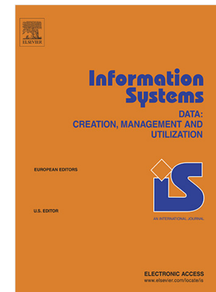
Journal Pre-proof



Two-stage optimization for machine learning workflow

Alexandre Quemy

PII: S0306-4379(19)30535-6
DOI: https://doi.org/10.1016/j.is.2019.101483
Reference: IS 101483

To appear in: *Information Systems*

Received date : 30 June 2019
Revised date : 16 October 2019
Accepted date : 3 December 2019

Please cite this article as: A. Quemy, Two-stage optimization for machine learning workflow, *Information Systems* (2019), doi: https://doi.org/10.1016/j.is.2019.101483.

This is a PDF file of an article that has undergone enhancements after acceptance, such as the addition of a cover page and metadata, and formatting for readability, but it is not yet the definitive version of record. This version will undergo additional copyediting, typesetting and review before it is published in its final form, but we are providing this version to give early visibility of the article. Please note that, during the production process, errors may be discovered which could affect the content, and all legal disclaimers that apply to the journal pertain.

© 2019 Elsevier Ltd. All rights reserved.

Journal Pre-proof

Declaration of interests

☒ The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

☐The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:

Journal Pre-proof

Journal Pre-proof

Highlights - Two-stage Optimization for Machine Learning Workflow

- The importance of optimizing data pipeline comparatively to hyperparameter tuning is studied to show data pipeline are often more important than hyperparameter tuning
- A two-stage optimization process is proposed to speed-up the search for a machine learning workflow
- This process is empirically validated with several policy to allocate time.
- The iterative and adaptive policies are more robust than usual policies and allow to find a better configuration faster
- A new metric to determine if a data pipeline is independent from the algorithm is provided and empirically validated.

Journal Pre-proof

Manuscript File

Journal Pre-proof

ed References

Two-stage Optimization for Machine Learning Workflow

Alexandre Quemy

IBM Krakow Software Lab, Cracow, Poland

Faculty of Computing, Poznań University of Technology, Poznań, Poland

Abstract

Machine learning techniques play a preponderant role in dealing with massive amount of data and are employed in almost every possible domain. Building a high quality machine learning model to be deployed in production is a challenging task, from both, the subject matter experts and the machine learning practitioners.

For a broader adoption and scalability of machine learning systems, the construction and configuration of machine learning workflow need to gain in automation. In the last few years, several techniques have been developed in this direction, known as AutoML.

In this paper, we present a two-stage optimization process to build data pipelines and configure machine learning algorithms. First, we study the impact of data pipelines compared to algorithm configuration in order to show the importance of data preprocessing over hyperparameter tuning. The second part presents policies to efficiently allocate search time between data pipeline construction and algorithm configuration. Those policies are agnostic from the metaoptimizer. Last, we present a metric to determine if a data pipeline is specific or independent from the algorithm, enabling fine-grain pipeline pruning and meta-learning for the coldstart problem.

Key words: data pipelines, hyperparameter tuning, AutoML, CASH

1. Introduction

In practical machine learning, data are as important as algorithms. Algorithms received a lot of interest in hyperparameter tuning methods [1, 2],

Email address: aquemy@pl.ibm.com (Alexandre Quemy)

Journal Pre-proof

Journal Pre-proof

that is to say, the art of adjusting parameters that are not dependent on the instance data. Contrarily, data pipeline construction and configuration received little interest. For instance, the platform OpenML[1] offers a collection of datasets and tools to share reproducible machine learning experiments. However, many datasets are already preprocessed and it is difficult to know what transformations already have been applied to the data. Moreover, in 2009, [3] notices that algorithm hyperparameter tuning is performed in 16 out of 19 selected publications while only 2 publications study the impact of data preprocessing. A decade later, this proportion still seems to hold. This can probably be explained by the fact that the research community mainly uses ready-to-consume datasets, hence occulting de facto this problematic. In practice however, raw data are rarely ready to be consumed and must be transformed by a carefully selected sequence of preprocessing operations.

In fact, building a high quality machine learning model to be deployed in production is a challenging task that is time consuming and computationally demanding. The usual machine learning workflow described by Figure 1 is broken down into two parts:

1. finding the correct sequence of data transformations such that the dataset is consumable by a machine learning algorithm,

2. selecting the proper machine learning algorithm and its hyperparameters, such that the model provides a good generalization w.r.t. a given performance metric.

Figure 1: Typical machine learning workflow

On one hand, there are plenty of reasons that can explain why a data source cannot be used directly and require preprocessing: too many variables, imbalanced dataset, missing values, outliers, noise, specific domain restriction

[1] https://www.openml.org

Journal Pre-proof

of the algorithms, etc. On the other hand, data preprocessing has a huge impact on the model performances [3–5].

The data pipeline depends both on the data source and the algorithm such that there is no universal pipeline that can work for every data source and every algorithm [6]. The data pipeline is usually defined by trial and error, using the experience of data scientists and the expert knowledge about the data. This step is so crucial it may represent up to 80% of data scientist time [7].

The techniques to automate the construction of machine learning workflows are called AutoML. Broadly speaking, AutoML consists in solving the following black-box optimization problem: given a dataset,

$$\text{Find } \boldsymbol{\lambda}^* \in \underset{\boldsymbol{\lambda} \in \boldsymbol{\Lambda}}{\arg\max}\ \mathcal{F}(\boldsymbol{\lambda}), \tag{1}$$

where $\boldsymbol{\Lambda}$ is the space of machine learning configuration, and $\mathcal{F}(\boldsymbol{\lambda})$ the performance of the model learned over the dataset using the configuration $\boldsymbol{\lambda}$.

This paper presents a two-stage optimization approach to solve the AutoML problem. Specifically, we are interested in understanding the respective impact of the data pipelines and the algorithm configurations on the model performances. Additionally, we study the best allocation of time between the two phases of the machine learning workflow.

To the best of our knowledge, most of AutoML systems tackle the problem by aggregating the data pipeline operators, the algorithm portfolio and their respective configuration space into a single gigantic search space (corresponding to $\boldsymbol{\Lambda}$ in Eq. (1)). In the approach presented in this paper, we divided the search for a solution into two steps, namely the data pipeline construction and configuration, and the algorithm selection and configuration.

Under the assumption that the two steps are roughly independent, the two-stage optimization brings the two following advantages:

1. by splitting the search space into two smaller ones, it speeds up the overall optimization process,
2. it is possible to statistically assess if a data pipeline is specific to an algorithm or rather *universal* w.r.t. the dataset, enabling meta-learning at a lower granularity level (see Section 7).

The paper is an extension of the preliminary work [8] that mainly focused on the influence of data pipelines on machine learning performances. The

Journal Pre-proof

novel work includes a more formal definition of the search space, pipeline prototype and the main problem (Section 3), the whole two-stage optimization process (Section 4) and its evaluation (Section 7), and an extension of the experiment of [8] to include also hyperaparameter tuning (Section 5).

The rest of the paper is organized as follows: in Section 2, we present work related to AutoML and discuss the limitations of current approaches. A reformulation of CASH is proposed in Section 3. The two-stage optimization process is described in Section 4, followed by an experimental validation presented in Section 5 and discussed in Section 6. Finally, Section 7 presents an indicator to evaluate the dependency of a data pipeline to the algorithm.

2. Related work

In this section, we introduce the main problem the AutoML community focuses on, namely Combined Algorithm Selection and Hyperparameter-tuning (CASH). After presenting the different approaches to solve it in Section 2.1, we discuss in Section 2.5, why CASH is a difficult problem, why its formulation might not be the most suitable and why current tools are not satisfying w.r.t. to the way machine learning practitioners work.

For a broader review on AutoML, we refer the reader to [1, 2].

2.1. CASH *problem*

The learning problem consists in finding or constructing an approximation of an unknown function $f : \mathcal{X} \to \mathcal{Y}$. A *learning* algorithm A maps a set of training points $\{d_i\}_{i=1}^n$ with $d_i = (\mathbf{x}_i, y_i) \in \mathcal{X} \times \mathcal{Y}$ to $\mathcal{Y}^{\mathcal{X}}$. A learning algorithm A is parametrized by some hyperparameters $\boldsymbol{\lambda} \in \boldsymbol{\Lambda}$ that modify the way the algorithm $A_{\boldsymbol{\lambda}}$ learns. Each hyperparameter λ_i belongs to a space Λ_i and $\boldsymbol{\Lambda}$ is a subset of the cross-product of each domain, i.e. $\boldsymbol{\Lambda} \subset \Lambda_1 \times \ldots \times \Lambda_n$. In general, Λ can be more structured (conditional tree, directed acycle graph,...).

CASH is formulated as the following optimization problem:

Definition 2.1 (Combined Algorithm Selection and Hyperparameter-tuning)**.** Given a portfolio of algorithms $\mathcal{A} = \{A^{(1)}, \ldots, A^{(m)}\}$ with associated hyperparameter spaces $\boldsymbol{\Lambda}^{(1)}, \ldots, \boldsymbol{\Lambda}^{(m)}$, the Combined Algorithm Selection and Hyperparameter Optimization (CASH) problem is defined by:

$$A^*_{\boldsymbol{\lambda}^*} = \operatorname*{arg\,min}_{A^{(j)} \in \mathcal{A}, \boldsymbol{\lambda} \in \boldsymbol{\Lambda}^{(j)}} \frac{1}{k} \sum_{i=1}^{k} \mathcal{L}(A^{(j)}_{\boldsymbol{\lambda}}, \mathcal{D}^{(i)}_{\text{train}}, \mathcal{D}^{(i)}_{\text{test}}), \qquad \text{(CASH)}$$

Journal Pre-proof

where $\mathcal{L}$ is a loss function (e.g. error rate) obtained on the test set by the model learned by algorithm A parametrized by $\boldsymbol{\lambda}$ over the training set.

2.2. Black-box optimization and surrogate learning

The most basic technique for hyperparameter tuning is a grid search or factorial design [9] which consists in exhaustively testing parameter configurations on a grid. In practice, this approach is computationally intractable. An efficient alternative, called Randomized search [10], consists in testing configurations (pseudo)randomly until a certain budget is exhausted.

Beyond those naive approaches, the most prominent approach to tackle CASH consists in iteratively building an approximation, called *surrogate model*, of the optimization objective.

Recall the black-box optimization problem (1) and the optimization objective $\mathcal{F}$. At step t, surrogate model $\hat{\mathcal{F}}$ is learned from the history $\{\boldsymbol{\lambda}_k, \mathcal{F}(\boldsymbol{\lambda}_k)\}$ for $k = 1...t$ of previously selected configurations and associated performances. An acquisition function is used to determined the most promising configuration for $\boldsymbol{\lambda}^*_{t+1}$.

The difference between the various surrogate approaches lies into the model space assumption and the acquisition function. Sequential Model-based Algorithm Configuration (SMAC) [11, 12] is based on Random Forest, so are frameworks based on SMAC such as AutoWeka [13, 14] or AutoSklearn [15]. Hyperopt [16] uses a Tree-structured Parzen Estimator (TPE) while Spearmint [17] is based on Gaussian processes (GP).

An acquisition function is used to determined the next configuration to be sampled. Most of those functions are based on Bayesian optimization [18, 19]. One popular strategy is to select $\boldsymbol{\lambda}_{k+1}$ such that it maximizes the expected improvement [20].

As an alternative to Bayesian optimization, [21] proposes to use Monte-Carlo Tree Search to iteratively explore a tree-structured search space while pruning the less promising configurations.

2.3. Multi-fidelity optimization

The black-box optimization problem (1) is expensive: an iteration means preprocessing the whole dataset through the pipeline and then training a model, often many times as cross-validation is preferred to validate the result. Multi-fidelity optimization focuses on decreasing the computational cost by using large number of *cheap* low-fidelity evaluations. For this, several approaches have been considered.

Journal Pre-proof

Extrapolating the model learning curve from the available training data provides an early stop criterion that reduces the computation time [22, 23].

Bandit-based algorithms such as Successive halving [24] or Hyperband [25] find configurations by greedily allocating more budget to promising configurations. For instance, Hyperband randomly selects configurations and iteratively removes unpromising candidates while increasing the budget for the promising ones.

In [26], the authors uses a genetic algorithm to sample a subset of representative input vectors in order to speed-up the model training while increasing the model performances. Genetic algorithms are also used to search for the whole pipeline as in Tpot[27] or AutoStacker[28].

Last, despite a small configuration space, ReinBo[29], a reinforcement learning approach, has been shown to outperform TPE, AutoSklearn and Tpot on many datasets.

2.4. Meta-learning and warm-start

Another research direction consist predicting and recommending good pipelines or operators for a given dataset or task. Referred to as meta-learning, it is particularly important for Bayesian optimization, extensively used to solve CASH, since the quality of the results is conditioned by the surrogate initialization. To solve this problem, known as coldstart, several solutions have been investigated.

In AutoSklearn [15], about 140 datasets are represented as vectors made of 38 meta-features, and associated to the best pipeline ever found. When a new dataset is used, AutoSklearn initializes the search process with the best configuration found for the closest dataset w.r.t. Euclidian distance in the meta-feature space. A more generic approach consists in learning the metric between datasets, using e.g. a Siamese Network [30].

At a lower level, [31] proposes to predict the impact of individual preprocessing operators rather than the whole pipeline.

For more extensive surveys on meta-learning, we refer the reader to [1, 32]

2.5. Limits of current approaches

The intrinsic difficulty of building a machine learning pipeline lies in the nature of the search space:

- the objective is non-separable i.e., the marginal performance of an operator a depends on all the operators in all the paths leading to a from the source,

Journal Pre-proof

- within the configuration space of a specific operator a, there might be some dependencies between the hyperparameters (e.g. for Neural Networks, the coefficients α, β_1 and β_2 make sense only for Adam solver [33]).

Therefore, building a machine learning pipeline is a mix between selecting a proper sequence of operations and, for each operation, selecting the proper configuration in a structured and conditional space. On the contrary, most AutoML systems handle the problem by aggregating the whole search space, losing the sequential aspect of it. A notable exception is Mosaic [21], inspired by Alpha3Dm, that explores the search space in terms of actions on operators (insertion, deletion, etc.).

A second limitation is that most AutoML frameworks propose a static search space and, even more constraining, a fixed *pipeline prototype* i.e., a high-level structure defining an ordered sequence of operator types (a precise definition is given in Section 3). For instance, AutoSklearn has a fixed pipeline made of one feature selection operator among 13 operators, and one up to three data preprocessing operator among only four. Those data preprocessing operators are of various nature: one-hot encoder, a specific imputation, balancing and rescaling method. There is no possibility to add custom operators nor to specify additional constraints in case some additional knowledge is available (e.g. no need for imputation since there is no missing values).

To the best of our knowledge, the only approach that uses a non-predetermined sequence of operators is Tpot [27], but it is not possible to add additional constraints.

3. Data Pipeline Selection and Hyperparameter Optimization

3.1. Operators and pipelines

As mentioned before, most approaches model pipelines as fixed ordered sequences of m algorithms and define for each step a specific set of algorithms that can be used. We define a more general version of pipelines, and then constrain this definition for a tradeoff between practical usage and flexibility.

We define a machine learning pipeline $\mathbf{p}$ as a Directed Acyclic Graph (DAG) with two distinguished nodes: a *source*, from which all paths start,

Journal Pre-proof

Journal Pre-proof

and a *sink*, from which all paths end[2]. The source is the node by which the data enter, and the sink produces the solution to the machine learning problem. Generally, the nodes represent operators or algorithms. Note that the sink is not necessarily a machine learning algorithm but might also be an operator that aggregates the result of several algorithms (or paths) in an ensemble fashion.

Generally, an operator parametrized by $\boldsymbol{\gamma} \in \Gamma$ is a function defined by

$$a_{\boldsymbol{\gamma}} \colon \mathcal{X}_1^{n_1} \to \mathcal{X}_2^{n_2} \times \mathcal{X}_2^{\mathcal{X}_1}$$
$$\mathcal{D}_1 \mapsto \mathcal{D}_2, T_{\mathcal{D}_1}.$$

The function operates over a dataset expressed in a certain representation space $\mathcal{X}_1$ and expresses it into another representation space $\mathcal{X}_2$. The size of the dataset may be modified (e.g. rebalancing operation), the input and output space may be the same (e.g. removing outliers) and the dimension of the input and output space may differ (e.g. Principal Component Analysis). The operator initially operates using the whole dataset (e.g. PCA or missing values imputation using a value derived from the available data such as the mean or median). It also returns a functor from $\mathcal{X}_1$ to $\mathcal{X}_2$ that is used to project a single element during the prediction phase. For instance, a PCA on the training set expresses this dataset in a k dimensional space. The associated functor is the projector from the original space to the new space. For a rebalancing operator, as it operates only on the training set, the functor is the identity. Note that the functor is implicitly parametrized by $\boldsymbol{\gamma}$ (e.g. the parameter k in a PCA is forwarded to the functor).

Two operators are **compatible** if the representation output space $\mathcal{X}_2$ of the first operator is the same as the representation input space $\mathcal{X}_1$ of the second one. A pipeline is said **compatible** if all the connected nodes are compatible.

3.2. DPSH *problem*

We propose a reformulation of CASH that better takes into account the nature of machine learning pipelines and the way practitioners work.

Given a collection of operators $\mathcal{A}$, the Data Pipeline Selection and Hy-

[2]The words *source* and *sink* come from Flow Network vocabulary.

perparameter Optimization (DPSH) problem is defined by:

$$\mathbf{p}^*_{\boldsymbol{\gamma}^*} = \underset{m\in\mathbb{N},\mathbf{p}\in\mathcal{G}_m(\mathcal{A}),\boldsymbol{\gamma}\in\boldsymbol{\Gamma}(\mathbf{p})}{\arg\min} \frac{1}{k}\sum_{i=1}^{k}\mathcal{L}(\mathbf{p}_{\boldsymbol{\gamma}}, \mathcal{D}^{(i)}_{\text{train}}, \mathcal{D}^{(i)}_{\text{test}}), \quad \text{(DPSH)}$$

where $\mathcal{G}_m(\mathcal{A})$ is the set of compatible pipelines with m vertices selected in $\mathcal{A}$ (with possible replacement), and $\boldsymbol{\Gamma}(\mathbf{p})$ the cartesian product of the configuration space of each operator in the pipeline $\mathbf{p}$. The size m of the sequence is unknown a priori.

3.3. Pipeline prototypes

Current approaches that intend to handle the pipeline construction uses a fixed sequence of few steps with a strict order, without possibility to change this topology. While this is useful for end-user with zero knowledge, it is also too restrictive for more complex workflows or end-user with slightly more advanced expertise. On the contrary, the problem depicted by DPSH is far too general to be directly handled because of the enormous number of operators one can imagine, and the number of possible compatible pipelines built over them.

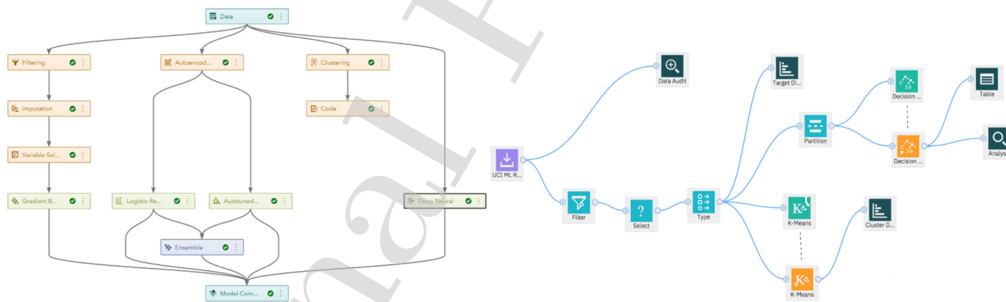


Figure 2: Example of real-life pipelines designed with SAS (left) and IBM Watson Studio (right). The flow can be complex, and the topology totally depends on the problem to solve.

Figure 2 depicts two real-life pipelines created by two different pipeline modelers. They are more complex than the usual small pipelines studied in the AutoML literature. Also, the network topology is highly different from one to another, notably because each pipeline is designed to answer a specific question on specific data. While the data scientist might not know

Journal Pre-proof

exactly what concrete operations will perform the best, she has a general idea of the topology and some orders on the operations. This knowledge can be translated into constraints on the set of compatible pipelines.

We define a **pipeline prototype** as a particular graph topology organized in **layers**. Each layer groups specific operations by their purpose, with a level of granularity to be decided by the user. For instance, a layer could be specifically dedicated to imputation if the user knows some values are missing, or generically labelled as *feature engineering* with all possible sorts of preprocessing operators. An overly specialized prototype is easy to optimize but restricts the search space and thus the possibility to find outstanding configurations. On the contrary, too broadly defined prototype make the search very difficult as a lot of combinations are either non-feasible or lead to poor results. For instance, in Figure 2, left part, a possible prototype could be made of four layers: data preprocessing (orange), model building (green), ensembling (dark blue) and model aggregation (light blue).

To control the pipeline compatibility, two combined approaches are used at the *software* level:

- for each operator, we defined the input and output space in terms of types (e.g `int`, `float`, dictionary, vector of mixed types, etc.). When a pipeline is selected, a fast preliminary check can be done by graph traversal without having to feed the pipeline,
- during the execution, non-compatible pipelines throw an exception that can be caught as early as possible.

In both cases, the pipeline is declared *incompatible* by setting the loss function to $+\infty$, with the benefit of preventing the metaoptimizer to explore close areas of the search space, most likely incompatible as well.

Working with **pipeline prototypes** allows to drastically reduce the search space of DPSH while aligning on real-life practices.

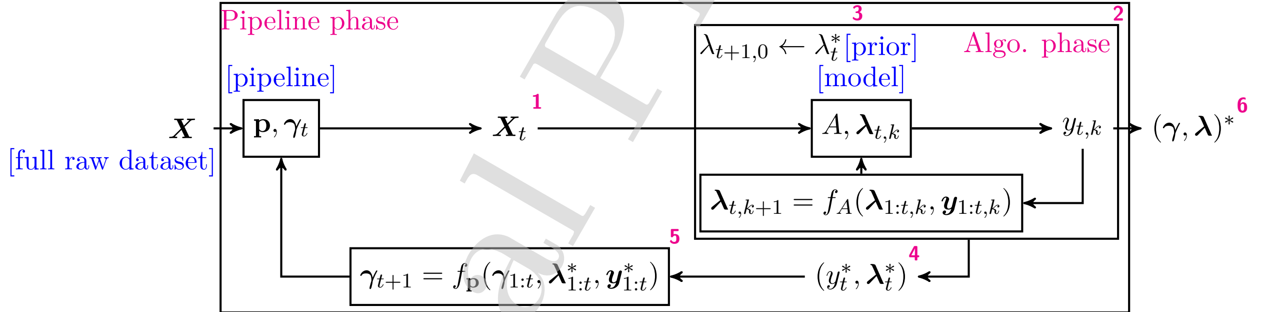
4. Two-stage optimization under time-budget

For our purpose, we consider a simplified version of the problem where the algorithm is given such that we are left with its configuration. To solve this problem, rather than considering one large search space, we break the optimization process into two smaller problems: searching for a good pipeline, and searching for a good algorithm configuration.

Journal Pre-proof

The rationale behind breaking down the AutoML optimization process lies into the distinct nature of the two steps described by Figure 1. *Feature engineering* deals for most with improving the quality of the data, like a craftsman that transforms raw material into remarkable and consumable objects. In general, it has little to do with the person that will buy the object, or to speak less metaphorically, with the algorithm that consumes the preprocessed data. Obviously, this assumption is true up to a certain point since algorithms might have different structural requirements on their input space (e.g. categorical features for tree-based methods, and the influence of encoders on their performances), and some algorithms might be more sensitive to some preprocessing steps due to their mathematical properties. For instance, for a given time budget, a dimensionality reduction grants, theoretically a least, a better advantage to an algorithm with a $O(N^2)$ time complexity rather than a $\mathcal{O}(\log N)$ algorithm where N is the initial input space dimension.

4.1. Architecture



[1] A single pipeline transforms the whole dataset at each iteration.
[2] The output $y_{t,k}$ in the inner loop is a validation measure (e.g. cross-validation).
[3] The inner loop is initialized with the previous best configuration as prior.
[4] The inner outputs the best prediction and configuration at iteration t.
[5] f_M returns the best promising configuration w.r.t. the best achievable metric.
[6] The whole process returns the best configuration to be used in production.

Figure 3: Two-stage optimization process

The proposed two-stage optimization process is illustrated by Figure 3. Due to the sequential nature of the machine learning workflow, it consists in

Journal Pre-proof

two imbricated loops. The inner loop performs a cross-validation on $\mathbf{X}_t$ for a given time budget or until a Cauchy criterion is verified (e.g. the difference in the cross-validation score between two iterations of the inner loop is lower than a threshold ε).

The rationale behind using the previous optimal algorithm configuration $\boldsymbol{\theta}_t^*$ as prior for the next inner loop is that if the "distance" between the processed data $\mathbf{X}_t$ at iteration t and $\mathbf{X}_{t+1}$ is small, we can expect the new optimal configuration for the algorithm to be rather close, shortening the inner loop computation time if a Cauchy criterion is used.

Notice that this architecture is independent of the metaoptimizer. Even more, different metaoptimizers can be used for the two stages, and different optimization criteria might be used for each step. For instance, one can imagine optimizing the data pipeline for a fairness criterion and a standard performance metric such as the (cross-validation) accuracy for the algorithm.

4.2. Policies

In practical situations, the limiting factor to construct a machine learning workflow is time. Therefore, we are interested in optimization under time constraint, and thus finding how to allocate time between both steps and through iterations. Given a time budget of T, we define different policies of time allocation.

Split policy: The budget is split between T_1 and T_2, allocated respectively for the data pipeline configuration search and the algorithm hyperparameter tuning. In the first phase, the metaoptimizer is used during T_1 to build the data pipeline. During the second phase, the metaoptimizer is used to configure the algorithm during T_2.

Considering the convex combination $T = (1 - \omega)T_1 + \omega T_2$, for $\omega = 0$, no hyperparameter tuning is done because the whole budget is spent on building the data pipeline. Conversely, for $\omega = 1.0$, the process is fully dedicated to tune hyperparameters of the algorithm.

Iterative policy: Each step alternates during a short runtime of t seconds. The best configuration found during a step is reused during the next step, iteratively until the total budget is expired. It is relatively fast for the metaoptimizer to find a better pipeline configuration than the baseline, but then, stagnates. Therefore, the time would be better allocated to the search for a better algorithm configuration. If the data pipeline was modified, the data

Journal Pre-proof

used to train the algorithm changes. Those variations might help during the hyperparameter tuning to explore a new region of the search space compared to the previous iteration.

Adaptive policy: This policy reuses the iterative policy. However, the time allocation is not fixed per iteration. If during an iteration the cross-validation score improved, the time allocated to the next iteration of the same type (data pipeline or algorithm configuration) is multiplied by two. Conversely, if after two iterations of the same type, the score is not improved, the allocated time for this type of iteration is divided by two.

Joint policy: This policy simply uses the union of both search space. In other words, it is equivalent to what is usually done in practice by current meta-optimizers, i.e., searching over the whole space of machine learning pipelines.

5. Experimental Setting

5.1. Experimental setting

Datasets and algorithms. We performed the experiments on 3 datasets: Wine[3], Iris[4] and Breast[5][6]. We used 4 classification algorithms: SVM, Random Forest, Neural Network and Decision Tree. The implementation is provided by Scikit-Learn [34].

Data pipeline search space. We created a *pipeline prototype* made of three layers: "rebalance" (handling imbalanced dataset), "normalize" (scaling features), "features" (feature selection or dimension reduction). For each step, we selected few possible concrete operators with a specific configuration space summarized in Table A.6. The topology of the pipeline is defined

[3] `https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_wine.html`

[4] `https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_iris.html`

[5] `https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_breast_cancer.html`

[6] The choice of small datasets is justified by the need to know the optimal score in the search space to effectively evaluate the metaoptimizer and policies results.

Journal Pre-proof

by Figure 4. Each node can be instantiated with an operator or left empty. When both "features selection" slots are instantiated, the final vector is obtained by stacking the output of both operators. There is a total of 4750 possible pipeline configurations. This is roughly the same as in AutoSklearn or Mosaic, and about 5 times less than AutoWeka.

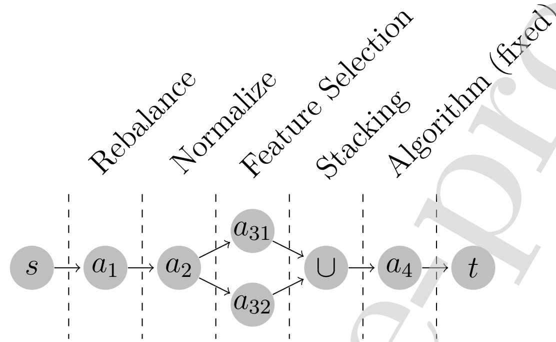


Figure 4: Pipeline topology created for the experiments.

Algorithm hyperparameter search space. For each algorithm, we defined a reasonable hyperparameter space with approximately the same size as the data pipeline space. The search space size contains 4800 elements for Random Forest and Decision Tree, 1944 for Neural Network and 768 for SVM.

Metaoptimizer. We selected Hyperopt [16] as metaoptimizer as it has been shown to perform better than alternatives for high dimensional problems [35]. A 10-fold cross-validation is used to assess the pipeline performances.

Resources. The machine used for the experiments is equipped with an Intel i7-6820HQ and 32GB RAM.

5.2. Protocol and goals

We decompose the experiments in two parts. First, given a restricted budget, we want to assess the respective influence of searching for a data pipeline and configuring the algorithm. The second part focuses on the optimization with a time budget and the evaluation of two-stage optimization process, and

Journal Pre-proof

in particular the different policies. All experiments are reproducible and the source-code is documented and available in a dedicated GitHub repository[7].

Experiment 1. We would like to quantify the impact of each phase on the final result. For this, we proceed in two steps. First, we perform an exhaustive search in the data pipeline search space, followed by a search using Hyperopt with a budget of 100 configurations to explore (about 2% of the configuration space). The algorithm uses the default configuration of its implementation. In a second step, we perform the same for hyperparameter tuning with the baseline data pipeline. Those two steps have been repeated for each algorithm and each dataset. We report the density of configurations depending on the accuracy, both for the exhaustive search and for the restricted budget.

Experiment 2. We ran the two-stage optimization process for $T = 300$ seconds, for each dataset, each method and each policy. For the **split policy**, we performed the experiment for each $\omega \in \{0, 0.1, ..., 0.9, 1\}$ to show the effect of different allocations between the two stages. For the **iterative policy**, we setup the iteration runtime to 15 seconds. Similarly, the default iteration time for the **adaptive policy** has been fixed to 15 seconds.

6. Results

6.1. Experiment 1

Figure 5 provides the result obtained with Random Forest on Breast and Wine. A summary of the results is provided by Table 1. All results are qualitatively similar. Figure 5, on Breast (top part), shows that the baseline score is 0.9384 and the best score 0.9619 i.e. an error reduction of 38% is achievable in the pipeline search space. Similarly, on Wine (bottom part), the best accuracy is 0.9906, i.e. a reduction of 25% of the error rate. Most configurations deteriorate the baseline score. However, Hyperopt is skewed towards better configuration compared to the exhaustive search. It indicates Hyperopt has a better probability to find a good configuration than random search. The right parts show that Hyperopt starts to improve the baseline score after only 4 iterations and reached its best configuration after 19 iterations on Breast (resp. 5 and 20 for Wine).

[7] `https://github.com/aquemy/DPSO_experiments`

Journal Pre-proof

Journal Pre-proof

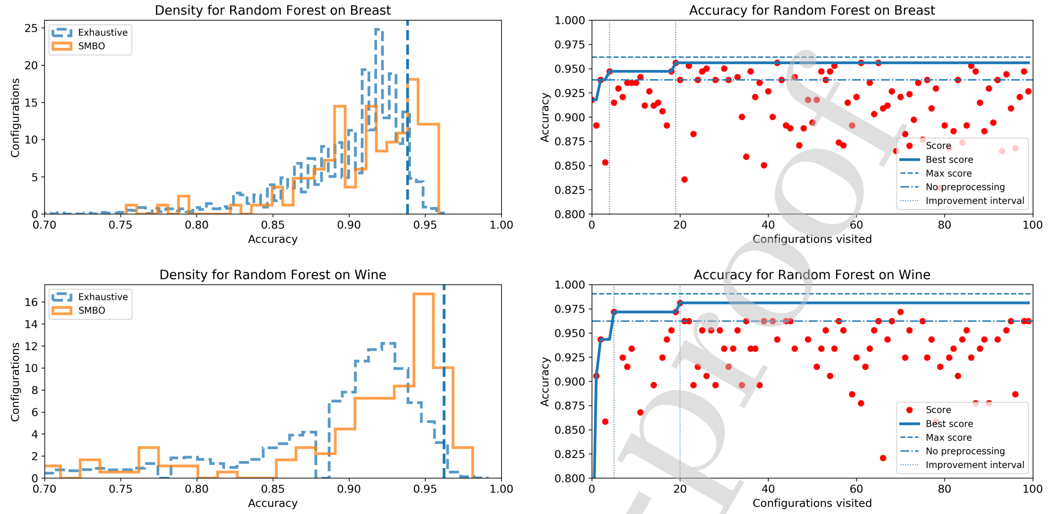


Figure 5: Density of pipeline configurations (left). The vertical line represents the baseline score. Evolution of the accuracy iteration after iteration (right).

Table 1 shows that similar results are obtained for all methods on all datasets. Hyperopt always found a better pipeline than the baseline, in at most 17 iterations. In average, the best score is achieved around 20 iterations (excluding Decision Tree on Iris and Breast). Decision Tree was able to reach the optimal configuration on Iris (resp. Wine) after 1 (resp. 5) iterations. In general, the score in the normalized score space belongs to $[0.9780, 1.000]$. To summarize, in average, with 20 iterations (0.42% of the search space) Hyperopt is able to decrease the error by 58.16% compared to the baseline score and found configurations that reach 98.92% in the normalized score space.

Exhaustive results for the algorithm step are better for Random Forest and Decision Tree on Breast only and similar to the data pipeline one only for Random Forest on Wine. In all other nine cases, the optimal score reachable in the search space is better for the data pipeline phase.

Regarding the score after 100 iterations, Hyperopt found a better accuracy for the data pipeline construction in all cases, except for Random Forest on all datasets, and SVM on Iris dataset.

Table 2 reports the best pipelines obtained when optimizing for the data pipeline only. Note that the interpretation is limited by the fact that several

Journal Pre-proof

Table 1: Optimization results.

	Baseline	Exhaustive DP	Exhaustive Algo.	Cross-validation Score DP	Cross-validation Score DP (norm.)	Cross-validation Score Algo.	Cross-validation Score Algo. (norm.)	Imp. Interval DP	Imp. Interval Algo.
Iris									
SVM	0.9667	**0.9889**	-	0.9778	0.9831	**0.9866**	-	[11, 11]	[2, 10]
Random Forest	0.9222	**0.9778**	0.9667	**0.9667**	0.9828	**0.9667**	1.0000	[8, 27]	[1, 70]
Neural Net	0.9667	**0.9889**	0.9778	**0.9778**	0.9831	0.9667	0.9820	[17, 17]	[1, 11]
Decision Tree	0.9222	**0.9889**	0.9667	**0.9889**	1.0000	0.9667	1.0000	[1, 83]	[12, 36]
Breast									
SVM	0.9501	**0.9765**	-	**0.9765**	1.0000	0.9474	-	[12, 20]	-
Random Forest	0.9384	0.9619	**0.9765**	0.9560	0.9780	**0.9685**	0.8982	[4, 19]	[4, 46]
Neural Net	0.9326	**0.9765**	0.9472	**0.9707**	0.9903	0.9175	0.9600	[1, 7]	[3, 13]
Decision Tree	0.9296	0.9619	**0.9648**	**0.9589**	0.9900	0.9527	0.9605	[0, 67]	[13, 23]
Wine									
SVM	0.9151	**1.0000**	0.9728	**0.9906**	0.9811	0.9728	1.0000	[3, 13]	[1, 3]
Random Forest	0.9623	**0.9906**	**0.9906**	0.9811	0.9818	**0.9906**	1.0000	[5, 20]	[1, 23]
Neural Net	0.9057	**0.9906**	0.9434	**0.9906**	1.0000	0.9245	0.9778	[1, 25]	[1, 13]
Decision Tree	0.9057	**0.9811**	0.9528	**0.9811**	1.0000	0.9339	0.9671	[5, 35]	[11, 75]

DP and Algo. represents respectively the Data Pipeline phase and the algorithm phase. The column norm. is the cross-validation score normalized within the search space. The last column is the interval where the left bound is the number of configurations required for HYPEROPT to improve the baseline score, and the right, the number of configurations before reaching the best score.

Journal Pre-proof

different pipelines may result in the optimal score. The case of the rebancing operator is clear: for Breast and Wine, the operators and respective configurations are the same accross the methods. For Iris, the operators seems to depend on the type of method (hyperplan-based versus tree-based methods). This observation holds also for the normalization and feature selection operator on Wine. On Breast, it seems that the optimal pipeline for Decision Tree stands out from the three other methods w.r.t. normalization and feature selection. The first three method normalize using rather close operators (a standard scalar and a robust scalar that excludes the two extreme deciles) and no feature selection while Decision Tree returned the best results with no normalization but a union of a PCA and a k-best feature selection. As mentioned above, this could come from the fact that several distinct pipelines may output the optimal result. Last, on Iris, things are slightly different: the dataset has four features s.t. PCA with k=4 or K-best with k=4 is equivalent to do nothing. Also, the dataset being easy and small, normalization is most likely not helping the method to find the optimal solution such that it is not surprising to see different operators.

In conclusion, for a similarly large and realistic search space, and a given number of configurations to explore, a good data pipeline with a default algorithm configuration provides better results than a tuned algorithm with no pre-processing operations. A notable exception is Random Forest that largely benefits from hyperparameter tuning. A study of the optimal data pipelines indicates that the best pipelines might be rather *universal* for a dataset up to a certain limit which corresponds to the specificity of the method employed. In particular, hyperplan-based methods seems to have different operator *affinity* than tree-based methods. However, this conclusion needs to be confirmed on a larger set of datasets with a more quantitative study. A step toward that direction is proposed in Section 7 with the NMAD indicator.

6.2. Experiment 2

Figure 6 shows the accuracy depending on different time allocations between each phase. By construction, the accuracy reached on the second phase can be only equal or higher to the best accuracy reached by the first phase. For the same method, depending on the dataset it might be better to allocate the whole budget on a phase or another. This is the case for Decision Tree: allocating the whole budget to the algorithm configuration on Breast returns better results than spending the whole budget on the data pipeline

Journal Pre-proof

Table 2: Best pipelines found when optimizing for the data pipeline only.

	Rebalance	Normalize	Feature Selection
Iris			
SVM	SMOTE, k=6	None	PCA, k=4
Random Forest	None	PowerTransform	PCA, k=4, KBest, k=2
Neural Net	SMOTE, k=6	RobustScaler [10, 90]	KBest, k=4
Decision Tree	None	MinMaxScaler	PCA, k=4
Breast			
SVM	SMOTE, k=7	RobustScaler [10, 90], scaling, centering	None
Random Forest	SMOTE, k=7	RobustScaler, [10, 90], scaling	None
Neural Network	SMOTE, k=7	StandardScaler, centering, scaling	None
Decision Tree	SMOTE, k=7	None	PCA, k=4, KBest, k=2
Wine			
SVM	NearMiss, k=2	PowerTransform	None
Random Forest	NearMiss, k=3	MinMaxScaler	PCA, k=2
Neural Net	NearMiss, k=3	PowerTransform	None
Decision Tree	NearMiss, k=3	MinMaxScaler	PCA, k=4

construction. The exact opposite is observed on Wine. In general, the best accuracy is obtained by a tradeoff between the two phases which seems to depend both on the algorithm and the dataset. Therefore, more advanced time allocation policies such as iterative or adaptive may help. Learning to predict optimal time allocation for a new dataset and algorithm based on previous runs is left for future work.

We reported the evolution of the best accuracy through time in Figure 7. Joint policy returns a lower or equal accuracy in all cases but Random Forest on Wine. In general, it is far slower to reach a good score compared to other policies except for Random Forest on Iris where it is the first policy to reach the plateau. It is particularly visible for Decision Tree on Iris where all policies behave the same and reach the same score. In this case, Joint policy took about 20 seconds to reach the best score against 5 seconds for the second worst and 2 seconds for the best.

The results of Split 0 policy with Neural Net are really poor for all datasets. The reason is the little number of configurations visited due to the time required to train the algorithm without feature selection. On the contrary, Split 0 performs relatively well for Random Forest on all datasets. This seems to indicate that the importance of both optimization stages are

Journal Pre-proof

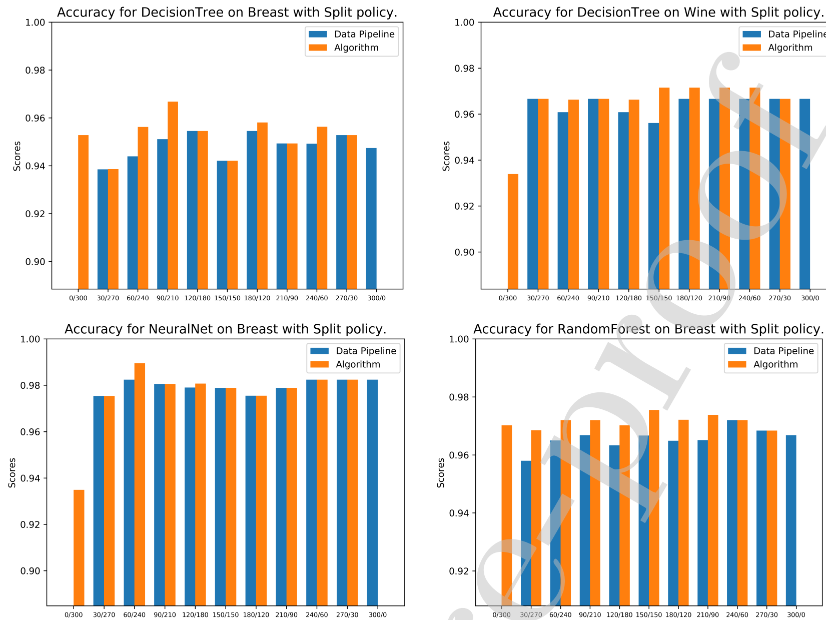


Figure 6: Accuracy depending on the time spent on each phase of the optimization process.

different depending on the algorithm.

The Adaptive policy reaches the best score in five cases, and Iterative policy in three cases. In general, at the exception of Decision Tree on Wine, the Adaptive policy manages to reach similar or better scores with less configurations sampled. The best examples are Decision Tree on Breast where the Adaptive policy reaches the best score but samples only 251 configurations against 320 and 427 for the second best policies, as well as Neural Network on Iris with 49 configurations versus 22 and 124 for a lower score.

From those results, it appears that the Iterative and Adaptive policies are the most robust ones to quickly provide good results for any method and any dataset. However, depending on the specificities of the method and the dataset, spending 100% of the time on one phase or the other might be better. In this case and in theory, the adaptive policy is equivalent to one of those Split policy after some time, i.e. the optimization process spends most of its time on one phase.

Journal Pre-proof

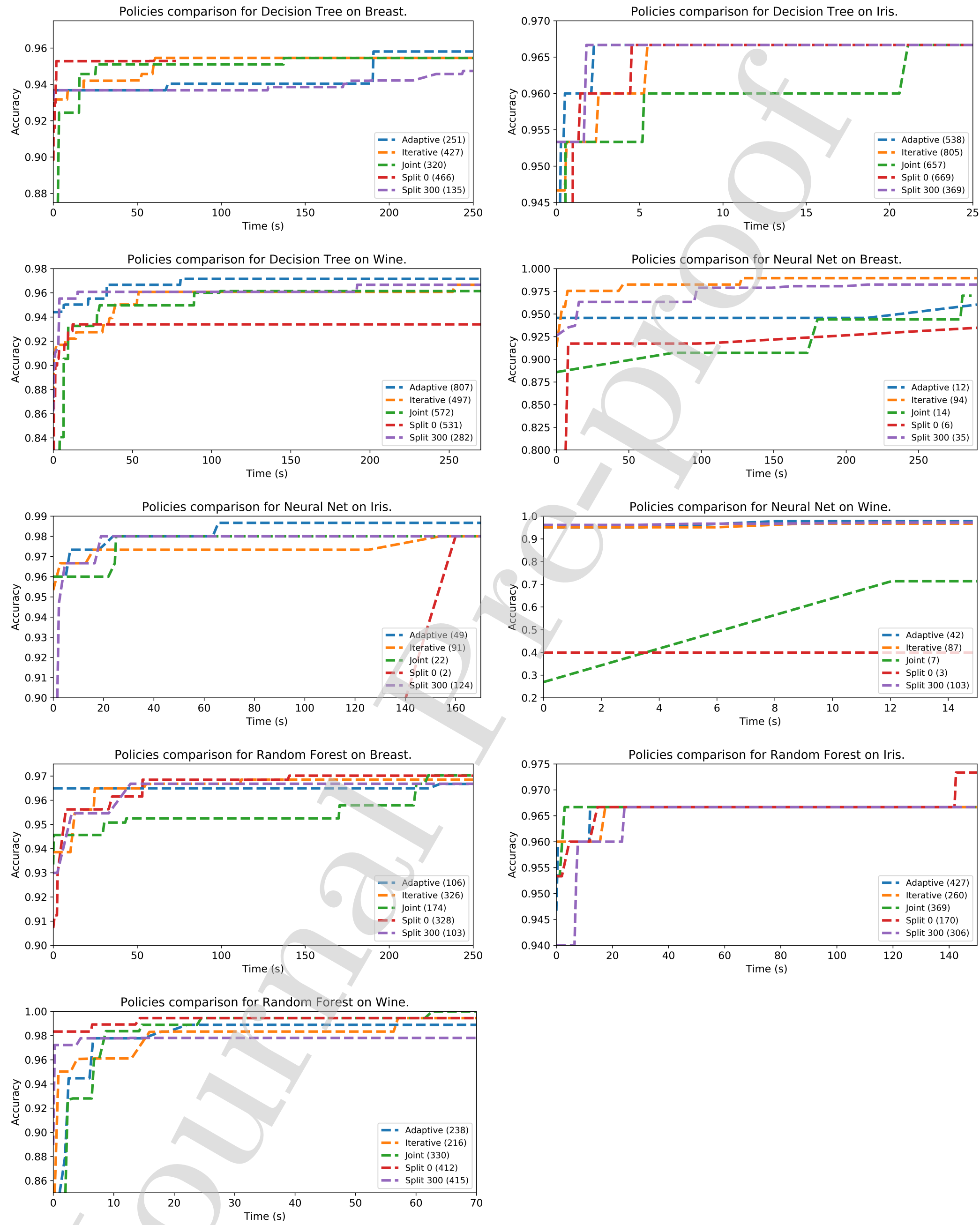


Figure 7: Evolution of the best score in time for different policies. Split 300 and Split 0 are respectively fully dedicated to the data pipeline selection or the algorithm configuration. In brackets is indicated the number of configuration visited.

Journal Pre-proof

7. Algorithm-specific configuration

A pipeline is obviously specific to the dataset or the data distribution it works on. However, our initial assumption was that the pipeline might be more or less independent from the algorithm. Therefore, we would like to quantify how much an optimal configuration is specific to an algorithm or is *universal*, i.e. works well regardless of the algorithm. For this, the optimization process might be performed on a collection of methods $\mathcal{A} = \{A_i\}_{i=1}^N$. The result is a sample of optimal configurations $\mathbf{p}^* = \{p_i^*\}_{i=1}^M$ where $M \geq N$ since an algorithm might have several distinct optimal configurations. After normalizing the configuration space to bring each axis to $[0, 1]$, the link between the processed data and the methods can be studied through a new indicator named Normalized Mean Absolute Deviation (NMAD). The idea behind this metric is to measure how much the optimal points are distant from a reference optimal point. If the optimal configuration does not depend on the algorithm, the expected distance between the optimal configurations is 0. Conversely, if a point is specific to an algorithm, the other points will be in average far from it.

Working in the normalized configuration space has two advantages. First, it forces all parameters to have the same impact. Secondly, it allows the comparison from one dataset to another since the NMAD belongs to $[0, 1]$ for any number of algorithms or dimensions of the configuration space.

The Normalized Mean Absolute Deviation is the norm 1 of the Mean Absolute Deviation[8], divided by the number of dimensions K of the configuration space.

Definition 7.1 (Normalized Mean Absolute Deviation (NMAD))**.**

$$\mathrm{NMAD}(\mathbf{p}^*, r) = \frac{1}{K}\frac{1}{N}||\big(\sum_{i=1}^{N}|p_i^* - r|\big)||_1$$

To measure how much each optimal point p_i^* is specific to an algorithm A_j, we use it as a reference point and calculate the NMAD using a sample composed of all the optimal points. However, an algorithm might have several optimal points and to be fair, we use as a representant of each algorithm, the closest point to the reference point.

[8]As we work on a discrete space, we used the norm 1, but the Euclidean norm is probably a better choice in continuous space.

Journal Pre-proof

7.1. Experimental Settings

As the configuration space described in Section 5.1 is not a metric space, we cannot directly use the NMAD. To avoid introducing bias with an *ad-hoc* distance, we perform another experiment with a configuration space that is embedded in $\mathbb{N}$.

We collected 1000 judgements documents provided by the European Court of Human Rights (ECHR) about the Article 6. The database HUDOC[9] provides the ground truth corresponding to a violation or no violation. The cases have been collected such that the dataset is balanced. The conclusion part is removed. To confirm the results, we used a second dataset composed of 855 documents from the categories atheism and religion of 20newsgroups.

Each document is preprocessed using a data pipeline consisting in tokenization, stopwords removal, followed by a n-gram generation. The processed documents are combined and the k most frequent tokens across the corpus are kept, forming the dictionary. Each case is turned into a Bag-of-Words using the dictionary.

There are two hyperparameters in the preprocessing phase: n the size of the n-grams, and k the number of tokens in the dictionary. We defined the parameter configuration domain as follows:

- $n \in \{1, 2, 3, 4, 5\}$,
- $k \in \{10, 100, 1000, 5000, 10000, 50000, 100000\}$.

We used the same four algorithms as in Section 5. As we are interested in the optimal configurations, we performed an exhaustive search.

7.2. Results

For both datasets, Figure 8 shows that the classifier returns poor results for a configuration with a dictionary of only 10 or 100 tokens. Both parameters influence the results, and too high values deteriorate the results.

Table 3 summarizes the best configurations per method. For the first dataset, there are 3 points that gives the optimal value for Random Forest and Linear SVM, however, in practice lowest parameters values are better because they imply a lower preprocessing and training time. It is interesting to notice that $(5, 50000)$ returns the best accuracy for every model, as this

[9] `https://hudoc.echr.coe.int/`

Journal Pre-proof

Journal Pre-proof

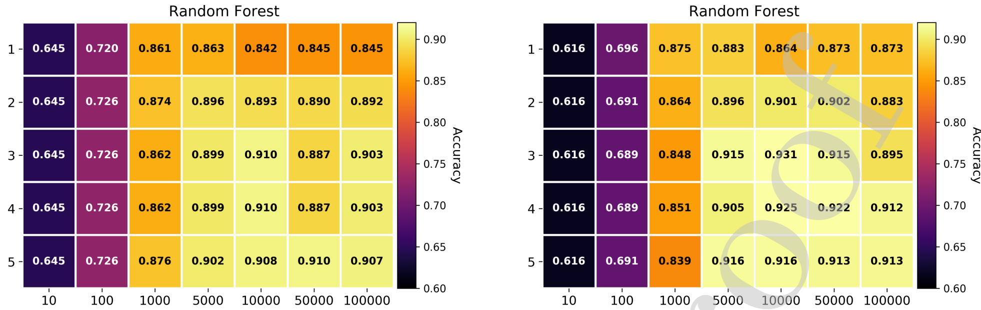


Figure 8: Heatmap depicting the accuracy depending on the pipeline parameter configuration on ECHR (left) and Newsgroup (right).

point would be a sort of *universal* configuration for the dataset, taking the best out of the data source, rather than being well suited for a specific algorithm. On the contrary, on Newsgroup, all optimal points are different. Our hypothesis is that the more structured a corpus is, the less algorithm-specific are the optimal configurations, because the preprocessing steps become more important to extract markers used by the algorithms to reach good performances. As ECHR dataset describes standardized justice documents, it is far more structured than Newsgroup. This would also explain why generating n-grams for $n = 5$ still improves the results on ECHR while degrading them on Newsgroup.

This hypothesis is partially confirmed by Table 4, where it is clear that the n-gram operator has a strong impact on the accuracy variation on ECHR dataset (up to 9.8% accuracy improvement) while almost none on Newsgroup dataset (at the exception of Random Forest).

Table 5 contains the NMAD value for each distinct optimal configuration reported in Table 3. As expected, the point $(5, 50000)$ has a NMAD of 0 since the point is present for every algorithm: $(5, 50000)$ is a *universal* pipeline configuration for this data pipeline and dataset. The point $(4, 50000)$ appears only once but it is really close to $(5, 50000)$ (itself in the 3 other algorithms results) s.t. its NMAD is low. It can be interpreted as belonging to the same area of optimal values. On the opposite, $(3, 10000)$ and $(4, 10000)$ have high NMAD w.r.t. the other points, indicating they are isolated points and may be algorithm specific. Their NMAD values are rather low because despite the points are isolated, they differ significantly from the others points only on the

Journal Pre-proof

Table 3: Best configurations depending on the method

Method	(n, k)	accuracy
ECHR		
Decision Tree	(5, 50000)	0.900
Neural Network	(5, 50000)	0.960
Random Forest	(3, 10000), (4, 10000), (5, 50000)	0.910
Linear SVM	(3, 50000), (4, 50000), (5, 50000)	0.921
Newsgroup		
Decision Tree	(4, 5000), (4, 100000)	0.889
Neural Network	(5, 50000)	0.953
Random Forest	(3, 10000)	0.931
Linear SVM	(2, 100000)	0.946

Table 4: Impact of parameter n on the accuracy, measured as the relative difference between the best results obtained only using $(1, k)$ and the best results obtained for any configuration (n, k).

Method	$p = (1, k)$	$p = (n, k)$	Δ acc
ECHR			
Decision Tree	0.850	0.900	5.9%
Neural Network	0.874	0.960	9.8%
Random Forest	0.863	0.910	5.4%
Linear SVM	0.892	0.921	6.6%
Newsgroup			
Decision Tree	0.885	0.889	0.5%
Neural Network	0.949	0.953	0.4%
Random Forest	0.883	0.931	5.4%
Linear SVM	0.945	0.946	0.1%

second component. In comparison, if $(1, 10)$ would be an optimal point for Random Forest, its NMAD would be 0.5. On the contrary, for Newsgroup, the NMAD value is rather high and similar for all points, indicating that they are at a similar distance from each other and really algorithm specific.

Journal Pre-proof

Table 5: Normalized Mean Average Deviation for each optimal configuration found.

ECHR		Newsgroup	
Point	NMAD	Point	NMAD
(5, 50000)	0	(4, 5000)	0.306
(3, 10000)	0.275	(4, 100000)	0.300
(4, 10000)	0.213	(5, 50000)	0.356
(3, 50000)	0.175	(3, 10000)	0.294
(4, 50000)	0.094	(2, 100000)	0.362

To summarize, the NMAD metric is coherent with the conclusion drawn from the heatmaps and Table 3, and suggests that there exist two types of optimal configurations: *universal* pipeline configurations that work well on a large range of algorithms for a given dataset, and algorithm-specific configurations. Thus, we are confident the NMAD can be used in larger configuration spaces where heatmaps and exhaustive results are not available for graphical interpretation, and help to reuse configurations and initialize surrogate models.

8. Conclusion

In this paper, we proposed a reformulation of the main AutoML problem to fit better the way data scientists work in practice. We studied the importance of the data pipeline and algorithm hyperparameter tuning phase on a reasonably realistic search space. We found that spending more time on setting a proper data pipeline usually leads to better results. We proposed a two-stage process to optimize a general machine learning workflow, articulated around the data pipeline construction and the hyperparameter tuning. Under time constraint, we studied four different policies to allocate time between these two phases. We found that the best results are obtained with a tradeoff that seems to depend on both the algorithm and the dataset. Last, we found that the most robust policies, in terms of reaching a good score within little time, are the most advanced policies: iterative and adaptive.

In addition, we provided a metric to study if a data pipeline is more or less independent from the algorithm. This metric can be use to suggest good data pipelines depending on meta-features as described in [31, 36]]. Future work

Journal Pre-proof

Journal Pre-proof

should focus on an online version s.t. the pipeline is tuned in a streaming way. Also, the NMAD indicator works only in euclidian spaces which is not the case for the first experiment. Therefore, further work should focus on extending the NMAD to non-vector space.

More generally, we plan a larger experimental campaign using OpenML-CC18 benchmark suit [37] with different pipeline prototypes.

A generalization to multi-stage optimization is also considered. Each layer of the pipeline could benefit from a specific time allocation depending on its marginal contribution. Some preliminary work has been done toward this direction in [38].

Last but not least, we are currently working on a library based on the code of those experiments, dedicated to the creation of flexible pipelines and budget allocation strategies. By using the operator and pipeline prototype definitions provided in this paper, we will provide an easy yet generic way to define and extend pipeline search spaces, which is currently not feasible by any AutoML systems.

References

[1] R. Elshawi, M. Maher, S. Sakr, Automated machine learning: State-of-the-art and open challenges (2019). arXiv:1906.02287.

[2] F. Hutter, L. Kotthoff, J. Vanschoren, Automatic machine learning: methods, systems, challenges, Challenges in Mach. Learn.

[3] S. F. Crone, S. Lessmann, R. Stahlbock, The impact of preprocessing on data mining: An evaluation of classifier sensitivity in direct marketing, Eur. J. Oper. Res. 173 (3) (2006) 781 – 800.

[4] T. Dasu, T. Johnson, Exploratory data mining and data cleaning, Vol. 479, John Wiley & Sons, 2003.

[5] N. M. Nawi, W. H. Atomi, M. Z. Rehman, The effect of data pre-processing on optimized training of artificial neural networks, Procedia Technology 11 (2013) 32 – 39, int. Conf. Elect. Eng. Info.

[6] D. H. Wolpert, The lack of a priori distinctions between learning algorithms, Neural Comput. 8 (7) (1996) 1341–1390.

Journal Pre-proof

Journal Pre-proof

[7] M. Chessell, F. Scheepers, N. Nguyen, R. van Kessel, R. van der Starre, Governing and managing big data for analytics and decision makers, IBM Redguides for Business Leaders.

[8] A. Quemy, Data pipeline selection and optimization, in: Pro. Int. Workshop on Design, Optim., Languages and Anal. Processing of Big Data, 2019.

[9] D. C. Montgomery, Design and analysis of experiments, John wiley & sons, 2017.

[10] J. Bergstra, Y. Bengio, Random search for hyper-parameter optimization, J. Mach. Learn. Res. 13 (Feb) (2012) 281–305.

[11] F. Hutter, H. H. Hoos, K. Leyton-Brown, Sequential model-based optimization for general algorithm configuration, in: Proc. Int. Conf. Learn. Intel. Optim., Springer-Verlag, Berlin, Heidelberg, 2011, pp. 507–523.

[12] J. Bergstra, R. Bardenet, Y. Bengio, B. Kégl, Algorithms for hyper-parameter optimization, in: Proc. Int. Conf. Neural Inf. Process. Syst., 2011, pp. 2546–2554.

[13] C. Thornton, F. Hutter, H. H. Hoos, K. Leyton-Brown, Auto-weka: Combined selection and hyperparameter optimization of classification algorithms, in: Int. Conf. Knowl. Disc. Data Min., ACM, 2013, pp. 847–855.

[14] L. Kotthoff, C. Thornton, H. H. Hoos, F. Hutter, K. Leyton-Brown, Auto-weka 2.0: Automatic model selection and hyperparameter optimization in weka, J. Mach. Learn. Res. 18 (1) (2017) 826–830.

[15] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, F. Hutter, Efficient and robust automated machine learning, in: C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, R. Garnett (Eds.), Proc. Int. Conf. Neural Inf. Process. Syst., 2015, pp. 2962–2970.

[16] J. Bergstra, B. Komer, C. Eliasmith, D. Yamins, D. D. Cox, Hyperopt: a python library for model selection and hyperparameter optimization, Comput. Sci. & Discovery 8 (1) (2015) 014008.

Journal Pre-proof

Journal Pre-proof

[17] J. Snoek, H. Larochelle, R. P. Adams, Practical bayesian optimization of machine learning algorithms, in: Proc. Int. Conf. Neural Inf. Process. Syst., 2012, pp. 2951–2959.

[18] J. Wilson, F. Hutter, M. Deisenroth, Maximizing acquisition functions for bayesian optimization, in: Proc. Int. Conf. Neural Inf. Process. Syst., 2018, pp. 9884–9895.

[19] P. I. Frazier, A tutorial on bayesian optimization, arXiv preprint arXiv:1807.02811.

[20] J. Močkus, On bayesian methods for seeking the extremum, in: Optimization Techniques IFIP Technical Conference, Springer, 1975, pp. 400–404.

[21] H. Rakotoarison, M. Sebag, AutoML with Monte Carlo Tree Search, in: Workshop AutoML 2018 @ ICML/IJCAI-ECAI, Pavel Brazdil, Christophe Giraud-Carrier, and Isabelle Guyon, Stockholm, Sweden, 2018.

[22] T. Domhan, J. T. Springenberg, F. Hutter, Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves, in: Int. Conf. Artif. Intel., 2015.

[23] K. Swersky, J. Snoek, R. P. Adams, Freeze-thaw bayesian optimization, arXiv preprint arXiv:1406.3896.

[24] K. Jamieson, A. Talwalkar, Non-stochastic best arm identification and hyperparameter optimization, in: Artificial Intelligence and Statistics, 2016, pp. 240–248.

[25] L. Li, K. Jamieson, Hyperband: A novel bandit-based approach to hyperparameter optimization, J. Mach. Learn. Res. 18 (2018) 1–52.

[26] J. Nalepa, M. Myller, S. Piechaczek, K. Hrynczenko, M. Kawulok, Genetic selection of training sets for (not only) artificial neural networks, in: Proc. Int. Conf. Beyond Databases, Architectures Struct., 2018, pp. 194–206.

[27] R. S. Olson, N. Bartley, R. J. Urbanowicz, J. H. Moore, Evaluation of a tree-based pipeline optimization tool for automating data science, in: Proc. Gen. and Evol. Comput. Conf., ACM, 2016, pp. 485–492.

Journal Pre-proof

Journal Pre-proof

[28] B. Chen, H. Wu, W. Mo, I. Chattopadhyay, H. Lipson, Autostacker: A compositional evolutionary learning system, in: Proc. Gen. and Evol. Comput. Conf., 2018, pp. 402–409.

[29] X. Sun, J. Lin, B. Bischl, Reinbo: Machine learning pipeline search and configuration with bayesian optimization embedded reinforcement learning, CoRR abs/1904.05381.

[30] J. Kim, S. Kim, S. Choi, Learning to warm-start bayesian hyperparameter optimization, arXiv preprint arXiv:1710.06219.

[31] B. Bilalli, A. Abelló, T. Aluja-Banet, On the predictive power of meta-features in openml, Int. J. Appl. Math. Comput. Sci. 27 (4) (2017) 697–712.

[32] J. Vanschoren, Meta-learning: A survey, arXiv preprint arXiv:1810.03548.

[33] D. P. Kingma, J. Ba, Adam: A method for stochastic optimization, arXiv preprint arXiv:1412.6980.

[34] 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, E. Duchesnay, Scikit-learn: Machine learning in Python, J. Mach. Learn. Res. 12 (2011) 2825–2830.

[35] K. Eggensperger, M. Feurer, F. Hutter, J. Bergstra, J. Snoek, H. Hoos, K. Leyton-Brown, Towards an empirical foundation for assessing bayesian optimization of hyperparameters, in: NIPS workshop on Bayesian Optimization in Theory and Practice, 2013.

[36] B. Bilalli, A. Abelló, T. Aluja-Banet, R. Wrembel, Intelligent assistance for data pre-processing, Computer Standards & Interfaces (2018) 101 – 109.

[37] B. Bischl, G. Casalicchio, M. Feurer, F. Hutter, M. Lang, R. G. Mantovani, J. N. van Rijn, J. Vanschoren, Openml benchmarking suites and the openml100, arXiv preprint arXiv:1708.03731.

Journal Pre-proof

Journal Pre-proof

[38] A. Chowdhury, M. Magdon-Ismail, B. Yener, Quantifying error contributions of computational steps, algorithms and hyperparameter choices in image classification pipelines, CoRR abs/1903.02521.

Journal Pre-proof

Journal Pre-proof

AppendixA. Pipeline configuration space

Table A.6: Pipeline search space.

	$\#\lambda$	$\|\Lambda\|$	impl.
Rebalance			
No operator	0	0	-
Near Miss	1	3	imblearn
Condensed Nearest Neighbour	1	3	imblearn
SMOTE	1	3	imblearn
Normalize			
No operator	0	0	-
Standard Scaler	2	4	sklearn
Power Transform	0	0	sklearn
MinMax Scaler	0	0	sklearn
Robust Scaler	3	12	sklearn
Features			
No operator	0	0	-
PCA	1	4	sklearn
SelectKBest (F-score)	1	4	sklearn
PCA ∪ SelectKBest	2	16	sklearn

The column $\#\lambda$ is the number of parameters while $|\Lambda|$ is the total number of values in the operator configuration space. The column *impl.* indicates the implementation of the operator (scikit-learn or imbalanced-learn).

Journal Pre-proof