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

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.

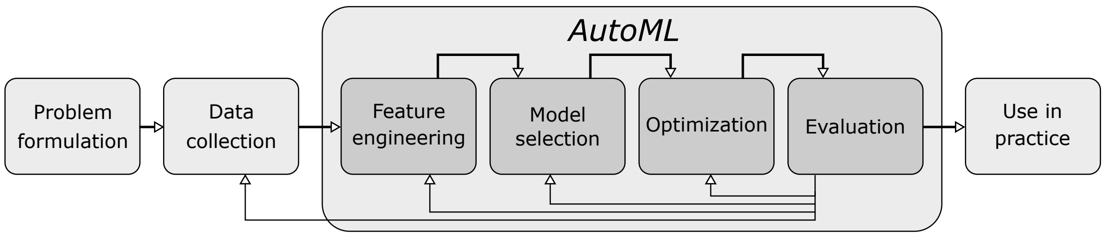

Figure 1.1: A typical machine learning workflow

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. 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. More on this subject is discussed in the following chapters.

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])

exploration vs exploitation...

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 7), and the second is called *environmental selection* or *recombination* (line 15). 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, reproduction operators are applied on parents selected by the mating selection, as can be seen on lines 9 and 11.

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
1
2  $P(0) \longleftarrow$ population of size $k$
3  while $c$ is not met:
4      for individual ind:
5          compute fitness $f(ind)$
6      for $i$ in range($k/2$):
7          $i_1, i_2 \longleftarrow$ select two individuals from $P(n)$
8          if $p_{cx}$:
9              $crossover(i_1, i_2)$
10         if $p_{mut}$ for $k = 1, 2$:
11             $mutation(i_k)$
12
13         add $i_1, i_2$ to offspring population $P_o(n)$
14
15     $P(n+1) \longleftarrow$ select $k$ individuals from $P_o(n)$
16
17 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*

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

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 $PF^* = \{\, f = (f_1(x^*), \ldots f_k(x^*)) \mid x^* \in P^*) \,\}$.*

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 above-mentioned 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. The *cell* 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

2.1 Examples of methods solving CASH

- Auto-WEKA — uses Bayesian optimization to search in space of WEKA algorithm implementations. The property of the selected model is converted to a hyperparameter, induces several 'conditional' hyperparameters (e.g. some are present only if the method is an ensemble,...). Limits ensemble subestimator count to 5, does not allow to have ensembles as ensemble subestimators. One feature preprocessing. First defined the CASH problem.

-

2.1.1 Pipelines of arbitrary size

- TPOT

- the hierarchical networks

3. Our solution

Conclusion

Bibliography

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

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.

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.

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.

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

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.

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.

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.

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`.

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`.

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

A. Attachments

A.1 First Attachment