's kappa having a wider range of values, the differences can be observed in more detail.

Table 4.2: Boxplot statistics of the sample experiment (magic)

	minimum	median	confidence interval	maximum	eval. time[1]
per_gen	0.8645	0.8792	$(0.8773, 0.8811)$	0.882	28 min
per_ind	0.8720	0.8795	$(0.8772, 0.8819)$	0.882	34 min

[1] *Evaluation time of the first generation*

Table 4.3: Boxplot statistics of the sample experiment (wilt)

	minimum	median	confidence interval	maximum	eval. time[1]
full	0.8695	0.8798	$(0.8736, 0.8861)$	0.8908	25 min
per_gen	0.8684	0.8730	$(0.8699, 0.8760)$	0.8878	11 min
per_ind	0.8661	0.8736	$(0.8708, 0.8764)$	0.8877	6 min

[1] *Evaluation time of the first generation*

Results For each run we have chosen 5 pipelines with the highest score from the Pareto front, or the full Pareto front if it had been shorter than 5 pipelines. These pipelines were again evaluated using the 10-fold cross-validation. Finally, the maximum of the scores was chosen. Figures 4.1, 4.2 and 4.3 visualize the distribution of the 10 scores per strategy. Relevant statistics of the boxplots

Table 4.4: Boxplot statistics of the sample experiment (wine-quality-white)

	minimum	median	confidence interval	maximum	eval. time[1]
full	0.5524	0.7059	$(0.6491, 0.7627)$	0.7097	53 min
per_gen	0.5295	0.5391	$(0.5372, 0.5409)$	0.5518	27 min
per_ind	0.5340	0.5380	$(0.5343, 0.5417)$	0.5454	35 min

[1] *Evaluation time of the first generation*

are listed in Tables 4.2, 4.3, 4.4. We also list the evaluation times of the first generation in the tables. As the evaluation time greatly depends on generated pipelines, the differences in running times are subject to random fluctuations and are listed only for illustration.

In the figures, middle line of the boxplot represents the median, while notches determine the 95% confidence interval of the median. The whiskers determine the minimum and maximum values respectively (without outliers). The lower and upper edges of the box represent the first and the third quartile respectively.

In all of the experiments we found no statistically significant difference between per-gen and per-ind methods. However, not counting the outliers, per-ind method had a greater variance in all three cases.

The full strategy has been shown to be significantly better on the wine-quality-white dataset. On the wilt dataset, although most of the scores resulting from the full method were better than results of the methods using sampling, the 95% confidence intervals of all strategies overlap. As such, it is not possible to say that the full method is significantly better than the sampling. In addition to that, its result scores had a great variance in both cases.

The overall conclusion from this experiment is that on small datasets, or when the longer duration is not overly limiting, the full method should be preferred.

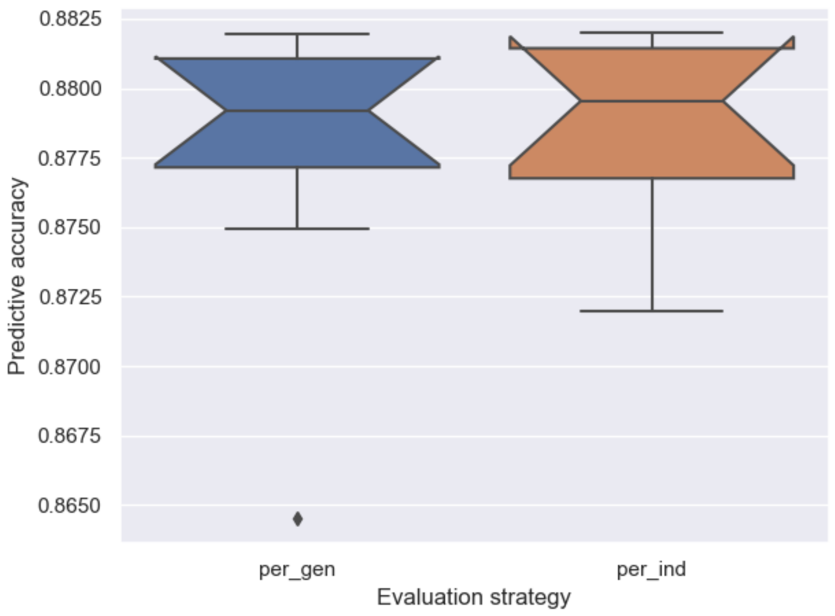


Figure 4.1: Comparison of strategies on the magic dataset

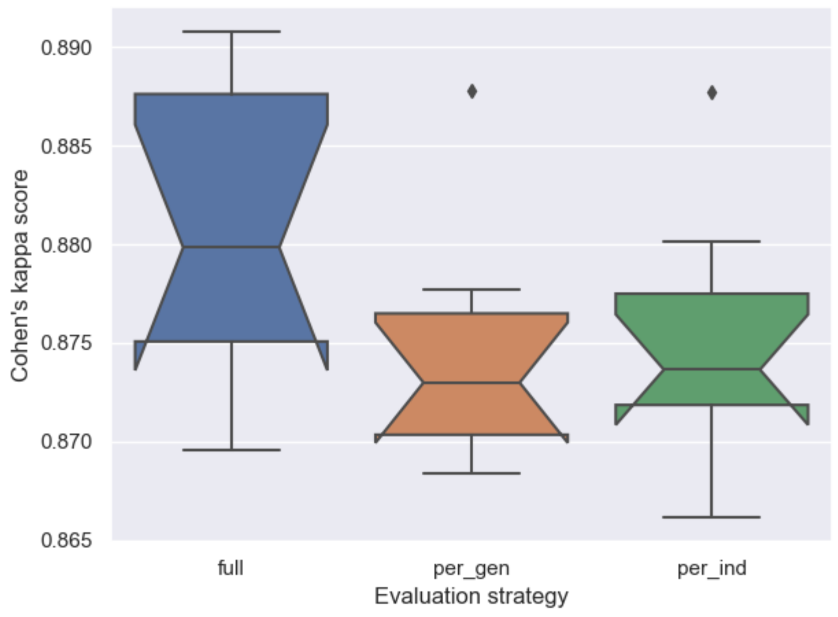


Figure 4.2: Comparison of strategies on the wilt dataset

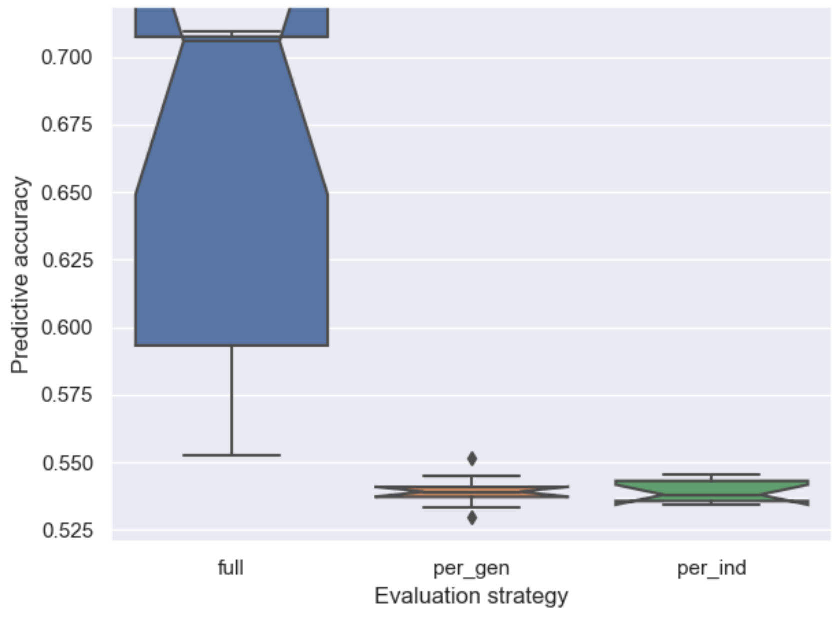


Figure 4.3: Comparison of strategies on the wine-quality-white dataset

In other cases the per-gen method with a reasonable sample size is a good compromise.

4.2 Combinations of genetic operators

The goal of this experiment was to test whether the absence of some genetic operators influences the result score. The motivation is that if we observe a significant difference in results, we can focus on improving a particular genetic operator or alter the default probability settings.

We used two datasets with different properties:

- wilt — Medium sized dataset, see section 4.1
- climate-model-simulation-crashes (climate) — Small dataset (540 instances, 21 features), the task is to predict whether a particular climate model simulation crashes due to numerical or other reasons [Lucas et al., 2013].

Setting As described in section 3.1.3, there is one crossover operator and three mutation operators — the point mutation (or node mutation), subtree mutation and hyperparameter mutation. The following sub-experiments were performed:

- **all** — all genetic operators included
- **no-arg** — hyperparameter mutation turned off
- **no-cx** — crossover turned off
- **no-node** — node mutation turned off
- **no-subtree** — subtree crossover turned off
- **nothing** – all genetic operators turned off

In every sub-experiment all operators that are not excluded keep their default probabilities. The probabilities and other system hyperparameter settings are listed in Table 4.5, with a more detailed explanation of their meaning in Chapter 3. Sample size for the wilt dataset was the same as in section 4.1, the evaluation strategy was per-gen. We did not use any sampling with the climate dataset, as it has only a small number of samples and features. For fitness computation we used the 5-fold cross-validation. Final evaluation and data collection was the same as in the previous experiment — for each run we have chosen 5 pipelines with the highest score from the Pareto front (or less, if the Pareto front was smaller). These pipelines were once again evaluated using the 10-fold cross-validation. The maximum of the scores was chosen as the overall result of the run.

Results The results of this experiment are depicted in Figures 4.4 and 4.5, with boxplot statistics listed in Tables 4.6 and 4.7. Again, the evaluation time of the first generation is listed only for illustration purpose.

In case of the wilt dataset, only the 'no-arg' method did not introduce any improvement when compared to results from the setting with no genetic operators ('nothing'). This probably means that the hyperparameter mutation is essential

Table 4.5: System hyperparameters for the genetic operator experiment

Hyperparameter	**Value**
population size	200
maximum generation	15
crossover probability	0.5
subtree mutation probability	0.3
node argument mutation probability	0.6
node mutation probability	0.3
maximum tree height	5
timeout per method	7 minutes
group weights	*default*

on this dataset — in fact, the best pipelines usually contained a SVC, which relies heavily on hyperparameter optimization [Feurer et al., 2015].

The 'no-node' configuration was significantly better than almost all other methods on the climate dataset (some of the confidence interval slightly overlap though). The reason of this is not apparent and this behaviour may be one of the subjects of future research.

The confidence intervals of medians of the 'nothing' method and of the other methods, except for the 'no-node' setting on climate dataset, overlap. A possible explanation may be that there still is a chance that a good individual is found during the initialization, thus producing a better final score. The conclusion is that the genetic operators and corresponding probabilities need to be tuned to reliably produce better pipelines than those generated in the first generation.

Table 4.6: Boxplot statistics of the genetic operators experiment (wilt)

	minimum	median	confidence interval	maximum	eval. time[1]
all	0.8658	0.8749	$(0.8711, 0.8786)$	0.8820	9 min
no-arg	0.8526	0.8694	$(0.8684, 0.8703)$	0.8794	7 min
no-cx	0.8499	0.8748	$(0.8715, 0.8781)$	0.8875	10 min
no-node	0.8664	0.8747	$(0.8721, 0.8772)$	0.8864	10 min
no-subtree	0.8675	0.8738	$(0.8706, 0.8770)$	0.8850	4 min
nothing	0.8434	0.8659	$(0.8585, 0.8734)$	0.8787	3 min

[1] *Evaluation time of the first generation*

4.3 OpenML-CC18 benchmarking suite

OpenML is a machine learning environment designed for sharing of machine learning problem data [Vanschoren et al., 2013]. It provides a variety of datasets along with tasks and metadata of past runs. The data is accessible via APIs available for several programming languages, notably Python or R [OpenML, 2019]. OpenML classifies the components of solving a machine learning problem into following categories:

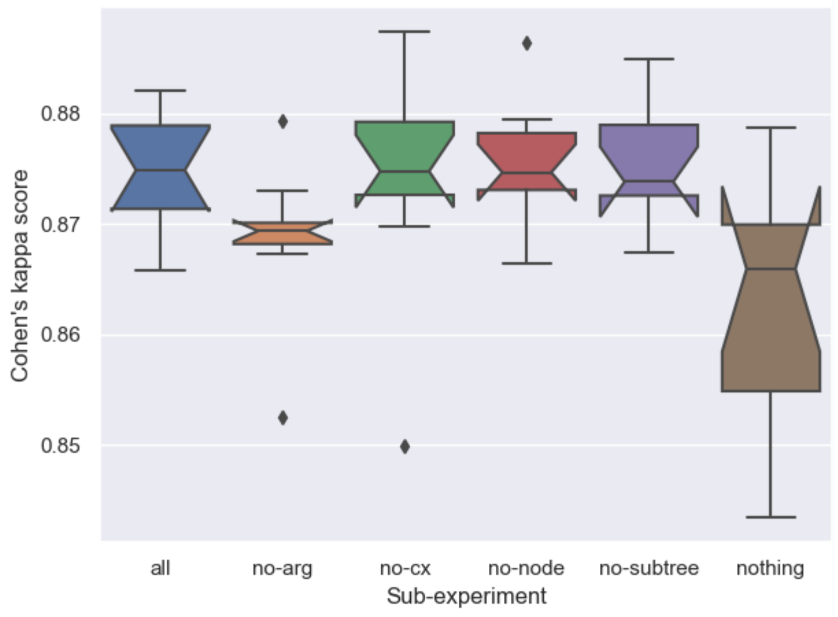


Figure 4.4: Test of genetic operators on the wilt dataset

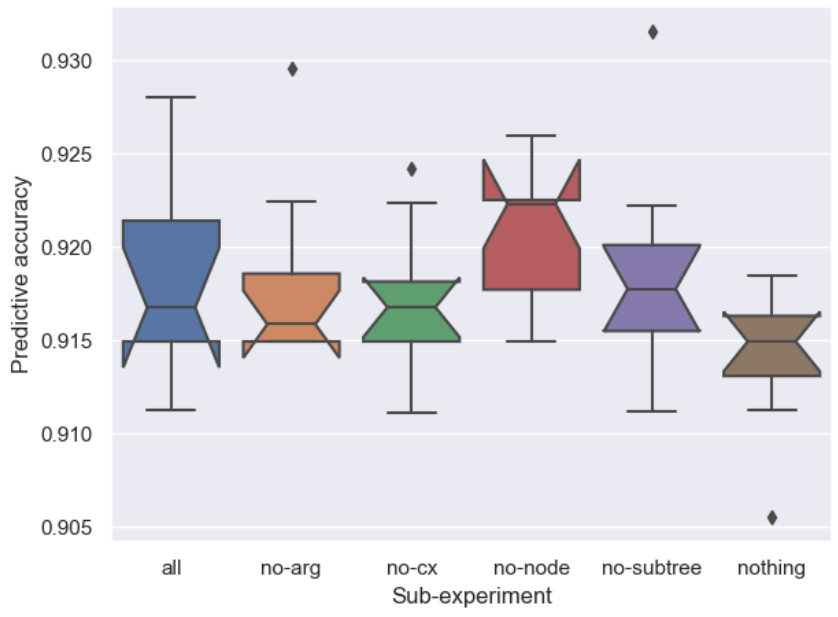


Figure 4.5: Test of genetic operators on the climate dataset

Table 4.7: Boxplot statistics of the genetic operators experiment (climate)

	minimum	median	confidence interval	maximum	eval. time[1]
all	0.9112	0.9167	$(0.9135, 0.9200)$	0.9280	2 min
no-arg	0.9149	0.9159	$(0.9140, 0.9177)$	0.9296	9 min
no-cx	0.9111	0.9168	$(0.9152, 0.9184)$	0.9242	8 min
no-node	0.9149	0.9223	$(0.9199, 0.9247)$	0.9259	8 min
no-subtree	0.9112	0.9177	$(0.9155, 0.9200)$	0.9316	5 min
nothing	0.9055	0.9149	$(0.9133, 0.9165)$	0.9185	2 min

[1]*Evaluation time of the first generation*

- Datasets — The datasets are stored along with description and other metadata.
- Tasks — A task is associated with a particular dataset. An example of a task is 10 times 10-fold cross-validation on the dataset.
- Flows — A flow describes a specific machine learning algorithm.
- Runs — A run is a record of a flow with specified hyperparameters applied to a task.

There are also several concept in development, like a study, which is a collection of data from above-mentioned categories.

In this experiment we have used a study named OpenML-CC18 [van Rijn, 2019]. This collection is a set of classification tasks created for the purpose of practical benchmarking. It contains 72 datasets which have been frequently used in recently published benchmarks and also satisfy several criteria, like a limit on instance count or reasonable task difficulty. The full list of criteria can be found in the OpenML documentation [OpenML, 2019]. Every dataset is associated with one task — a 10-fold cross-validation. Majority of the tasks has already a great number of runs recorded along it, therefore the suite can be used as reference data for our runs.

Setting In order to reduce the overall running time, sample size has been chosen according to dataset size, as can be seen in Table 4.8. If the product of feature count and instance count was less than $10,000$, no sampling was performed. The sizes were chosen in a heuristic manner by examining duration of runs from section 4.1 and 4.2.

Table 4.9 lists hyperparameter settings (their purpose is described in Chapter 3) for all of the benchmark runs. Although we showed in section 4.1 that the per-ind method had a greater variance, it was used here as this experiment preceded the sample experiment. For fitness computation, we used 5-fold cross-validation on a sample. The final evaluation was the 10-fold cross-validation on the full dataset as specified by the task definition. Also, before every run we performed imputation of missing values, because our system does not yet perform this preprocessing. We imputed the mean in case of numerical features and the median in case of categorical features.

Table 4.8: Sample size in relation to the dataset size (OpenML-CC18 experiment)

Row count limit	**Sample size**
small, less than $10,000$ entries[1]	1.0
$1,000$	0.5
$5,000$	0.25
$10,000$	0.1
$20,000$	0.05
large, less than 10 features	0.02
large, more than 10 features	0.01

[1] *Datasets of small number of features and instances were not sampled*

Table 4.9: System hyperparameters for the OpenML-CC18 experiment

Hyperparameter	**Value**
population size	200
maximum generation	15
crossover probability	0.5
subtree mutation probability	0.3
node argument mutation probability	0.6
node mutation probability	0.3
maximum tree height	5
timeout per method	7 minutes
evaluation strategy	per-ind
evaluation metric	predictive accuracy
group weights	*default*

Results Figures 4.6, 4.7, 4.8 and 4.9 show the results of our runs compared with results of runs uploaded to OpenML. For every dataset we have done only one run due to time limitations. For each run we have evaluated top 3 pipelines from the Pareto front and selected the maximum score. Data from runs uploaded to OpenML were extracted using the OpenML python API [OpenML-API, 2019].

The boxplots represent the distribution of predictive accuracies from all runs on a particular dataset, while the red circle marks the score of our run. It should be noted that the number of runs varies greatly between different datasets; some datasets are associated with more than $100,000$ runs, while others only with hundreds of runs or even less. Thus, the appearance of some boxplots may be skewed. Full statistics can be found on the OpenML web page or using the OpenML API [van Rijn, 2019, OpenML, 2019].

As can be seen from the plots, on most of the benchmark datasets our systems found a pipeline which scored above the upper quartile or only slightly below it. In some cases the score was equal or very close to the overall maximum. On the *madelon* dataset we found a pipeline which performed better than the maximum run score recorded in OpenML.

The datasets which were problematic for our system were notably *CIFAR-10* and the *jungle_chess*[1] dataset. The jungle_chess dataset is associated with only about $3,700$ runs on only few flows. Most of the flow pipelines contained a SVC, which performed particularly well on this dataset. Thus, our system did not find the optimal SVC hyperparameter configuration (or a better pipeline), but we cannot determine the relation to other yet to be tested flows.

In case of the CIFAR-10 dataset, there were only 76 runs. Moreover, most of the runs were from flows representing keras models [Chollet et al., 2015]. These are deep neural networks with a very large hyperparameter configuration space and several layers, which is beyond the scope of our system as we use only machine learning methods from scikit-learn.

Our system also performed less well on some small datasets. Usually, a more complex model was one of the individuals with the biggest fitness, but the resulting score was lower than the score of some simpler models. This may have happened due to overfitting, and should be solved in following versions of our system by employing a more elaborate evaluation method.

[1]Full dataset name, as in OpenML: *jungle_chess_2pcs_raw_endgame_complete*

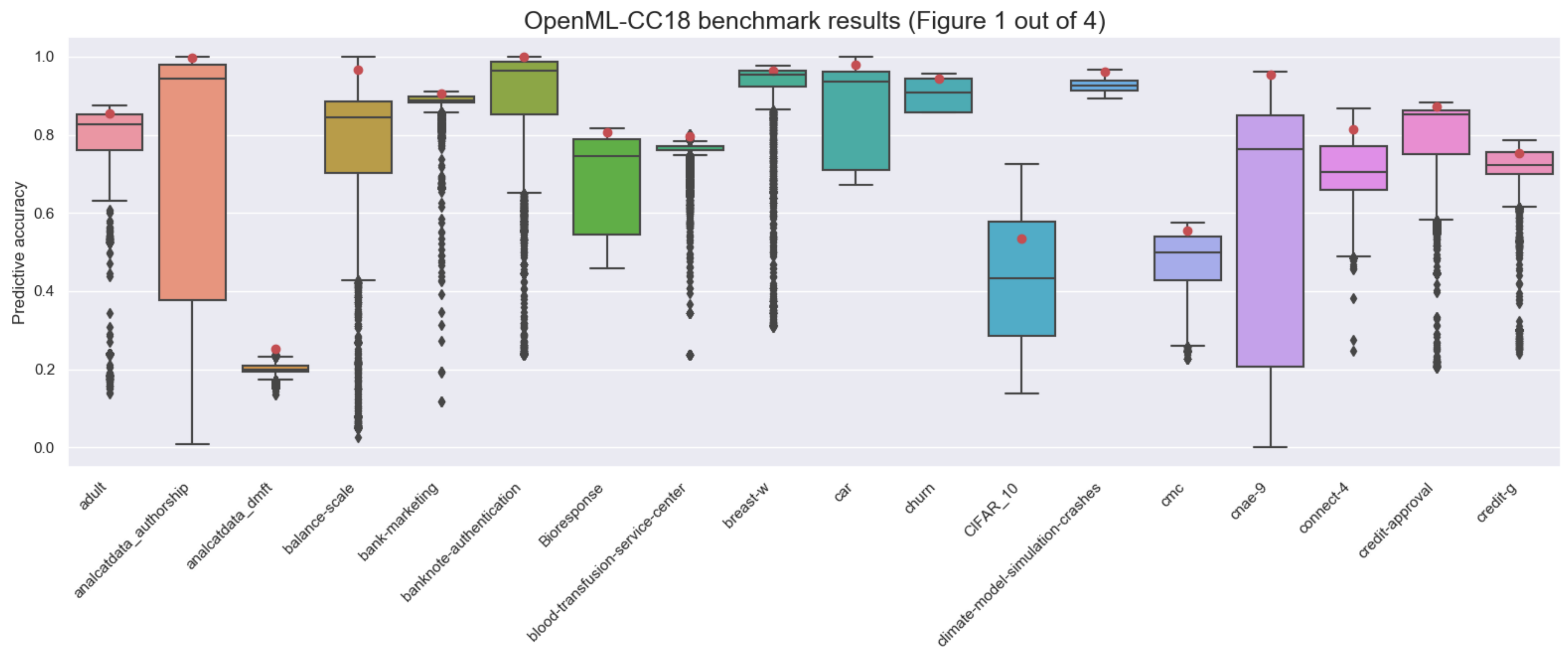


Figure 4.6: OpenML-CC18 benchmark result comparison (Figure 1 out of 4). Boxplots visualize distributions of predictive accuracies of all runs uploaded to OpenML. Red circles mark our accuracy score on datasets.

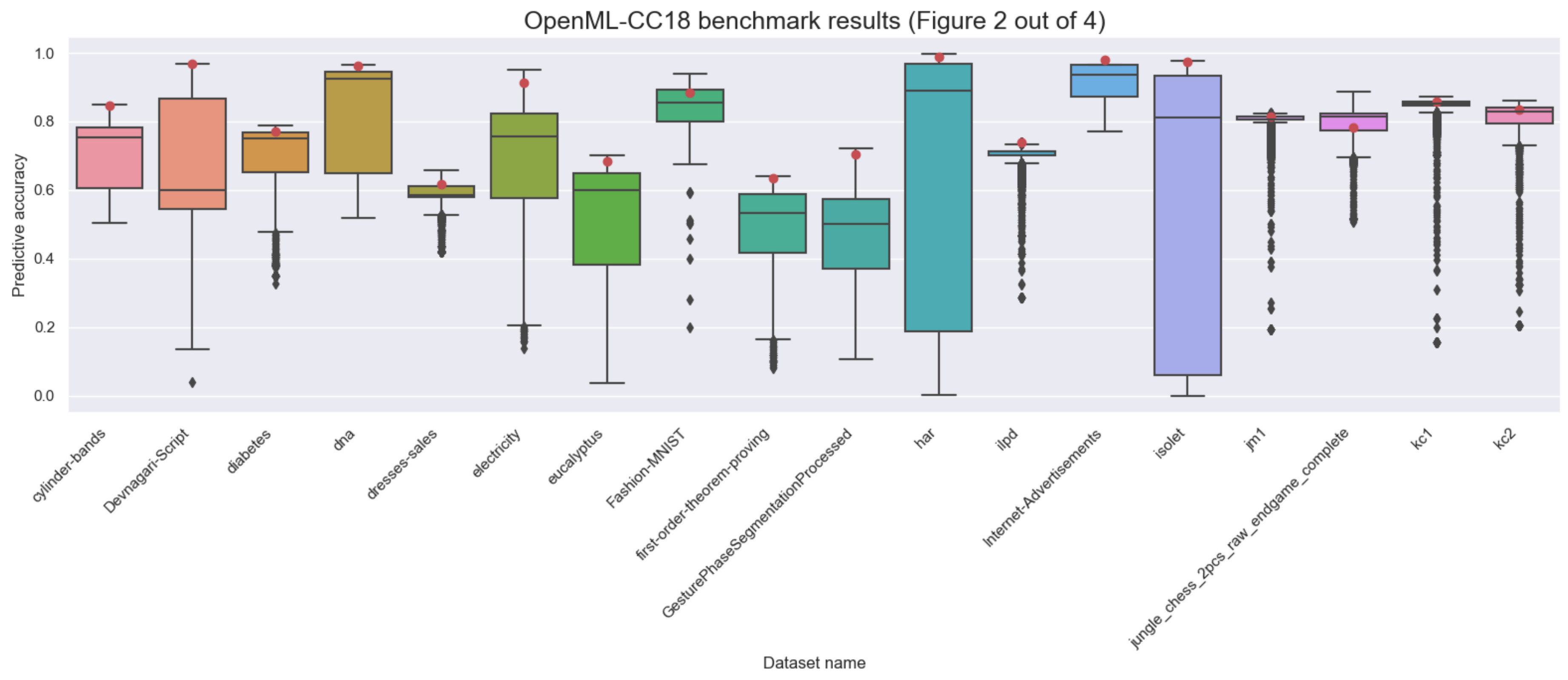


Figure 4.7: OpenML-CC18 benchmark result comparison (Figure 2 out of 4). Boxplots visualize distributions of predictive accuracies of all runs uploaded to OpenML. Red circles mark our accuracy score on datasets.

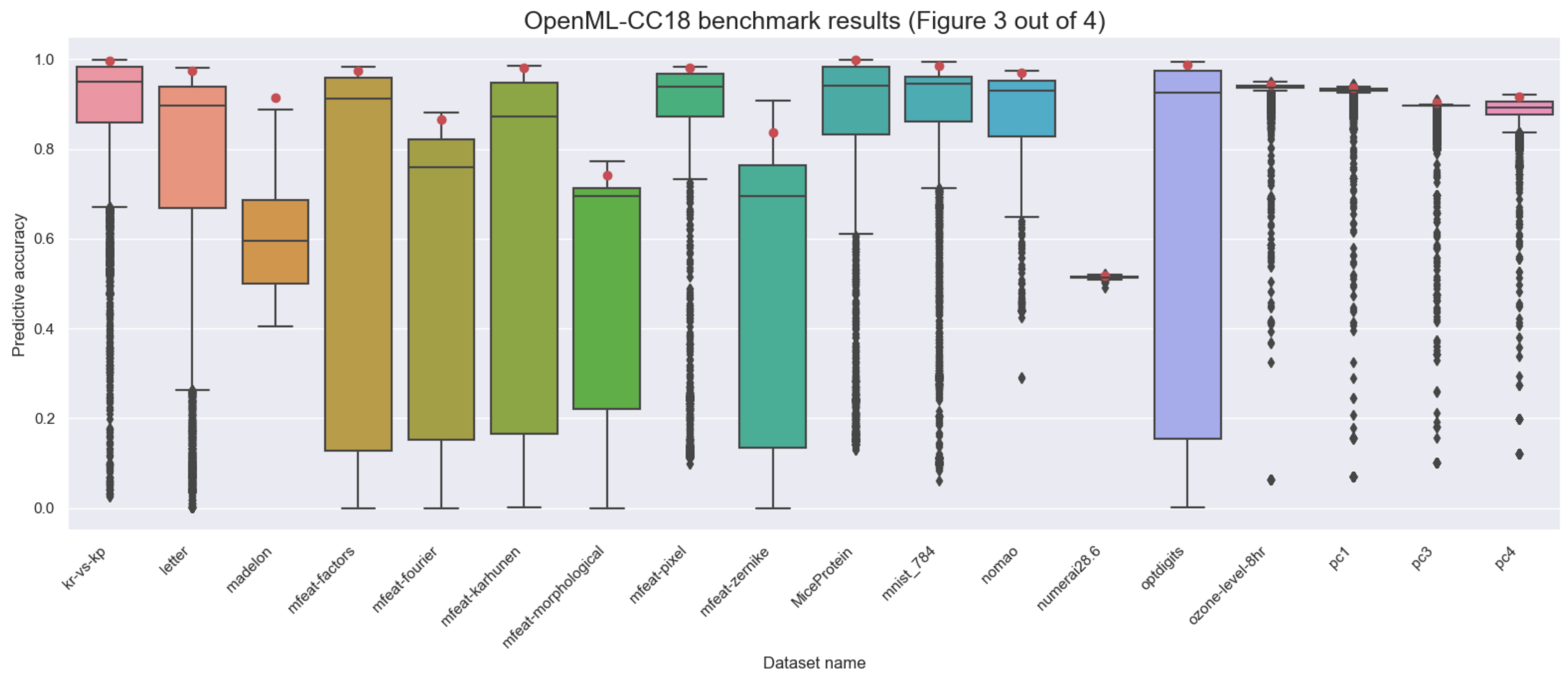


Figure 4.8: OpenML-CC18 benchmark result comparison (Figure 3 out of 4). Boxplots visualize distributions of predictive accuracies of all runs uploaded to OpenML. Red circles mark our accuracy score on datasets.

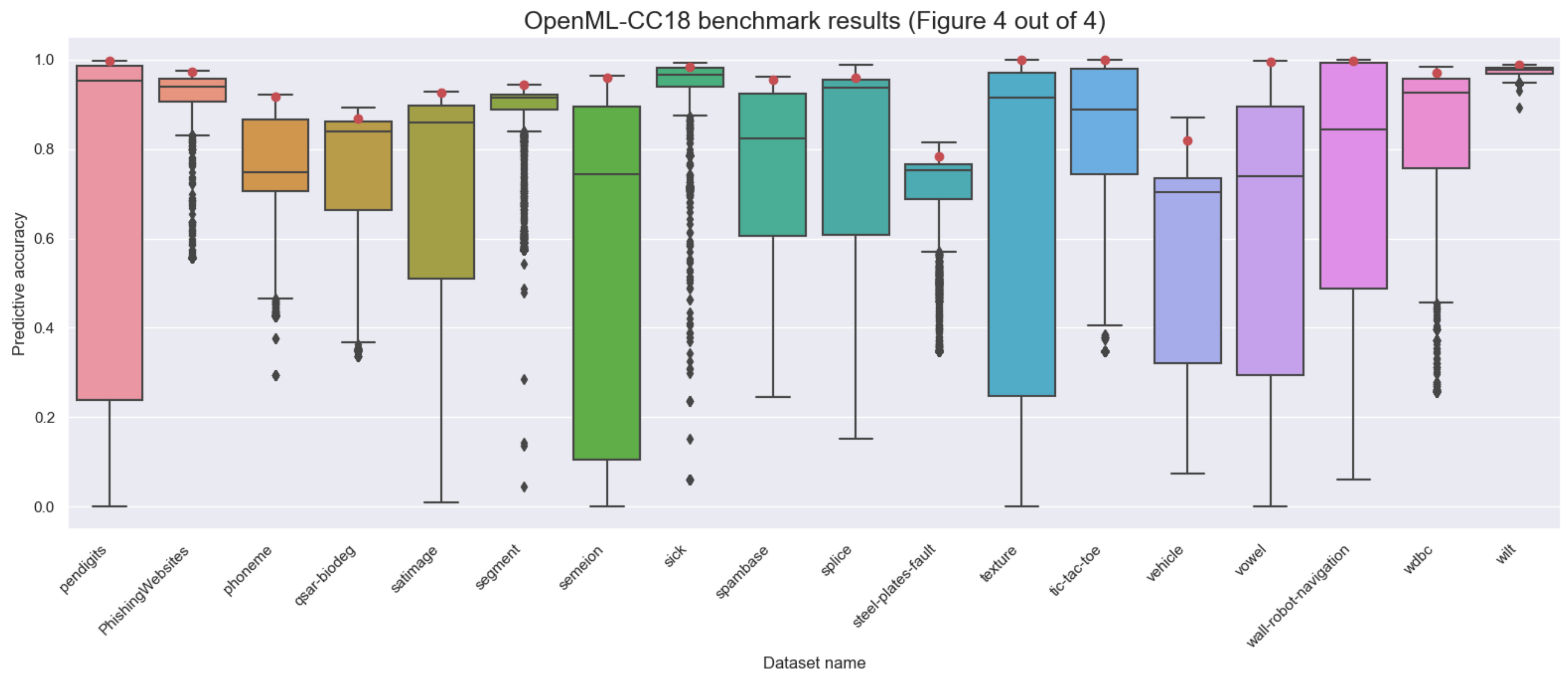


Figure 4.9: OpenML-CC18 benchmark result comparison (Figure 4 out of 4). Boxplots visualize distributions of predictive accuracies of all runs uploaded to OpenML. Red circles mark our accuracy score on datasets.

Conclusion

The main result of our work is the design of an AutoML system for automatic supervised learning pipeline optimization. Using the developmental GP, we created a general encoding of scikit-learn pipelines that converts a DAG pipeline into a tree representation.

The encoding was designed in an extensible way. The list of used methods can be easily extended with more ensembles and methods that comply to the scikit-learn API. We also implemented a tree individual that, compared to the DEAP tree individual, supports variable arity of nodes.

We introduced several bloat-reducing approaches. The grow method of the GP was modified to initialize less branched trees, which were more suitable for representation of pipelines. We also employed a weighted node selection, which prefers simpler methods. That way, we avoided nested ensembles which are computationally demanding, but do not significantly improve the score.

To decrease the running time of the optimization, we implemented some performance estimation strategies based on sampling. A sample of the input dataset was either created once per generation or per individual evaluation. The two strategies were then tested in the first experiment on two medium size dataset and one large dataset. The results of the experiment did not show any noticeable difference in the two approaches, but implied that by generating a sample for every evaluation, we increase the variance of results.

For the evolutionary optimization itself, we designed specific genetic operators. The subtree crossover and subtree mutation greatly alter the structure of the pipeline. The point mutation changes only one method of the pipeline while preserving the architecture. Finally, the node argument mutation changes a random hyperparameter of a random node, thus targeting the CASH problem.

To test whether the absence of a certain genetic operator influences the final result, we carried out the second experiment, where we turned the operators off one at a time. With this settings, we have done several runs on a small dataset and on a medium sized unbalanced dataset. The results of the experiments have shown that on the medium sized dataset, the hyperparameter mutation is essential for obtaining good results. On the second dataset, turning off the point mutation produced significantly better results. This behaviour is to be subject of future research. Finally, in all cases the results were better than the simple case where all genetic operators were turned off.

In the last experiment, we evaluated the performance on the system on the OpenML-CC18 benchmark suite. We ran the evolutionary optimization on each of the 72 datasets of the benchmark, while using one of the sampling strategies. Three best pipelines were evaluated and the maximum of the scores was chosen as the final output. We then compared our results with the statistic of runs uploaded to OpenML. Overall, on most datasets our system performed better than the upper quartile of OpenML runs or only slightly below it. In one case, our system outperformed all runs uploaded to OpenML.

In future extensions of our system, there are several concepts which are to be explored. As has been shown in the second experiment, the mutation of hyperparameters may significantly improve the results. As such, it would be suitable to employ a better optimization strategy that a simple selection of values from a list. One possibility would be to use the evolutionary strategies, which is a subfield of evolutionary algorithms particularly suited for continuous optimization. Another option is to create a Bayesian optimization method inspired by existing AutoML systems. Moreover, some additional methods may yield promising results, notably hill-climbing or mutation of multiple hyperparameters at once. As shown in experiment 2, it is necessary to further optimize the genetic operators. Again, more sophisticated methods could lead to interesting pipeline architectures. A related problem is the setting of weights used for initialization, which should be thoroughly explored.

To extend the methods used in the pipelines, a stacking estimator should be added to the primitive set, as it may lead to very promising results. Also, we should focus more on the data preprocessing methods, as users of this system still need to perform some initial encoding and imputation of missing data. Our system should be then compared with existing AutoML methods on suitable benchmark data.

Bibliography

Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics).* Springer-Verlag, Berlin, Heidelberg, 2006. ISBN 0387310738.

R.K. Bock, A. Chilingarian, M. Gaug, F. Hakl, T. Hengstebeck, M. Jiřina, J. Klaschka, E. Kotrč, P. Savický, S. Towers, A. Vaiciulis, and W. Wittek. 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*, 516(2):511 – 528, 2004. ISSN 0168-9002. doi: https://doi.org/10.1016/j.nima.2003.08.157. URL `http://www.sciencedirect.com/science/article/pii/S0168900203025051`.

Pavel Brazdil, Christophe Giraud-Carrier, Carlos Soares, and Ricardo Vilalta. *Metalearning: Applications to Data Mining.* Springer Publishing Company, Incorporated, 1 edition, 2008. ISBN 3540732624, 9783540732624.

Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

George Casella and Roger Berger. *Statistical Inference.* Duxbury Resource Center, June 2001. ISBN 0534243126.

François Chollet et al. Keras. `https://keras.io`, 2015.

Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960. doi: 10.1177/001316446002000104. URL `https://doi.org/10.1177/001316446002000104`.

Paulo Cortez, António Cerdeira, Fernando Almeida, Telmo Matos, and José Reis. Modeling wine preferences by data mining from physicochemical properties. *Decision Support Systems*, 47(4):547 – 553, 2009. ISSN 0167-9236. doi: https://doi.org/10.1016/j.dss.2009.05.016. URL `http://www.sciencedirect.com/science/article/pii/S0167923609001377`. Smart Business Networks: Concepts and Empirical Evidence.

Charles Darwin. *On the origin of species by means of natural selection, or, The preservation of favoured races in the struggle for life.* London, Murray, 1859.

Alex G. C. de Sá, Walter José G. S. Pinto, Luiz Otavio V. B. Oliveira, and Gisele L. Pappa. RECIPE: A grammar-based framework for automatically evolving classification pipelines. In James McDermott, Mauro Castelli, Lukas Sekanina, Evert Haasdijk, and Pablo García-Sánchez, editors, *Genetic Programming*, pages 246–261, Cham, 2017. Springer International Publishing. ISBN 978-3-319-55696-3.

K. Deb and H. Jain. An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part I: Solving problems with box constraints. *IEEE Transactions on Evolutionary Computation*, 18(4):577–601, Aug 2014. ISSN 1089-778X. doi: 10.1109/TEVC.2013.2281535.

K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *Trans. Evol. Comp*, 6(2):182–197, April 2002. ISSN 1089-778X. doi: 10.1109/4235.996017. URL `http://dx.doi.org/10.1109/4235.996017`.

A. E. Eiben and James E. Smith. *Introduction to Evolutionary Computing*. Springer Publishing Company, Incorporated, 2nd edition, 2015. ISBN 3662448734, 9783662448731.

Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural Architecture Search: A Survey. *arXiv e-prints*, art. arXiv:1808.05377, Aug 2018.

Andries P. Engelbrecht. *Computational Intelligence: An Introduction*. Wiley Publishing, 2nd edition, 2007. ISBN 0470035617.

Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Tobias Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'15, pages 2755–2763, Cambridge, MA, USA, 2015. MIT Press. URL `http://dl.acm.org/citation.cfm?id=2969442.2969547`.

Peter Flach. *Machine Learning: The Art and Science of Algorithms That Make Sense of Data*. Cambridge University Press, New York, NY, USA, 2012. ISBN 1107422221, 9781107422223.

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

Yoav Freund and Robert E. Schapire. Experiments with a new boosting algorithm. In *Proceedings of the Thirteenth International Conference on International Conference on Machine Learning*, ICML'96, pages 148–156, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc. ISBN 1-55860-419-7. URL `http://dl.acm.org/citation.cfm?id=3091696.3091715`.

Yoav Freund and Robert E Schapire. A decision-theoretic generalization of online learning and an application to boosting. *J. Comput. Syst. Sci.*, 55(1): 119–139, August 1997. ISSN 0022-0000. doi: 10.1006/jcss.1997.1504. URL `http://dx.doi.org/10.1006/jcss.1997.1504`.

Frédérique Gruau. *Neural Network Synthesis using Cellular Encoding and the Genetic Algorithm*. PhD thesis, Laboratoire de l'Informatique du Parallelisme, École Normale Superieure de Lyon, France, 1994. URL `ftp://ftp.ens-lyon.fr/pub/LIP/Rapports/PhD/PhD1994/PhD1994-01-E.ps.Z`.

Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren, editors. *Automated Machine Learning: Methods, Systems, Challenges*. Springer, 2018. In press, available at http://automl.org/book.

Joblib, 2019. Joblib, 2019. URL `https://joblib.readthedocs.io/en/latest/`. Accessed 2019-05-02.

Brian Alan Johnson, Ryutaro Tateishi, and Nguyen Thanh Hoan. A hybrid pansharpening approach and multiscale object-based image analysis for mapping diseased pine and oak trees. *International Journal of Remote Sensing*, 34(20):6969–6982, 2013. doi: 10.1080/01431161.2013.810825. URL `https://doi.org/10.1080/01431161.2013.810825`.

John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992. ISBN 0-262-11170-5.

John R. Koza, Forrest H. Bennett, David Andre, and Martin A. Keane. Evolutionary design of analog electrical circuits using genetic programming. In Ian C. Parmee, editor, *Adaptive Computing in Design and Manufacture*, pages 177–192, London, 1998. Springer London. ISBN 978-1-4471-1589-2.

Tomáš Křen, Martin Pilát, and Roman Neruda. Automatic creation of machine learning workflows with strongly typed genetic programming. *International Journal on Artificial Intelligence Tools*, 26:1–24, 2017.

Trang T. Le, Weixuan Fu, and Jason H. Moore. Scaling tree-based automated machine learning to biomedical big data with a dataset selector. *bioRxiv*, 2018. doi: 10.1101/502484. URL `https://www.biorxiv.org/content/early/2018/12/20/502484`.

D. D. Lucas, R. Klein, J. Tannahill, D. Ivanova, S. Brandon, D. Domyancic, and Y. Zhang. Failure analysis of parameter-induced simulation crashes in climate models. *Geoscientific Model Development*, 6(4):1157–1171, 2013. doi: 10.5194/gmd-6-1157-2013. URL `https://www.geosci-model-dev.net/6/1157/2013/`.

Hector Mendoza, Aaron Klein, Matthias Feurer, Jost Tobias Springenberg, and Frank Hutter. Towards automatically-tuned neural networks. In Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren, editors, *Proceedings of the Workshop on Automatic Machine Learning*, volume 64 of *Proceedings of Machine Learning Research*, pages 58–65, New York, New York, USA, 24 Jun 2016. PMLR. URL `http://proceedings.mlr.press/v64/mendoza_towards_2016.html`.

Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs (3rd Ed.).* Springer-Verlag, Berlin, Heidelberg, 1996. ISBN 3-540-60676-9.

Thomas M. Mitchell. *Machine Learning.* McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997. ISBN 0070428077, 9780070428072. URL `http://www.cs.cmu.edu/~tom/mlbook.html`.

Felix Mohr, Marcel Wever, and Eyke Hüllermeier. Ml-plan: Automated machine learning via hierarchical planning. *Machine Learning*, 107(8):1495–1515, Sep 2018. ISSN 1573-0565. doi: 10.1007/s10994-018-5735-z. URL `https://doi.org/10.1007/s10994-018-5735-z`.

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. ISBN 978-3-319-31204-0. doi: 10.1007/978-3-319-31204-0_9. URL `http://dx.doi.org/10.1007/978-3-319-31204-0_9`.

OpenML, 2019. OpenML-CC18 documentation, 2019. URL `https://docs.openml.org/benchmark/#openml-cc18`. Accessed 2019-05-12.

OpenML-API, 2019. OpenML API documentation, 2019. URL `https://docs.openml.org/APIs/`. Accessed 2019-05-12.

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

Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A Field Guide to Genetic Programming.* Lulu Enterprises, UK Ltd, 2008. ISBN 1409200736, 9781409200734.

Lior Rokach. Taxonomy for characterizing ensemble methods in classification tasks: A review and annotated bibliography. *Comput. Stat. Data Anal.*, 53(12): 4046–4072, October 2009. ISSN 0167-9473. doi: 10.1016/j.csda.2009.07.017. URL `http://dx.doi.org/10.1016/j.csda.2009.07.017`.

Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009. ISBN 0136042597, 9780136042594.

Robert E. Schapire. The strength of weak learnability. *Mach. Learn.*, 5(2): 197–227, July 1990. ISSN 0885-6125. doi: 10.1023/A:1022648800760. URL `https://doi.org/10.1023/A:1022648800760`.

scikit-learn, 2019. scikit-learn scorer interface, 2019. URL `https://scikit-learn.org/stable/modules/model_evaluation.html`. Accessed 2019-05-15.

N. Srinivas and Kalyanmoy Deb. Muiltiobjective optimization using nondominated sorting in genetic algorithms. *Evol. Comput.*, 2(3):221–248, September 1994. ISSN 1063-6560. doi: 10.1162/evco.1994.2.3.221. URL `http://dx.doi.org/10.1162/evco.1994.2.3.221`.

Gabriela Suchopárová. An AutoML system for workflow optimization, 2019. URL `https://github.com/gabrielasuchopar/genens`. Accessed 2019-05-02.

Chris Thornton, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Auto-WEKA: Automated selection and hyper-parameter optimization of classification algorithms. *CoRR*, abs/1208.3719, 2012. URL `http://arxiv.org/abs/1208.3719`.

Jan N. van Rijn. OpenML-CC18, 2019. URL `https://www.openml.org/s/99`. Accessed 2019-05-12.

Joaquin Vanschoren. Meta-learning: A survey. *CoRR*, abs/1810.03548, 2018. URL `http://arxiv.org/abs/1810.03548`.

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

D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, April 1997. ISSN 1089-778X. doi: 10.1109/4235.585893.

Quanming Yao, Mengshuo Wang, Hugo Jair Escalante, Isabelle Guyon, Yi-Qi Hu, Yu-Feng Li, Wei-Wei Tu, Qiang Yang, and Yang Yu. Taking human out of learning applications: A survey on automated machine learning. *CoRR*, abs/1810.13306, 2018. URL `http://arxiv.org/abs/1810.13306`.

Marc-André Zöller and Marco F. Huber. Survey on automated machine learning. *CoRR*, abs/1904.12054, 2019.

List of Algorithms

List of Figures

List of Tables