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

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date .... signature of the author

Dedication.

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: Abstract.

Keywords: Machine learning Evolutionary computing Meta-learning Workflows

Contents

Introduction

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 — AutoML. Finally, we dedicate two chapters to the heuristic optimization method — evolutionary computation and its subfield, genetic programming.

1.1 Machine learning

The field of machine learning 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 algorithm 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]) The character of the learning process varies with different machine learning problems. There are three main classes of tasks: *supervised*, *unsupervised* and *reinforcement* learning.

In case of supervised learning, the training data is a set of labelled examples and the task is to predict labels 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*. The continuous case is then called *regression*. In contrast to supervised learning, in unsupervised learning the training data is unlabelled. The key task is therefore 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 typically some history of previous actions and corresponding rewards.

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 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*'. A low-complexity model will not overfit, but a lot of errors will be made, thus introducing a certain 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 output. A balance can rarely be achieved in practice, hence it is often necessary to choose the side that is less harmful to the task. Another option is to use some of the ensemble methods described in 1.1.1. For example, bagging is a method of variance reduction, while boosting noticeably reduces the bias. More on this topic can be found in Flach's book. (Flach [2012, p. 93–94, 338])

1.1.1 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.

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, Shapire proved that a model is weakly learnable if and only if it is strongly learnable. [Schapire, 1990] 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 sections, 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 then used to generate predictions, which can be then *aggregated* by voting or averaging.

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. Moreover, there is a vast number of model and parameter combinations to choose from, not to mention model ensembles. Furthermore, it is necessary to preprocess the data and construct features for the learning process. All these aforementioned tasks require human expertise and are typically time-consuming. AutoML aims to automatize the process and make machine learning available to non-experts.

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 methods 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 linear acyclic graph and some parts of the flow may by cyclic, as can be seen in Figure 1.1. The term *pipeline* is also used for models which comprise several feature preprocessing methods and/or model ensembles.

The process can be decomposed into well-defined steps, which are depicted in Figure 1.1. The first two steps can be 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, feature extraction and feature selection. After that, a suitable machine learning model must be selected along with suitable hyperparameters; the latter is performed in the optimization step. Finally, the resulting model is evaluated, most often on a set of previously unseen *validation data*.

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

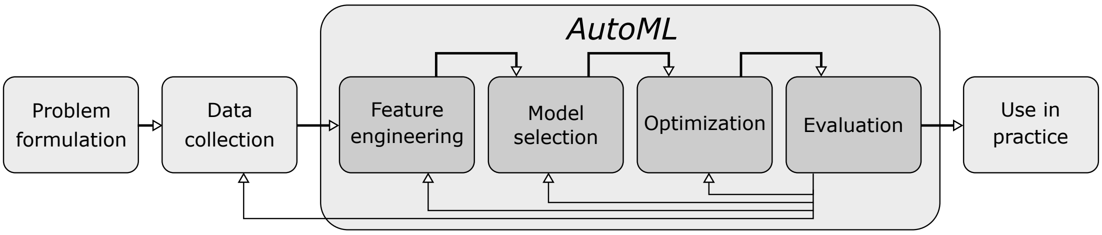


Figure 1.1: A typical machine learning workflow

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. As has already been mentioned, there is no silver-bullet method that would solve all machine learning problems. Furthermore, every method depends on some hyperparameters that greatly influence its performance. Combining this, we get a large search space of possible models. This problem is called] *combined algorithm selection and hyperparameter optimization problem* (CASH) and a formal definition can be found in [Thornton et al., 2012].

There are several limitations which complicate the search through this space. For large models (for example in deep learning) and large dataset, the learning process and evaluation take a lot of time. Moreover, the configuration space may be quite complex, with many continuous hyperparameters which also 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 a neural networks, which is currently a very popular area of research. It also needs to be considered in traditional machine learning when employing model ensembles. 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 could be very useful in the process, enabling to start with promising models instead of spending time on poor ones [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 is described 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 method are universal properties of data and the learning process itself. It is often employed in the design of pipelines or in neural architecture search, as it can significantly improve the performance and running time [Vanschoren, 2018].

Just as in case of the traditional learning — or also *base-learning* — 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 a several runs or many different tasks.

In the research area of model recommendation, which is one of the applications of metalearning, the training data is most frequently a history of previous runs. A suitable model is then chosen by examining which models were successful on similar tasks. Therefore, we need to accumulate some *metadata.* Another problem is that the input data may differ significantly and as such, the tasks usually cannot be compared directly. Thus, it is necessary to create so-called *metafeatures* which can be extracted even from substantially different data.

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

1.3 Evolutionary computing

Evolutionary computing is a heuristic 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 an evolutionary algorithm, the goal is to find the "best" solution to the given problem by optimizing a *objective function.* The term 'population' refers to a set of solutions encoded as chromosomes which represent the defining features of a particular solution. This corresponds to the genotype–phenotype relationship from genetics. The 'natural selection' can be then understood as a stochastic search through the space of possible chromosome values. (Engelbrecht [2007])

1.3.1 Evolutionary algorithms in detail

As can be seen in algorithm 1, a genetic algorithm should define a suitable *selection* method, *mutation* and/or *crossover* operators and a *fitness function*. The algorithm terminates when some *stopping condition* is met.

Selection The selection may be divided into two steps. The first is the *parent selection*, also called *mating selection* (line 8), 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]. An example of an elitist selection algorithm is NSGA-II, which is described 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 genomes. An example of the operators can be found in section 1.4.1. In the schema of the evolutionary algorithm (EA), 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. Most usually, 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. The structure of the genome is randomly changed in the hope of creating a more fit individual. Thus, it must be applied with care, as it is also possible that some good part may be 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, a objective function threshold or termination after no improvement is observed. ([Engelbrecht, 2007])

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($k/2$):
        $i_1, i_2 \longleftarrow$ 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 $P_o(n)$

    $P(n+1) \longleftarrow$ select $k$ individuals from $P_o(n)$

return $P(c)$
```

The advantage of genetic 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 $f_i(x), i = 1, \ldots, k$, the objective vector is defined as $f(x) = (f_1(x), f_2(x), \ldots, f_k(x))$.

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

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

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 2 (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 evolutionary computation to solve multi-objective problems, the algorithm needs to be modified. As not every 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 one 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). The niching factor is defined as

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

where

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

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 abovementioned problems, it also outperforms other elitist algorithms [Deb et al., 2002].

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 (*terminals*) and variables. Together, all possible functions and terminals form the *primitive set*. Every function of the set must have a well-defined arity value.

An extension of the genetic programming is *strongly typed GP*. It constraints the primitive set in such a way that every primitive has an output type and furthermore every function defines input types of its arguments.

The initialization step 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]

The fitness is usually computed by running the program and comparing the result with the desired output. It is also possible to apply genetic programming on multi-objective problems, where the second objective may be the running time of the problem or some other domain-specific property [Poli et al., 2008]. More about multi-objective optimization can be read in section 1.3.2.

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 well present two methods that are among the simplest and most used ones — *grow* and *full*.

In both cases, nodes are 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 grow method. Also, instead of one height limit, a range of values is used to introduce more diversity. [Poli et al., 2008]

Genetic operators The most common type of crossover is *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 *subtree mutation*. Just like in subtree crossover, a mutation point is randomly chosen. 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. In the case of strongly typed GP, both input an output types must match the types of the previous node. [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 responds 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.

More about the encoding and modifying operations will be presented in chapter 3.

2. Related work

In this section we present existing AutoML system. The examples are divided into two groups, where the first comprises of methods focusing mainly on simple models and the second of systems which enable more complex architectures. Finally we compare one of the systems (TPOT) with our approach.

Hyperparameter optimization methods The first attempt to solve the CASH problem 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, they support only a small number of model architectures. The algorithms used in Auto-WEKA are general enough, but the number of subestimators is limited to 5 and only base-learners can be used. Auto-sklearn abandoned ensembles altogether in the optimization process and 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.

There are only few systems that allow unlimited pipeline sizes. The system *RECIPE* uses grammar-based GP 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). However, none of these methods support more complex ensemble structures.

Comparison of TPOT with our system 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. TPOT also focuses more on the feature preprocessing step of the workflow, providing additional built-in transformers beyond the scikit-learn provided algorithms. 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 ours uses the score and evaluation time.

3. Our solution

In our solution we design an AutoML system for workflow optimization based on developmental genetic programming. Compared to existing systems it supports arbitrary-sized pipelines as well as complex ensemble structures. An overview of the process is shown in algorithm 2. In section 3.1 we describe how we apply the developmental GP to this problem (line 2). Evaluation of pipelines (line 3) 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  $individuals$ ⟵ run developmental GP on $d$ with $c$
3  $pipelines$ ⟵ Compile($individuals$)
4
5  return pipelines
```

3.1 Evolutionary optimization of pipelines

In this section, we describe the necessary components of the genetic algorithm. The process corresponds to the schema of a general genetic algorithm 1. A more detailed schema is presented in algorithm 3. Individuals of this particular EA are pipelines encoded as trees via developmental GP; the encoding is summarized in section 3.1.1. The initialization procedure (line 12) is explained in section 3.1.2. All reproduction operators (line 20) are presented in section 3.1.3. Finally, the selection and fitness are specified in section 3.1.4 (lines 16 and 38).

3.1.1 Individual encoding

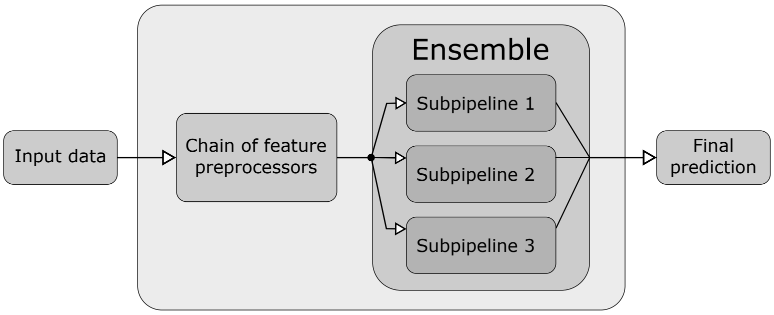


Figure 3.1: Schema of an example pipeline

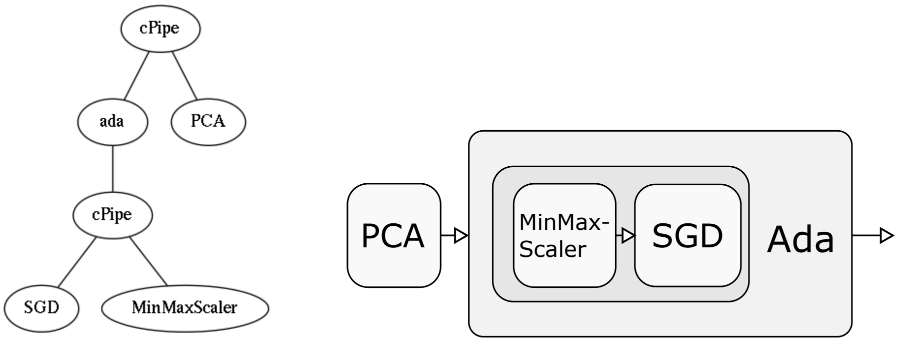
The individual encoding is one of the most important parts of this system. With ensembles and complex feature preprocessing methods like stacking or feature union, most of the pipelines become in fact directed acyclic graphs (Figure 3.1). Therefore, we cannot directly use the simple tree-based encoding.

Algorithm 3: Pipeline optimization — developmental GP

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

def evaluate:
    $pipe \longleftarrow$ 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

while $n < max\_gen$:

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

    /* 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 $P_o(n)$

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

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.



(a) Tree encoding

(b) Encoded pipeline

Figure 3.2: An example pipeline encoded to a tree individual

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. The left son 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 hyperparameter of every pipeline step are stored aside the nodes and are not depicted in the figures.

Table 3.1 lists all nodes used in the current implementation of our system. Input type is defined as a cartesian product of types, output type is a single type; terminals have only the output type defined. Output types of child nodes must match the input type of parent node. The list is extensible, it is possible to add a definion 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

As the initial population we grow n trees where every tree has a random height between 1 and overall maximum height. The tree is grown from root, which is either a ‘cPipe’ or ‘cPred’ node (Table 3.1). Then, nodes are inserted into the tree according to the input type of the parent. Before the height limit is reached (during the *growing phase*), both functions and terminals are inserted into the tree. Then, only terminals are inserted to keep the limit.

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. Therefore, all node types listed in Table 3.1 may be used during the growing phase.

Moreover, we define special terminal nodes which are used only in the last level (footnote 2). It is necessary to include them, as otherwise it would be impossible to finish the tree in one level, but they would force the trees to be very short if used during the growing phase.

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.

The process is as follows: every node is assigned to a group and each group has a well defined weight. When selecting a node n 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, n is selected by a simple random choice from g.

Variable arity Some nodes, e.g. ensemble nodes, may have a *variable arity*. This means that the actual arity is determined just when the node is about to be inserted to the tree. The arity is determined by an interval which may or may not have an upper bound. If the upper bound is not provided, it is usually limited by a global arity limit to avoid bloat of the trees.

Method hyperparameters Every node has a list of possible values per hyperparameter associated with it. During selection, the actual values are randomly

Table 3.1: Nodes representing modifying operations

Node[1]	**In type**[2]	**Out type**[3]	**Operation**
cPipe	$ens \times data$	out	Create pipeline with a preprocessor chain and a predictor
cPred	ens	out	Create pipeline only with a predictor
cData	$featsel \times scale$	$data$	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.* $n \in < 1, max\_n >$*)*

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

selected from every list. The validity is not verified in this phase, instead it is handled in the evaluation phase 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. The mutation operators will be described in more detail.

Subtree mutation 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.

Node swap mutation In this type of mutation, a randomly chosen node is replaced with a new node. Both output and input types must match; if the new node supports variable arity, all input types of the old node must satisfy the bounds. For this type of mutation, any lower bound must be greater than zero.

Node argument mutation Mutates a hyperparameter of a random node — chooses a new value from the list of possible values. This method has many possible extensions which are more described in section **??** (future work).

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 — evaluation score and logarithmized evaluation time. The enviromental selection is done via NSGA-II and the parental 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 3.2, 3.3 and 3.4 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.

Table 3.2: 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 3.3: 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 3.4: 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. 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 k-fold cross-validation, but it is also possible to specify a separate validation set for scoring.

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, while the latter has proved to be more effective. A possible explanation is that if we generate only one sample per generation, we cannot ensure that it is representative enough. Thus, the individuals which score particularly well on this sample may not generalize well. The second approach suffers from this problem as well, but since we generate a sample per evaluation, every individual has a fair chance of getting a good sample. Possible improvements of this approach are presented in section ??.

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 3.3).

The components of the genetic 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].

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)

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 the 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. [OpenML-CC18]

4.1 Sampling strategies

In section 3.2.2 we presented two different performance estimation strategies aimed at reducing the evaluation time. Both of the 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 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), only few instances of the 'w' class are present (261) [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. [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 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 gen.	15
crossover pb.	0.5
subtree mutation pb.	0.3
hyperparam. mutation pb.	0.6
node mutation pb.	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. 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.

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

	minimum	median	confidence interval	maximum
per_gen	0.8645	0.8792	(0.8773, 0.8811)	0.882
per_ind	0.8720	0.8795	(0.8772, 0.8819)	0.882

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

	minimum	median	confidence interval	maximum
full	0.8695	0.8798	(0.8736, 0.8861)	0.8908
per_gen	0.8684	0.8730	(0.8699, 0.8760)	0.8878
per_ind	0.8661	0.8736	(0.8708, 0.8764)	0.8877

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, with relevant statistics of the boxplots listed in Tables 4.2, 4.3, 4.4. The middle line represents the median, while notches of the boxplot 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.

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

	minimum	median	confidence interval	maximum
full	0.5524	0.7059	$(0.6491, 0.7627)$	0.7097
per_gen	0.5295	0.5391	$(0.5372, 0.5409)$	0.5518
per_ind	0.5340	0.5380	$(0.5343, 0.5417)$	0.5454

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.

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

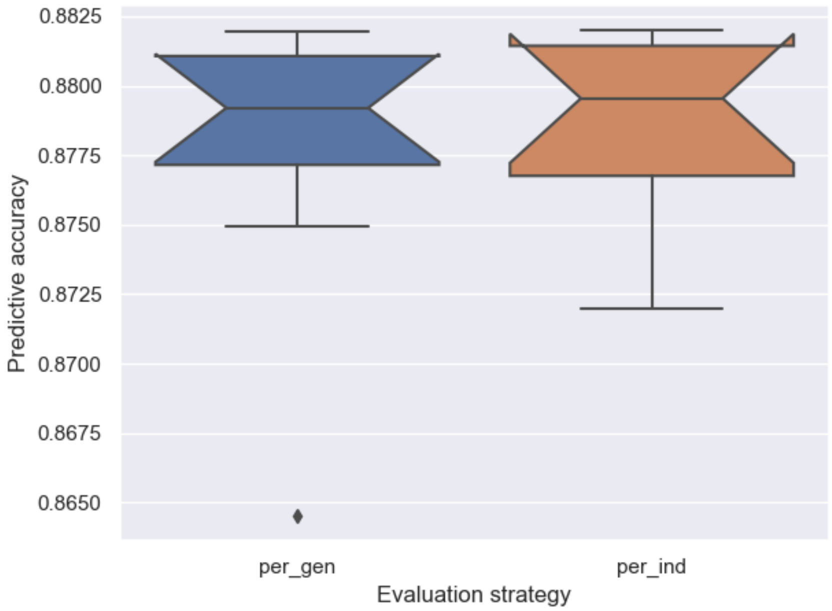


Figure 4.1: Comparison of strategies on the magic dataset

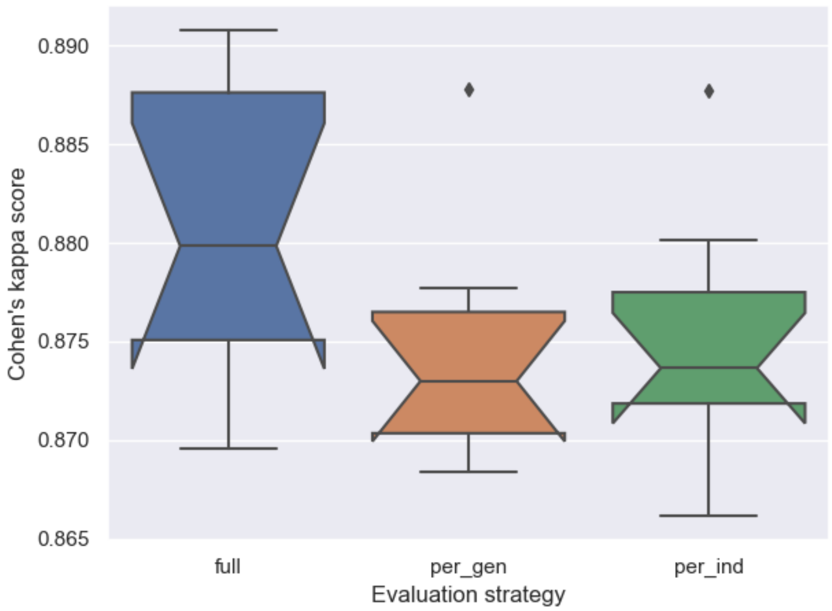


Figure 4.2: Comparison of strategies on the wilt dataset

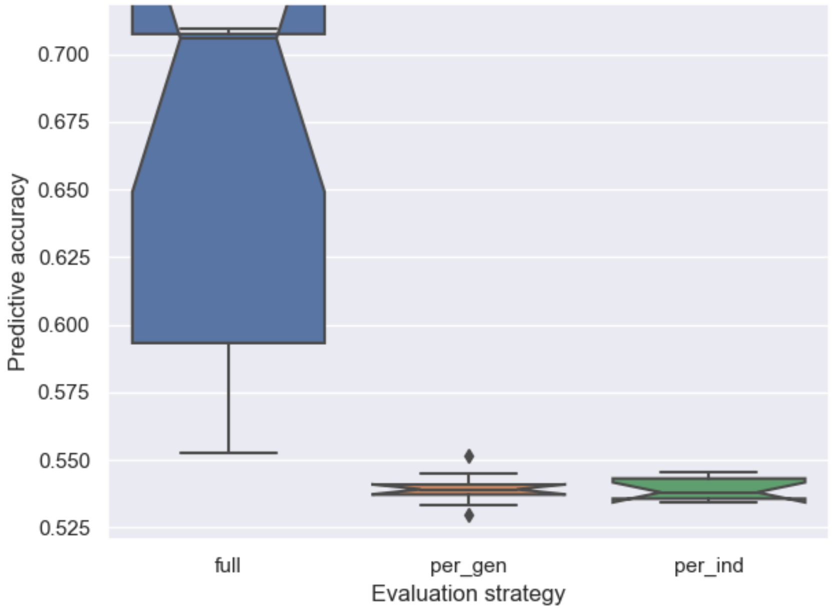


Figure 4.3: Comparison of strategies on the wine-quality-white dataset

- 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 — node swap, subtree mutation and hyperparameter mutation. There will be following sub-experiments:

- **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. 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 features. For fitness computation we used the 5-fold cross-validation. Final evaluation and data collection was the same as in the previous experiment.

Table 4.5: System hyperparameters for the genetic operator experiment

Hyperparameter	**Value**
population size	200
maximum gen.	15
crossover pb.	0.5
subtree mutation pb.	0.3
hyperparam. mutation pb.	0.6
node mutation pb.	0.3
maximum tree height	5
timeout per method	7 minutes
group weights	*default*

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.

In case of the wilt dataset, only 'no-arg' did not introduce any improvement when compared to results from 'nothing'. This probably means that the hyperparameter mutation is essential on this dataset — in fact, the best pipelines usually contained a SVC, which relies heavily on hyperparameter optimization. [Feurer et al., 2015]

TBD why is 'no-node' better than the others?

The confidence intervals of medians of ‘nothing’ and other methods, except ‘no-node’ 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
all	0.8658	0.8749	$(0.8711, 0.8786)$	0.8820
no-arg	0.8526	0.8694	$(0.8684, 0.8703)$	0.8794
no-cx	0.8499	0.8748	$(0.8715, 0.8781)$	0.8875
no-node	0.8664	0.8747	$(0.8721, 0.8772)$	0.8864
no-subtree	0.8675	0.8738	$(0.8706, 0.8770)$	0.8850
nothing	0.8434	0.8659	$(0.8585, 0.8734)$	0.8787

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

	minimum	median	confidence interval	maximum
all	0.9112	0.9167	$(0.9135, 0.9200)$	0.9280
no-arg	0.9149	0.9159	$(0.9140, 0.9177)$	0.9296
no-cx	0.9111	0.9168	$(0.9152, 0.9184)$	0.9242
no-node	0.9149	0.9223	$(0.9199, 0.9247)$	0.9259
no-subtree	0.9112	0.9177	$(0.9155, 0.9200)$	0.9316
nothing	0.9055	0.9149	$(0.9133, 0.9165)$	0.9185

4.3 OpenML-CC18 benchmarking suite

The OpenML-CC18 suite is a set of classification tasks created for the purpose of practical benchmarking. It contains 72 datasets available at OpenML [OpenML-CC18], where every dataset is associated with one task — a 10-fold cross-validation. The datasets included in the suite have been frequently used in recently published benchmarks and also satisfy several criteria, like a limit on instance count. The full list of criteria can be found in the OpenML documentation [OpenML]. A task is moreover associated with several runs, where each run contains data about the used model along with its hyperparameters and resulting evaluations. A model without specified hyperparameters is called ‘flow’ in the OpenML API.

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. 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 for all of the benchmark runs. Although we showed in section 4.1 that the per-ind method had a greater variance,

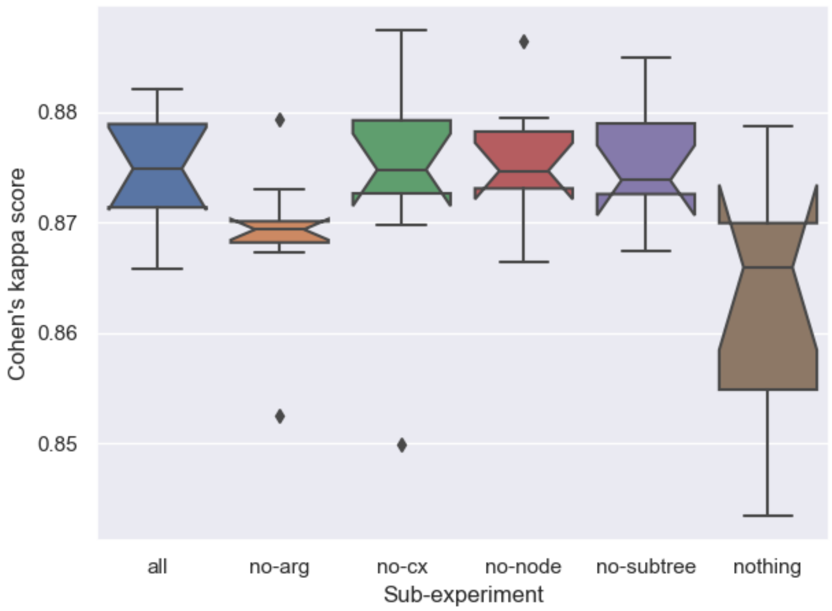


Figure 4.4: Test of genetic operators on the wilt dataset

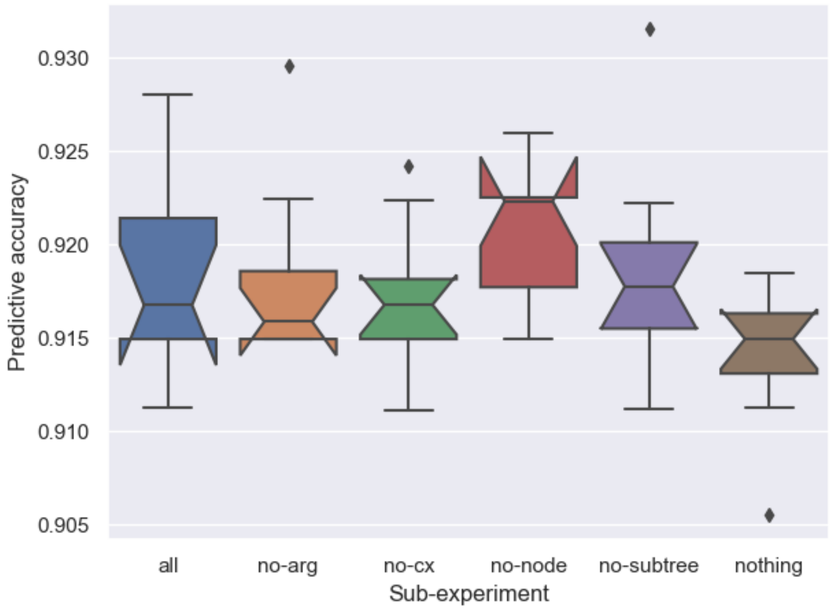


Figure 4.5: Test of genetic operators on the climate dataset

it was used here as this experiment preceded the sample experiment. For fitness computation, we used 5-fold cross-validation on a sample. Final evaluation was 10-fold cross-validation on the full dataset as specified by the task definition.

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.

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 [OpenML-CC18, OpenML].

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. This was however caused by the fact that our system contains the kNearestNeighborClassifier (as seen in Table 3.2.1), which was particularly suitable for this task but OpenML did not contain any run of this method on madelon.

The datasets which were problematic for our system were notably the *jungle_chess* [1] dataset, and *CIFAR-10*. 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, most of the runs were from flows representing keras models. [Chollet et al., 2015] Moreover, there were only 76 runs uploaded to the OpenML. As such, it is not possible to compare our model with other traditional methods (not using deep learning).

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

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

Row count limit	**Sample size**
1000[1]	0.5
5000	0.25
10000	0.1
20000	0.05
large[2]	0.02
large	0.01

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

[2] *Large datasets with less than 10 features*

Table 4.9: System hyperparameters for the OpenML-CC18 experiment

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

Our system also performed less well on small datasets and image-recognition tasks. The former may have happened due to overfitting, the latter because of the absence of tuned more complex models. Both of this problems are to be targeted in possible extensions of our system.

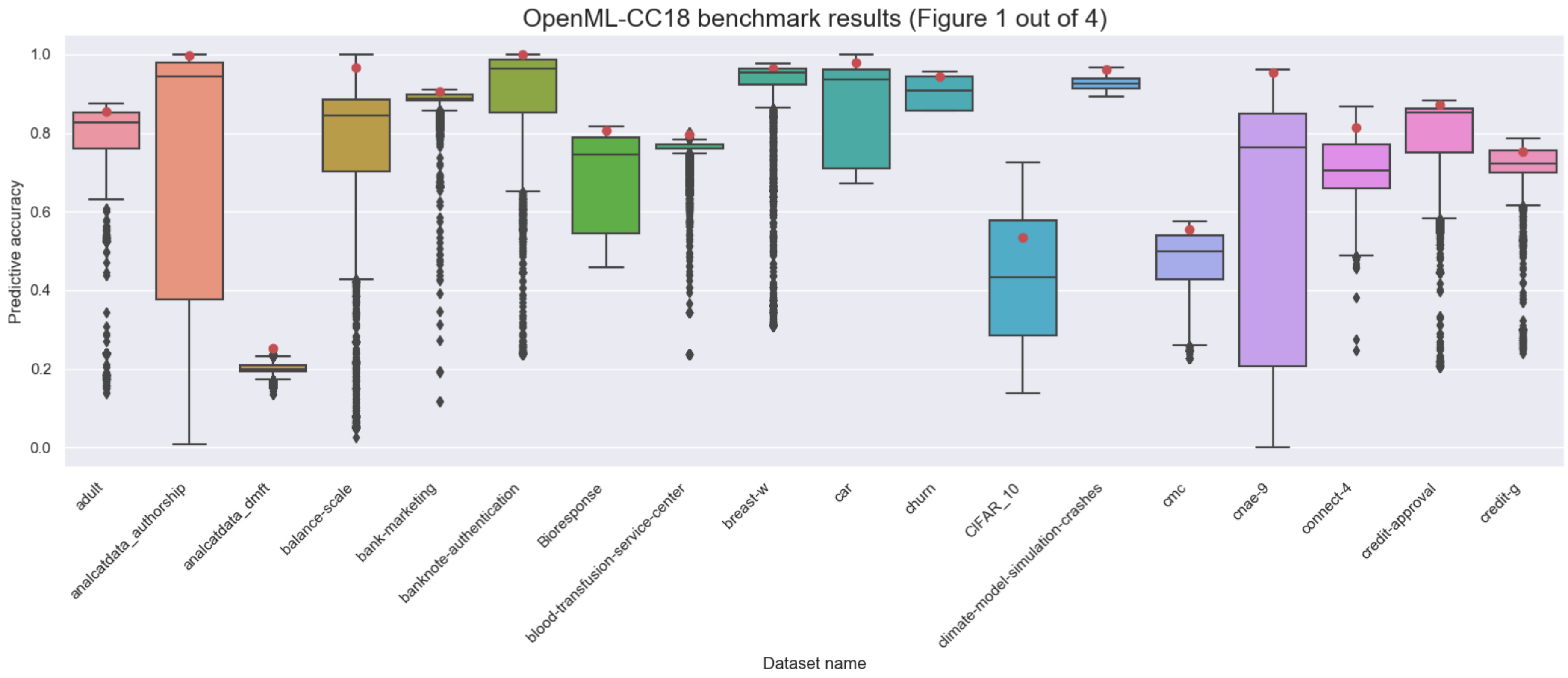


Figure 4.6: OpenML-CC18 benchmark result comparison (Figure 1 out of 4). Boxplots visualize distributions of predictive accuracies of all runs uploaded to OpenML. Red circles mark our accuracy score on datasets.

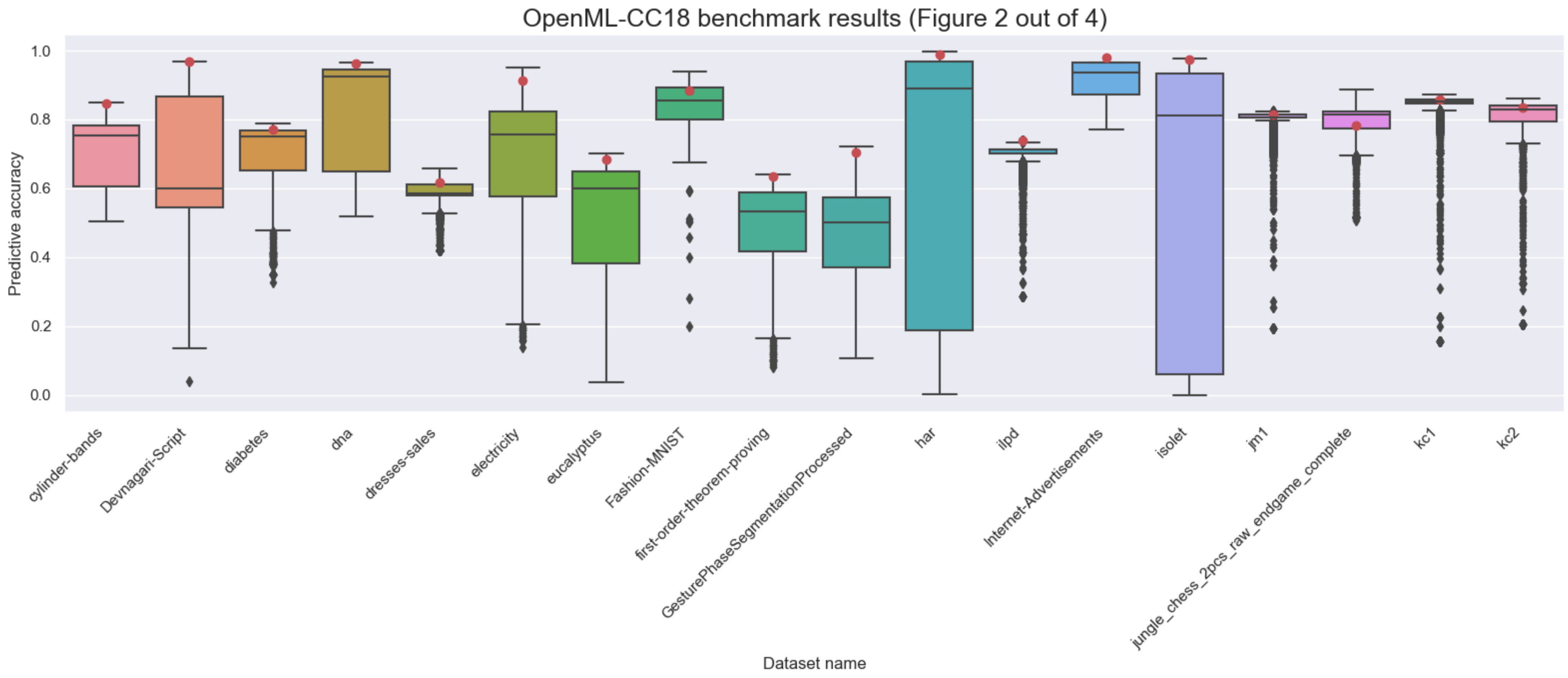


Figure 4.7: OpenML-CC18 benchmark result comparison (Figure 2 out of 4). Boxplots visualize distributions of predictive accuracies of all runs uploaded to OpenML. Red circles mark our accuracy score on datasets.

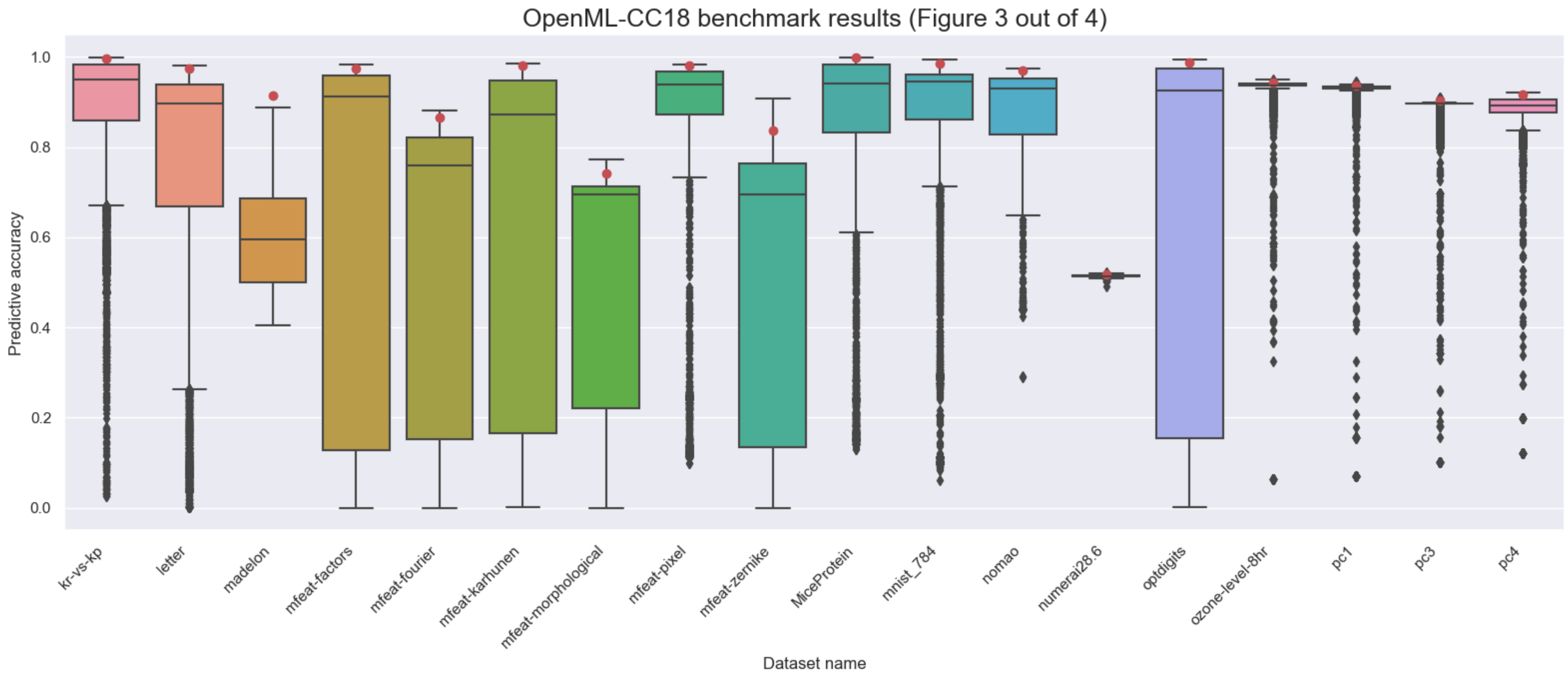


Figure 4.8: OpenML-CC18 benchmark result comparison (Figure 3 out of 4). Boxplots visualize distributions of predictive accuracies of all runs uploaded to OpenML. Red circles mark our accuracy score on datasets.

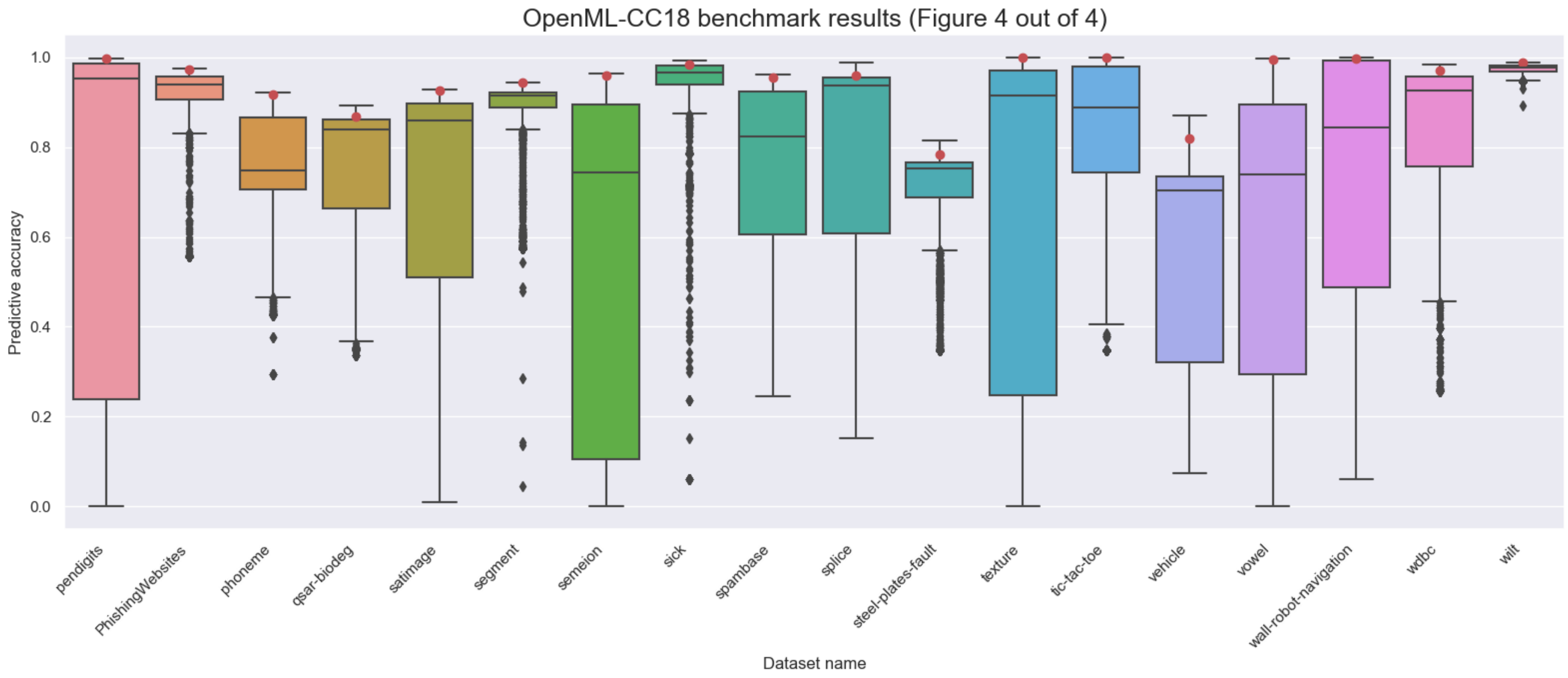


Figure 4.9: OpenML-CC18 benchmark result comparison (Figure 4 out of 4). Boxplots visualize distributions of predictive accuracies of all runs uploaded to OpenML. Red circles mark our accuracy score on datasets.

Conclusion

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.

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, 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 on-line 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. 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.

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. OpenML-CC18 documentation, 2019. URL `https://docs.openml.org/benchmark/#openml-cc18`. Accessed 2019-05-09.

OpenML-CC18. OpenML-CC18 web page, 2019. URL `https://www.openml.org/s/99`. Accessed 2019-05-09.

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`.

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`.

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`.

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

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`.

List of Algorithms

List of Figures

List of Tables

List of Abbreviations