Effective data pre-processing for AutoML

Joseph Giovanelli
University of Bologna
j.giovanelli@unibo.it

Besim Bilalli
Universitat Politècnica de Catalunya
bbilalli@essi.upc.edu

Alberto Abelló
Universitat Politècnica de Catalunya
aabello@essi.upc.edu

ABSTRACT

Data pre-processing plays a key role in a data analytics process (e.g., supervised learning). It encompasses a broad range of activities that span from correcting errors to selecting the most relevant features for the analysis phase. There is no clear evidence, or rules defined, on how pre-processing transformations (e,g., normalization, discretization, etc.) impact the final results of the analysis. The problem is exacerbated when transformations are combined into pre-processing pipeline prototypes. Data scientists cannot easily foresee the impact of pipeline prototypes and hence require a method to discriminate between them and find the most relevant ones (e.g., with highest positive impact) for their study at hand. Once found, these pipelines can be optimized using AutoML in order to generate executable pipelines (i.e., with parametrized operators for each transformation). In this work, we study the impact of transformations in general, and the impact of transformations when combined together into pipelines. We develop a generic method that allows to find effective pipeline prototypes. Evaluated using Scikit-learn, our effective pipeline prototypes, when optimized, provide results that get 90% of the optimal predictive accuracy in the median, but with a cost that is 24 times smaller.

1 INTRODUCTION

The decision making process has historically been key for the success of any organization or business activity. Lately, with the abundant presence of data, this process has become data-driven, where data are continuously analyzed to be transformed into knowledge. Along the way however, data undergo several (sometimes necessary) processing steps, shown in Figure 1. Firstly, data arriving in a raw format from different sources are sifted out such that only a relevant subset is selected. Next, this subset is pre-processed and is fed to a machine learning (ML) algorithm for it to be analyzed. The output of the analysis is then interpreted and the whole process iterates until the results obtained are satisfactory and significant for the decisions to be made.

Unfortunately, this well known process does not have universal well-defined practices for the different steps, which translates to the data scientist manually configuring and parameterising the operators for each step until an optimal solution is found — an optimal *data analytics pipeline*. To this end, most of the time is spent on the heavily laborious work of pre-processing (i.e., 50-80% of the time [17]), where the generated output is a *pre-processing pipeline*. Next, once the data is transformed into the proper form, different ML algorithms with different hyperparameters are evaluated over the dataset until an acceptable result is obtained — *ML pipeline*. This whole process requires expertise and is particularly challenging for novice, inexperienced data scientists. To help with the challenge of finding optimal analytics pipelines several techniques have been proposed, among them AutoML being the most prominent.

AutoML is a technique originally proposed for automatically selecting and parameterising the learning algorithm, which has been known as the Combined Algorithm Selection and Hyper-parameter Optimization (CASH) problem [15]. Given a budget in terms of time or number of iterations, an optimization algorithm iterates by visiting a potentially better configuration each time, until a near optimal solution is obtained. AutoML has become the de-facto standard procedure for CASH, demonstrating very good results [8] even in Kaggle competitions. However, traditionally its drawback has been that it focuses mainly on optimizing the hyperparameters of the learning algorithms, often overlooking the impact of data pre-processing. The process gets stuck on a near optimal hyper-parameter configuration, and no matter the number of iterations, the results do not improve. This hinders the real power of AutoML because pre-processing operators do not get thoroughly considered. Since CASH is one aspect of automatically finding the best data analytics pipeline, AutoML has been extended to also specifically cover the pre-processing part. The latter has been coined as the Data Pipeline Selection and Optimization (DPSO) problem [19], where a *pipeline prototype* (sequence of transformations, e.g., missing value imputation followed by normalization) is fed to AutoML and an optimal instance of the prototype, in the form of a pipeline (sequence of operators, e.g., imputation by mean followed by min-max normalization) is found. By considering pre-processing as an integral component of data analytics, and carefully configuring the pre-processing pipelines, it is easy to obtain results that go beyond the ones obtained by only optimizing the learning algorithm.

To briefly illustrate this, we perform an experiment on the well known `bank-marketing`[1] dataset, using HyperOpt [1] as an AutoML approach to optimize the parameters of three different ML algorithms, namely Naive Bayes (NB), K-Nearest Neighbor (KNN), and Random Forest (RF). We provide an initial budget of 50 iterations for optimizing the hyper-parameters of the algorithms, and after the 50th iteration, we fix the algorithm configuration to the best one achieved so far and start optimizing the pre-processing pipeline[2]. In Figure 2, the improvement ratio of predictive accuracy (i.e., ratio of the accuracy obtained after the i-th iteration to the baseline/default accuracy) is plotted against the number of different configurations visited by HyperOpt (i.e., iterations). Observe that after the 11th iteration for NB and KNN, and after the 26th iteration for RF, the lines remain flat. That is, from there on, no improvement is achieved by optimizing the hyper-parameters of the algorithms until the 50th iteration is reached. At this point, a sudden jump is observed and the results start to improve again, going clearly

Copyright © 2021 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

[1] https://archive.ics.uci.edu/ml/datasets/Bank+Marketing
[2] This order is used only for the sake of illustration.

CEUR-WS.org/Vol-2840/paper1.pdf

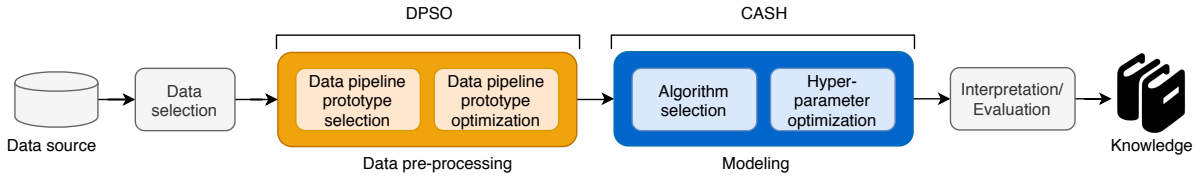

Figure 1: Data analytics pipeline generation in a knowledge discovery process.

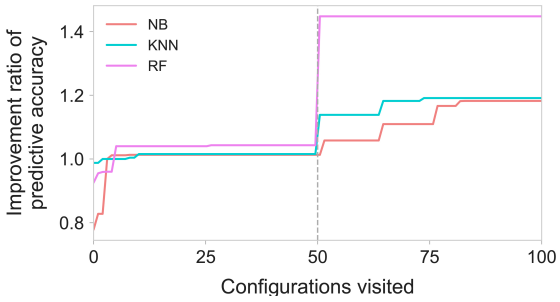

Figure 2: Evolution of predictive accuracy during the optimization process. The first 50 configurations optimize only the hyper-parameters and after the 50th configuration, the pre-processing pipeline is optimized instead.

beyond the ones obtained before, thanks to the optimizations performed now over the pre-processing pipeline. Yet, including the pre-processing in a free form in the optimization, heavily increases the search space, making the problem much harder. This is mitigated by creating a pre-processing pipeline prototype that fixes the order of transformations, leaving the freedom to only instantiate and parametrise them. Therefore, the challenge for data scientists is to find the right pre-processing pipeline prototype to optimize, that is, (i) which pre-processing transformations to consider in the prototype, and (ii) how to order them such that when optimizing the parameters of their operators, better results are obtained. The aim of this work is thus to study these two questions in order to propose a method for generating effective pre-processing pipeline prototypes, that once instantiated through AutoML improve the final result of the analysis. To keep discussions and experiments simpler, we stick to supervised learning tasks, which encompass algorithms generating a map function based on pairs of input-output exemplars. In particular, this work focuses on classification problems, where the output to be predicted is of categorical type.

Contributions. The main contributions of this paper can be summarized as follows:

- We empirically evaluate the impact of optimizing the exhaustive set of potential pipeline prototypes and find out that there is no single universal pipeline that works best for every dataset and algorithm considered.
- We define a method that given a learning algorithm and a set of pre-processing operators, is capable of generating the right order between operators, obtaining the most effective pre-processing pipelines.
- We perform a comprehensive evaluation by comparing the performance of optimizing the pipelines generated following our method, and find out that:
  - with 24 times less time budget, our proposed pipelines obtain results whose median is above 90% of the optimal ones.
  - on average, in 73% of the cases, splitting evenly the time budget between pre-processing and hyper-parametrisation outperforms the results of only optimizing the hyper-parameters of the ML algorithm.

The remainder of this paper is organized as follows. In Section 2, we discuss the related work. In Section 3, we present our method for constructing effective pipeline prototypes. In Section 4, we provide an extensive evaluation of the pipelines created using that method. Finally, in Section 5, we provide the conclusions and future work.

2 RELATED WORK

A lot of ongoing research aims at addressing the problem of providing user assistance for the different steps of the data analytics process. In general, there is a trend to develop (semi) automatic systems that assist the user in one or many steps altogether. At the beginning, the focus was to provide support exclusively for the learning step (i.e., the CASH problem). Recently however, the direction has shifted towards designing systems that additionally or specifically provide user assistance in the data pre-processing step (i.e., the DPSO problem).

When it comes to data pre-processing, different works have tackled this problem from different perspectives. For instance, there are works that aim to apply pre-processing for the sake of guaranteeing data quality, or enabling data exchange, or even data integration. That is, they consider data pre-processing in isolation or apart from data analysis [7, 11, 13, 14]. In this and our related work however, we consider only the works that see pre-processing as an integral part of data analytics and hence apply it for the sake of improving the final result of the analysis.

Finally, there are works that aim at fully automating the data analytics process (i.e., automatically generate data analytics flows), which roughly translates to combining DPSO with CASH, where the border line between the latter two becomes blurry. Nevertheless, we tentatively group the works based on the type of the problem they aim to solve.

2.1 DPSO

In DPD [19], the DPSO problem, as we use it in this work, is formally defined. Authors demonstrate the impact of optimizing the pre-processing pipeline, but considering only a single fixed pipeline prototype. However, as we will see later (Section 4.1), a single fixed prototype cannot perform best for every dataset. Therefore, we build on top of [19], yet instead of relying on a fixed prototype, we define a method to generate the right pipeline prototypes to be optimized.

In PRESISTANT [4–6], we tackled the problem of recommending pre-processing operators to the non-expert data analyst. The goal, and at the same time the challenge was to identify the pre-processing operators, and rank them in advance, based on their potential impact to the final analysis. However, we did not consider pre-processing pipelines, but only single transformations, expecting that the analyst applies the process iteratively. In this work, we consider sets of transformations and thus study the impact of combining transformations into a pipeline.

In ActiveClean [16], authors define an algorithm that aims at prioritizing the cleaning of records that are more likely to affect the results of the statistical modeling problems, assuming that the latter belong to the class of convex loss models (i.e., linear regression and SVMs). Hence, instead of recommending the transformations to be applied, the system recommends the subset of data which needs to be cleaned at a given point. The type of pre-processing to be applied is left to the user, assuming that the user is an expert.

In Learn2Clean [2], based on a reinforcement learning technique, for a given dataset, and an ML model, an optimal sequence of operators for pre-processing the data is generated, such that the quality of the ML model is maximized. Here, similarly to [19], the pipeline prototype is fixed in advance. Our work is a step further in that we help to choose the right pipeline prototype, instead of fixing it in advance.

In Alpine Meadow [20], authors follow a similar approach to ours in that they define two steps for the pre-processing phase. One, the so called *logical pipeline plan*, which is roughly equivalent to the *pipeline prototypes* defined in this work, and the second the *physical pipeline plan* which translates to *pipelines* used in this work. The physical plan is generated through a combination of Bayesian optimization, meta-learning, and multi-armed bandits. For the logical plans, they rely on rules but without clear evidence on how they are generated. Moreover, it is not clear whether the logical plan is fixed as in [19] and if some further adjustment from the user is required.

2.2 CASH

The task in solving the CASH problem is to automatically find an optimized instantiation for the hyper-parameters of the ML algorithm. Most of the works use Bayesian optimization methods to tune and optimize them [8, 18, 21]. Since Bayesian optimization is randomized, meta-learning has been used to find a good seed for the search [9]. Most of these works however, only minimally consider the data pre-processing step.

Auto-WEKA [21] and its counterpart package for Python, Auto-sklearn [8], solve the problem of learning algorithm selection and their associated hyper-parameter optimization in a combined search space. They use Sequential Model-based Algorithm Configuration (SMAC) to explore the large search space. These systems also consider pre-processing transformations to generate end-to-end analytic pipelines. Yet, they consider a small set of transformations and also consider a single fixed pipeline prototype. Our work is complementary to Auto-WEKA and Auto-sklearn.

TPOT [18] is a tree-based pipeline optimization tool using genetic programming while requiring little to no expertise from the user. In TPOT however, they only consider one transformation inside the optimization process (i.e., Feature Engineering).

Based on the work of [10], Microsoft developed an AutoML tool via Azure. They build predictive ML pipelines combining collaborative filtering and Bayesian Optimization. In particular they model the search space as probability distribution defined by a Probabilistic Matrix Factorization and then use expected improvement as acquisition function to choose the most promising pipeline.

To summarize, full automation of data analytics has been the ultimate goal of many research works. Yet, such an automation has shown to be computationally expensive, mainly due to the search space involved (i.e., pre-processing and mining operators). Therefore, the usability of these approaches in realistic scenarios is sometimes limited. Regardless of the latter, our approach of finding a set of effective pipeline prototypes can be seen as complementary to these solutions, since it helps in pruning the large search space.

3 DATA PRE-PROCESSING PIPELINE CONSTRUCTION

Following the notation from [19], we also distinguish between a *pipeline prototype* and a *pipeline*. The former is defined as a fixed, ordered sequence of kinds of pre-processing transformations, where each kind of transformation can be instantiated by a specific set of operators, into an actual executable *pipeline*. Typically, pipeline prototype construction is a manual and tedious task, where a data scientist exhaustively iterates over a staggeringly large number of possible pipeline orderings, until he/she finds one that works best for the problem at hand. This is a challenging task due to the fact that there are no clear rules and guidelines in terms of which permutation of kinds of transformations would work best (i.e., the final impact of a pipeline is difficult to foresee). To facilitate it, we propose a method, sketched in Figure 3, that in short breaks the combinatorial problem of finding the best pipeline into studying kinds of transformations in pairs, ultimately, generating effective pipeline prototypes. Some of the steps of the method are generic and thus can be applied regardless of the context, and yet others are specific, and depend on the context (i.e., AutoML framework used or dataset characteristics).

The process starts by selecting the ML library and AutoML framework to be used (e.g., Scikit-learn, AutoWeka), which on the one hand determine the potential kinds of transformations and their available instantiations, and on the other hand allow to generate *framework-related rules* reflecting the limitations in the concrete implementation of operators. These rules enable the generation of precedence relationships between the kinds of transformations for which they apply. Next, the process follows with a study over all the possible pairs of kinds of transformations, aiming to find the correct/meaningful order between them using *generic knowledge* about their behaviour. As a result, a set of *heuristic rules* that determines precedences between transformations is generated. Furthermore, for the pairs for which an order cannot be clearly devised, an additional empirical study is performed. This study relies on a testbed of dataset representatives, and thus it implicitly corresponds to *domain knowledge*. The output of this step is a set of *learnt rules* that determines promising precedences of transformations (i.e. an order that would potentially positively impact the final result of the

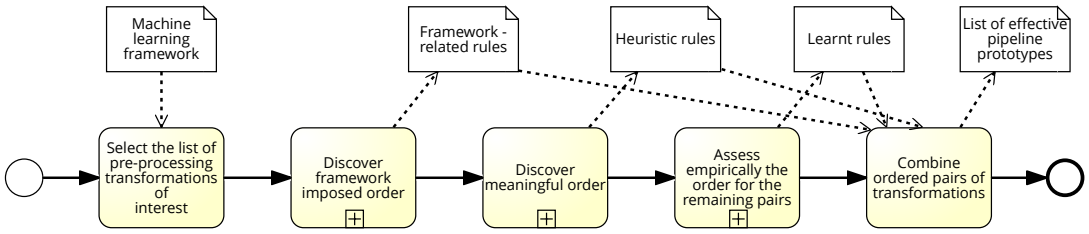

Figure 3: A method for pre-processing pipeline construction.

analysis). However, even after this phase, for some pairs of transformations an order may not be determined. These are pairs where the order is relevant but cannot be decided in advance, thus all their permutations need to be present. Finally, a step of composition follows, where given the overall set of devised rules (i.e., *framework-related*, *heuristic* and *learnt*), transformations are composed resulting in a set of valid and potentially effective pipeline prototypes.

3.1 Transformations and Operators

The first task in the process is deciding the kinds of transformations to be considered. In this work, they are the following:

- *Encoding (E).* The process of transforming categorical attributes into continuous ones.
- *Normalization (N).* The process of normalizing continuous attributes such that their values fall in the same range.
- *Discretization (D).* The process of transforming continuous attributes into categorical ones.
- *Imputation (I).* The process of imputing missing values.
- *Rebalancing (R).* The process of adjusting the class distribution of a dataset (i.e. the ratio between the different classes/categories represented).
- *Feature Engineering (F).* The process of defining the set of relevant attributes (variables, predictors) to be used in model construction.

An operator is an actual instantiation/implementation of a kind of transformation. Thus, several operators may implement the same kind of transformation, each having its own set of parameters. For our experiments, we selected the operators and parameters[3] from those available in the Scikit-learn[4] framework.

When combining two different kinds of transformations, it is important to check, (i) if the input and output types of transformations are compatible, (ii) if the combination makes sense, and (iii) if the combination provides good results for the analysis. As a result, when chaining a pair of transformations, the following precedence relationships arise:

(1) Compatible/Incompatible pairs. Depending on whether the representation output of the first transformation is accepted as the representation input of the second one (compatible), or not (incompatible) (see Section 3.2).
(2) Meaningful/Meaningless pairs. Depending on whether the precedence between them makes sense based on generic knowledge (i.e., based on the literature) over the behaviour of transformations (meaningful), or not (meaningless) (see Section 3.3).
(3) Promising/Unpromising pairs. Depending on whether the precedence between them is expected to provide positive impact on the final result of the analysis (promising), or not (unpromising) (see Section 3.4).

Thus, attending to the relationships between its transformations, a prototype can be described as either *compatible*, *well-formed*, or *effective*. A prototype is defined to be *compatible*, if all its precedence relationships are compatible. It is defined as *well-formed*, if all its precedence relationships are both compatible and meaningful. Finally, it is defined as *effective*, if all its precedence relationships are compatible, meaningful, and promising at the same time. In fact, the ultimate goal of our method is to find *effective pipelines*.

3.2 Framework-related rules

The compatibility of transformations is dependent of the selected ML framework. We studied the transformations implemented in Scikit-learn and detected a set of implicit rules that are shown through an adjacency matrix, corresponding to a precedence graph, in Table 1a. Each cell a_{ij} denotes a precedence relationship between the row i and column j. Hence, `1` means that an edge exists between the transformation in the row and the transformation in the column, while `0` means that such an edge does not exist, hence a precedence order is not established for that pair.

For example, most Scikit-learn transformations cannot be applied in the presence of missing values. This is why in every pair of transformations where Imputation is involved, except the one with Normalization[5], Imputation goes first. Furthermore, Scikit-learn transformations are applied only to all compatible attributes of a given dataset. Generally, categorical attributes are physically represented as strings and continuous attributes as numbers. However, a transformation that is meant to be applied, say to continuous attributes, cannot be applied over a dataset that contains both continuous and categorical attributes (i.e., a dataset containing both numbers and strings); Scikit-learn cannot deal with arrays of mixed types. In that case, all the categorical attributes need to be encoded into numeric representations, even if they represent a categorical value. That is, a value can be a number but represent a category. This is what happens when Normalization and Discretization are meant to be applied to a dataset containing mixed types of attributes. In order for them to be applied to datasets of mixed types, an Encoding transformation needs to be applied first. A similar constraint

[3]For a full list see:
https://josephgiovanelli.github.io/DOLAP_2021_supplementary_material/
[4]https://scikit-learn.org

[5]Normalization transformations are the only ones that Scikit-learn can apply on datasets with missing values.

Table 1: Precedence order between pairs of transformations, represented independently for each phase.

(a) Compatible precedence.

	E	*N*	*D*	*I*	*R*	*F*
E		1	1	0	1	1
N	0		0	0	0	0
D	0	0		0	0	0
I	1	0	1		1	1
R	0	0	0	0		0
F	0	0	0	0	0	

(b) Meaningful precedence.

	E	*N*	*D*	*I*	*R*	*F*
E		0	0	0	0	0
N	0		X	0	1	0
D	0	X		0	0	0
I	1	1	1		1	1
R	0	0	0	0		0
F	0	0	0	0	0	

(c) Promising precedence.

	E	*N*	*D*	*I*	*R*	*F*
E		0	0	0	0	0
N	0		0	0	0	1
D	0	0		0	0	1
I	0	0	0		0	0
R	0	0	0	0		0
F	0	0	0	0	0	

E - Encoding; ***N*** - Normalization; ***D*** - Discretization; ***I*** - Imputation; ***R*** - Rebalancing; ***F*** - Feature Engineering.
1 - a precedence edge exists between the row and the column, 0 - a precedence edge does not exist between the row and the column, X - the combination is meaningless.

is imposed when considering Rebalancing and Feature Engineering, since these transformations do not accept inputs containing strings (i.e., representing a categorical type). For the rest of the pairs of transformations there are no constraints imposed by the framework, thus any order of such transformations is permitted, reflected by a 0 in Table 1a. The graph obtained in this case exclusively corresponds to the limitations of Scikit-learn (as a matter of fact, if another framework were to be chosen, it may have looked differently).

3.3 Heuristic rules

In the previous section, we derived a precedence based on the constraints of the framework. Now, we want to study the precedence independently of the framework, and find *meaningful pairs*. That is, for every given pair, we want to find the relative order, based on generic (i.e., based on the literature), domain-independent knowledge about transformations and their applicability. To this end, note that some of the constraints imposed by the framework may be contradicted here, but this will be considered later when taking the union of both of the graphs (see Section 3.5). Table 1b shows the results obtained.

Observe that the constraints on the Imputation transformation still hold, that is, it is correct to apply Imputation first when combining it with another transformation, including when combining it with Normalization. However, the precedences of Encoding are not present, hence not considering the framework, an Encoding transformation makes sense to be combined in any order with the rest of transformations, except Imputation. For the sake of an example, Discretization combined with Encoding is a meaningful combination (when a mixed type dataset is considered), but incompatible from the point of view of Scikit-learn. Furthermore, notice that combining Discretization with Normalization does not make sense, due to the fact that after the Discretization step, continuous attributes are transformed into Categorical ones, and hence Normalization cannot be applied. Similarly, applying Normalization first, changes the scale of the values and hence impacts the Discretization step. Finally, another meaningful precedence can be derived when combining Normalization with Rebalancing. In this case, Normalization should be applied first, since otherwise Rebalancing would impact the scale of the values to be normalized.

3.4 Learnt rules

A third viewpoint to consider is that of learning a promising order by empirically studying the impact of the combinations on the final result of the analysis, using different classification problems in the training. Specifically, we considered three classification algorithms (i.e., *NB*, *RF*, *KNN*) on 60 out of 72 classification problems from the OpenML-CC18 benchmark [22]. We omitted 12 datasets since, due to their size, the exhaustive search (see Section 4.2) was taking unfeasible amount of computing time.

We could try to learn the precedence of every pair of transformations, but would just be a waste of resources, because we can see in Table 1a and 1b, that some precedences are already decided for one reason or another. Hence, only pairs of transformations with a 0 for both directions (in both Table 1a and 1b) need to be studied further. That is, they make sense to be combined together, but a precedence order could not be determined through *framework-related* or *heuristic rules*. Thus, for instance pairs involving Encoding are not considered in this phase, since for them an order is already imposed by the framework (see Table 1a). To this end, the pairs of transformations we consider for the third precedence graph include only $\{F, N\}$, $\{F, D\}$, $\{F, R\}$, and $\{R, D\}$.

3.4.1 Learning method. For every selected pair of transformations, for a given classification algorithm, we check which order of the pair improves most the performance (e.g., predictive accuracy) of the algorithm over all the datasets considered. To this end, for each dataset we get a precedence order that gives better results (i.e., promising precedence) in terms of predictive accuracy (other metrics could have been used as well).

To find a promising precedence order between a given pair of transformations, we follow the steps shown in Algorithm 1. To be able to compute the impact of transformations, we first take the accuracy of the classification algorithm over the original dataset, and use it as baseline for comparison (see line 1). Next for each precedence order (lines 2-3), we find both, the pipelines with the best parametrisations using Sequential Model-Based Optimization (SMBO) [12], and the accuracy of the ML algorithm (with default parametrisation) on the transformed datasets. Finally, the winner pipeline is selected (line 5). Notice that some previous validity check (line 4) is necessary because of the way SMBO works. When optimizing a pre-processing pipeline (in this case consisting of only two transformations), given a budget in terms of time or number of iterations, SMBO tries many different configurations of parameters for each of the operators implementing the transformations, yet, one of the possible configurations is also not instantiating at all a transformation (or both) in the pipeline (i.e., represented with a $\varnothing$ symbol). Hence, out

Algorithm 1 Find a promising pipeline prototype for transformations T_1 and T_2

Require: d, a # dataset, classification algorithm
Require: $T_1 \rightarrow T_2, T_2 \rightarrow T_1$ # precedence orders of a pair of transformations
1: $acc_{baseline} = Acc(d, a)$; # get baseline performance of algorithm on d
2: $[pipeline_{T_1 \rightarrow T_2}, acc_{T_1 \rightarrow T_2}] = SMBO(T_1 \rightarrow T_2, d, a)$ # get pipeline and accuracy for $T_1 \rightarrow T_2$
3: $[pipeline_{T_2 \rightarrow T_1}, acc_{T_2 \rightarrow T_1}] = SMBO(T_2 \rightarrow T_1, d, a)$ # get pipeline and accuracy for $T_2 \rightarrow T_1$
4: **if** $IsValid(acc_{T_1 \rightarrow T_2}, acc_{T_2 \rightarrow T_1}, acc_{baseline})$ **then** # see Table 2 for the rules applied
5: **return** $Winner([pipeline_{T_1 \rightarrow T_2}, acc_{T_1 \rightarrow T_2}], [pipeline_{T_2 \rightarrow T_1}, acc_{T_2 \rightarrow T_1}])$ # see column *Winner prototype* in Table 2
6: **else**
7: **return** $\emptyset$
8: **end if**

Table 2: Validation rules.

Nr.	Pipeline 1	Pipeline 2	Valid result	Valid score	Winner prototype
1.	$\emptyset \rightarrow \emptyset$	$\emptyset \rightarrow \emptyset$	Draw	$acc_{baseline}$	Baseline
2.	$\emptyset \rightarrow \emptyset$	$conf_{T_2} \rightarrow \emptyset$	Draw	$acc_{baseline}$	Baseline
3.	$\emptyset \rightarrow \emptyset$	$\emptyset \rightarrow conf_{T_1}$	Draw	$acc_{baseline}$	Baseline
4.	$\emptyset \rightarrow \emptyset$	$conf_{T_2} \rightarrow conf_{T_1}$	Draw $conf_{T_2} \rightarrow conf_{T_1}$	$acc_{baseline}$ $acc_{T_2 \rightarrow T_1}$	Baseline $T_2 \rightarrow T_1$
5.	$\emptyset \rightarrow conf_{T_2}$	$\emptyset \rightarrow \emptyset$	Draw	$acc_{baseline}$	Baseline
6.	$\emptyset \rightarrow conf_{T_2}$	$conf_{T_2} \rightarrow \emptyset$	Draw $\emptyset \rightarrow conf_{T_2}$ $conf_{T_2} \rightarrow \emptyset$	acc_{T_2}	T_2 T_2 T_2
7.	$\emptyset \rightarrow conf_{T_2}$	$\emptyset \rightarrow conf_{T_1}$	Draw	acc_{T_2} or acc_{T_1}	T_1 *or* T_2
8.	$\emptyset \rightarrow conf_{T_2}$	$conf_{T_2} \rightarrow conf_{T_1}$	Draw $conf_{T_2} \rightarrow conf_{T_1}$	acc_{T_2} $acc_{T_2 \rightarrow T_1}$	T_2 $T_2 \rightarrow T_1$
9.	$conf_{T_1} \rightarrow \emptyset$	$\emptyset \rightarrow \emptyset$	Draw	$acc_{baseline}$	Baseline
10.	$conf_{T_1} \rightarrow \emptyset$	$conf_{T_2} \rightarrow \emptyset$	Draw	acc_{T_1} or acc_{T_2}	T_1 *or* T_2
11.	$conf_{T_1} \rightarrow \emptyset$	$\emptyset \rightarrow conf_{T_1}$	Draw $conf_{T_1} \rightarrow \emptyset$ $\emptyset \rightarrow conf_{T_1}$	acc_{T_1}	T_1 T_1 T_1
12.	$conf_{T_1} \rightarrow \emptyset$	$conf_{T_2} \rightarrow conf_{T_1}$	Draw $conf_{T_2} \rightarrow conf_{T_1}$	acc_{T_1} $acc_{T_2 \rightarrow T_1}$	T_1 $T_2 \rightarrow T_1$
13.	$conf_{T_1} \rightarrow conf_{T_2}$	$\emptyset \rightarrow \emptyset$	Draw $conf_{T_1} \rightarrow conf_{T_2}$	$acc_{baseline}$ $acc_{T_1 \rightarrow T_2}$	Baseline $T_1 \rightarrow T_2$
14.	$conf_{T_1} \rightarrow conf_{T_2}$	$conf_{T_2} \rightarrow \emptyset$	Draw $conf_{T_1} \rightarrow conf_{T_2}$	acc_{T_2} $acc_{T_1 \rightarrow T_2}$	T_2 $T_1 \rightarrow T_2$
15.	$conf_{T_1} \rightarrow conf_{T_2}$	$\emptyset \rightarrow conf_{T_1}$	Draw $conf_{T_1} \rightarrow conf_{T_1}$	acc_{T_1} $acc_{T_1 \rightarrow T_2}$	T_1 $T_1 \rightarrow T_2$
16.	$conf_{T_1} \rightarrow conf_{T_2}$	$conf_{T_2} \rightarrow conf_{T_1}$	Draw $conf_{T_1} \rightarrow conf_{T_2}$ $conf_{T_2} \rightarrow conf_{T_1}$	acc_{T_1} or acc_{T_2} $acc_{T_1 \rightarrow T_2}$ $acc_{T_2 \rightarrow T_1}$	T_1 *or* T_2 $T_1 \rightarrow T_2$ $T_2 \rightarrow T_1$

$\emptyset$ - SMBO finds a better result without instantiating a transformation (or both) in the pair.
$conf_T$ - The configuration (i.e., operator and its parameters) found for T by SMBO.
acc_T - The accuracy of the ML algorithm over the data transformed with a pipeline T.

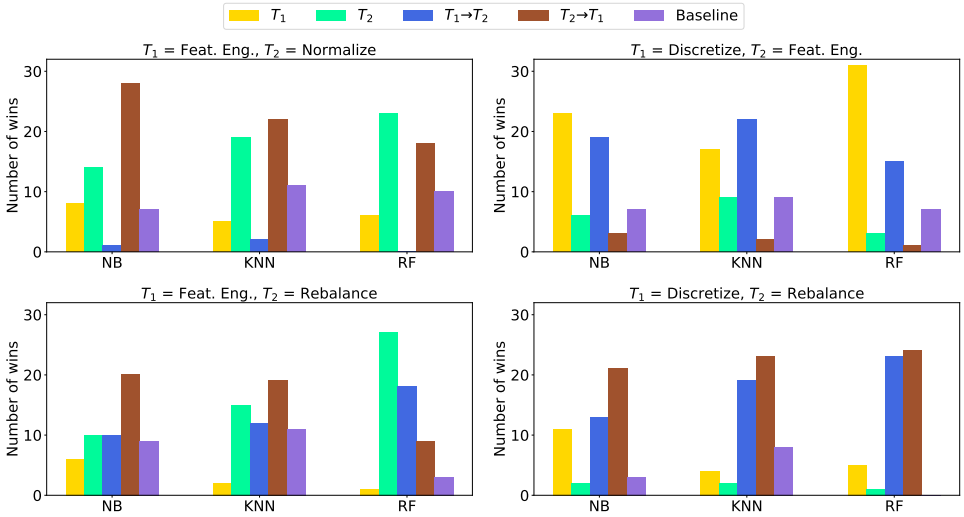

Figure 4: Number of datasets for which a given pipeline prototype is declared the winner.

Table 3: Signif. test for determining the order between pairs of transformations.

T_1	T_2	$T_1 \rightarrow T_2$	$T_2 \rightarrow T_1$	alpha	p-value
F	N	3	68	0.05	**0**
D	F	56	6	0.05	**0**
F	R	40	48	0.05	7.71e-01
D	R	55	68	0.05	8.60e-01

of two pairs of transformations — consisting of the same set of transformations but in different order, where one or both of them may not be instantiated, SMBO may generate 16 possible scenarios, which are listed in Table 2. If among the optimized pairs (with different order) obtained from SMBO, one or both of them contain a $\varnothing$ operator, their results are considered valid, if they have equal scores (i.e., a draw). Otherwise, if say, the first has a higher score, it means that during the optimization phase it was more advantageous than the second, since it could find a configuration that in fact should have been found by both of the pairs. In our SMBO runs, such invalid results account for less than 10% and in those cases datasets are discarded from the study (line 7).

In Table 2, the first two columns denote the pipeline instantiations for the respective orders of transformations (i.e., $T_1 \rightarrow T_2$ and $T_2 \rightarrow T_1$). Next, *Valid result* denotes the valid expected results when comparing the pipelines. For example, the first rule indicates that if both operations in the pipelines are not instantiated during the optimization, a valid result is a draw, and a *Valid score* for the respective result is the baseline accuracy. Finally, the *Winner prototype* is the prototype corresponding to the pipeline that achieves a valid result, which is the baseline in our example.

For better understanding, let us also consider rule 2 in Table 2. Running SMBO, the best result for the first pair is chosen to be the pipeline $\varnothing \rightarrow \varnothing$, and for the second pair (i.e., the opposite order), the pipeline $T_2 \rightarrow \varnothing$. In this case, the results of these pipelines should be equal, and the score should be that of the baseline (where no transformations are applied). Otherwise, if say, the score of the second pipeline is higher than that of the first, it would mean that for the first pair, SMBO was not given enough time to find the best pipeline, since it should have found $T_2 \rightarrow \varnothing$, which has a better score. The same logic applies also for the other rows where a $\varnothing$ operator is involved.

3.4.2 Results. Applying Algorithm 1, we obtain a promising order for each pair of transformations considered (i.e., $\{F, N\}$, $\{F, D\}$, $\{F, R\}$, $\{R, D\}$). Since SMBO is a randomized algorithm we experimented with (a) running it several times splitting the budget, and (b) running it only once with a the total budget. For the experiments considered, no significant differences where observed, therefore we opted for running it once with all the budget (i.e., 200 seconds per run), which allows for more configurations to be visited in a single run. Aggregating all the results, Figure 4 shows the number of datasets, for which a given combination (see Table 2, column *Winner prototype* for the list of labels) is selected as the winner. For instance, for the pair $\{F, N\}$ (i.e., Feature Engineering, Normalization), the most winning prototype for *KNN* and *NB* is $N \rightarrow F$, which means that for most datasets, better results are obtained if Normalization is applied before Feature Engineering. Next, the second position for *KNN* and *NB*, and the first for *RF* is only N, which means that for many datasets, in different algorithms, it is better to apply only Normalization without combining it with Feature Engineering. The third position is for $\varnothing \rightarrow \varnothing$, which means that better results are obtained if no transformation is applied. The remaining prototypes winning in some datasets are F (only Feature Engineering), and $F \rightarrow N$ (Feature Engineering preceding Normalization). Finally, for three datasets, which are omitted for simplicity, there were no winning pipelines (i.e., pipelines resulted in a draw). Since we want to find the best order for a given pair of transformations, we specifically focus on the performances of the corresponding precedence orders (i.e., $T_1 \rightarrow T_2$ versus $T_2 \rightarrow T_1$) of a pair of transformations, and disregard the comparison of the other possibilities (i.e., T_1, T_2, or *Baseline*) since they are not relevant for declaring a winner. Hence, we check whether the difference between these pipelines are statistically significant by running a binomial test assuming a theoretical probability of 0.5. The results are shown in Table 3. In summary, the results from Table 3 indicate that, with 95% confidence we can assume that for the pair $\{F, N\}$, $N \rightarrow F$ performs better than $F \rightarrow N$, hence Normalization should precede Feature Engineering. Similarly, for $\{D, F\}$, $D \rightarrow F$ performs better than $F \rightarrow D$, hence Discretization should precede Feature Engineering. Finally, for the remaining pairs, $\{F, R\}$ and $\{R, D\}$, a precedence order can not be assumed since the results obtained are not significant. Using these results, we create the *promising precedence* adjacency matrix shown in Table 1c, where as one can observe, precedence edges are introduced for $\{N, F\}$ and $\{D, F\}$, but no edges exist neither for $\{F, R\}$, nor for $\{R, D\}$.

3.5 Pipeline prototype composition

To generate the final pipeline prototypes, in this step we combine all the matrices generated by the previous steps. That is, we take the union of the edges (represented by `1`'s) from the matrices in Table 1 (a,b,c), and create a new final adjacency matrix, shown in Table 4. This is the matrix that will allow us to generate the final effective pipelines.

Table 4: Union of rules from Table 1.

	E	***N***	***D***	***I***	***R***	***F***
E		1	1	0	1	1
N	0		X	0	1	1
D	0	X		0	0	1
I	1	1	1		1	1
R	0	0	0	0		0
F	0	0	0	0	0	

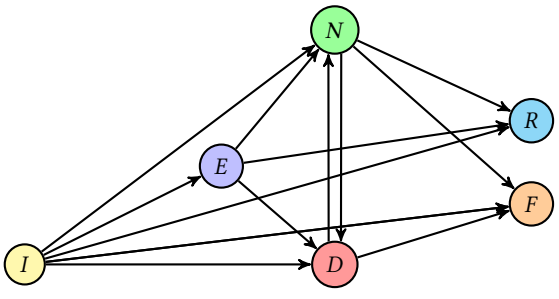

Figure 5: Precedence graph generated from Table 4.

Table 5: Effective pipeline prototypes.

ID	Pipeline prototype
1	$I \rightarrow E \rightarrow N \rightarrow R \rightarrow F$
2	$I \rightarrow E \rightarrow N \rightarrow F \rightarrow R$
3	$I \rightarrow E \rightarrow R \rightarrow D \rightarrow F$
4	$I \rightarrow E \rightarrow D \rightarrow R \rightarrow F$
5	$I \rightarrow E \rightarrow D \rightarrow F \rightarrow R$

Observing the table, one can realize that for pairs $\{F, R\}$ and $\{R, D\}$, no precedence edges exist. This means that these pairs are somewhat equally relevant from either direction (any order), and thus when generating the final prototypes, both options should appear.

For a better reading, in Figure 5, we visualize Table 4 in form of a graph, where nodes represent the kinds of transformations and the directed edges represent a precedence order between them. Out of the graph, we generate the final pipeline prototypes by taking all the maximum length variations (ordered arrangements without repetition) of the nodes, respecting the precedence rules (i.e., not contradicting the direction of existing edges). The result is the set of five pipeline prototypes shown in Table 5. This set consisting of *compatible*, *meaningful* and *promising* pairs of transformations is the set of recommended *effective pipeline prototypes*.

4 EVALUATION

The aim of our experimental study is three-fold:

(1) Check whether there exists a universal pipeline prototype that works best for every classification problem considered (i.e., dataset and ML alg.) (Section 4.1).
(2) Assess and compare the performance of the effective pipelines constructed using our method against the set of exhaustively generated pipeline prototypes (Section 4.2).
(3) Assess and compare the impact of dedicating a portion of the optimization time to the effective pipelines constructed using our method, with the impact of using the whole optimization time for the hyper-parameters of the ML algorithm (Section 4.3).

The experiments were performed on an Intel Core i7 machine with 12 cores, running at 3.20 GHz with 64 GB of main memory. As a platform for running the SMBO optimization algorithm we use HyperOpt. Furthermore, the datasets used in the experiments are the ones from the OpenML-CC18 suite (see Section 3.4). Finally, the classification algorithms considered are *NB*, *KNN*, and *RF*. All the experiments for a single algorithm, on average took approximately two weeks[6].

4.1 Universal pipeline prototype

The goal of this experiment is to demonstrate the difficulty of blindly finding the right pipeline prototype (i.e., without considering any meaningful or promising precedence). In Table 6, we list the exhaustive set of pipeline prototypes generated considering the compatible precedence graph in Table 1a (i.e., 24 compatible permutations). In a real scenario, this number is too high for splitting the time budget in order to optimize them. Yet, for the sake of this experiment, we exhaustively optimize all the prototypes, for each dataset. Thus, for each pipeline prototype and for each dataset, the SMBO algorithm is configured to assign a 200 seconds time budget to the phase of instantiating and optimizing the pipeline prototype, and another 200 seconds to the phase of optimizing the hyper-parameters of the ML algorithm.

[6]The source code and the datasets for reproducing the experiments can be found in https://github.com/josephgiovanelli/effective_preprocessing_pipeline_evaluation

Table 6: Exhaustive set of pipeline prototypes generated using the compatible precedence graph of Table 1a.

ID	Pipeline prototype	ID	Pipeline prototype
1	$I \rightarrow E \rightarrow N \rightarrow D \rightarrow F \rightarrow R$	13	$I \rightarrow E \rightarrow F \rightarrow N \rightarrow D \rightarrow R$
2	$I \rightarrow E \rightarrow N \rightarrow D \rightarrow R \rightarrow F$	14	$I \rightarrow E \rightarrow F \rightarrow N \rightarrow R \rightarrow D$
3	$I \rightarrow E \rightarrow N \rightarrow F \rightarrow D \rightarrow R$	15	$I \rightarrow E \rightarrow F \rightarrow D \rightarrow N \rightarrow R$
4	$I \rightarrow E \rightarrow N \rightarrow F \rightarrow R \rightarrow D$	16	$I \rightarrow E \rightarrow F \rightarrow D \rightarrow R \rightarrow N$
5	$I \rightarrow E \rightarrow N \rightarrow R \rightarrow D \rightarrow F$	17	$I \rightarrow E \rightarrow F \rightarrow R \rightarrow N \rightarrow D$
6	$I \rightarrow E \rightarrow N \rightarrow R \rightarrow F \rightarrow D$	18	$I \rightarrow E \rightarrow F \rightarrow R \rightarrow D \rightarrow N$
7	$I \rightarrow E \rightarrow D \rightarrow N \rightarrow F \rightarrow R$	19	$I \rightarrow E \rightarrow R \rightarrow N \rightarrow D \rightarrow F$
8	$I \rightarrow E \rightarrow D \rightarrow N \rightarrow R \rightarrow F$	20	$I \rightarrow E \rightarrow R \rightarrow N \rightarrow F \rightarrow D$
9	$I \rightarrow E \rightarrow D \rightarrow F \rightarrow N \rightarrow R$	21	$I \rightarrow E \rightarrow R \rightarrow D \rightarrow N \rightarrow F$
10	$I \rightarrow E \rightarrow D \rightarrow F \rightarrow R \rightarrow N$	23	$I \rightarrow E \rightarrow R \rightarrow D \rightarrow F \rightarrow N$
11	$I \rightarrow E \rightarrow D \rightarrow R \rightarrow N \rightarrow F$	23	$I \rightarrow E \rightarrow R \rightarrow F \rightarrow N \rightarrow D$
12	$I \rightarrow E \rightarrow D \rightarrow R \rightarrow F \rightarrow N$	24	$I \rightarrow E \rightarrow R \rightarrow F \rightarrow D \rightarrow N$

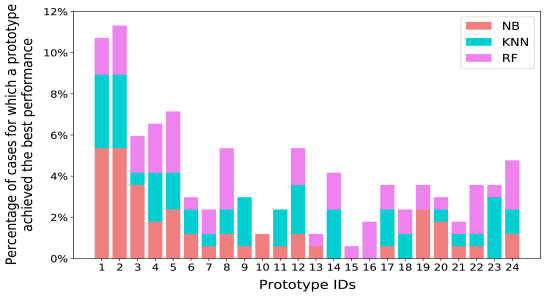

Figure 6: Comparison of the goodness of the exhaustive set of prototypes.

The results obtained are shown in Figure 6. The enumerated prototypes are listed in the ordinate axis and each stacked bar represents the percentage of cases for which that prototype achieved the best performance across different ML algorithms (the contribution of each algorithm is represented with a different color). In an ideal scenario, for a pipeline to be considered *universal*, it should perform best in all or at least most of the cases, which is clearly not happening. Observe that, even the best performing pipeline is only the best in 11% of the cases, which is obviously far from being *universal*. Hence all (or at least several) pipelines need to be evaluated together, in order to obtain optimal results.

4.2 Exhaustive versus effective prototypes

Given that there is no single universal pipeline, one can opt for feeding all the possible prototypes (see Table 6) to the optimization algorithm in order to get the optimal results. As before, we assign a budget of 200 seconds for the optimization of each prototype, hence 80 minutes in total for all the set of 24 *exhaustive prototypes* in order to find the optimal pipeline for every dataset. On the other hand, we take only the five *effective prototypes* resulting from the application of our method and assign just 40 seconds time budget for the optimization of each one of them, hence 200 seconds in total. With the aim of comparing the two, and thus roughly understanding how close we are to the optimal case, in both cases, we dedicated the same time budget (i.e., 200 seconds) for the phase of optimizing the hyper-parameters of the ML

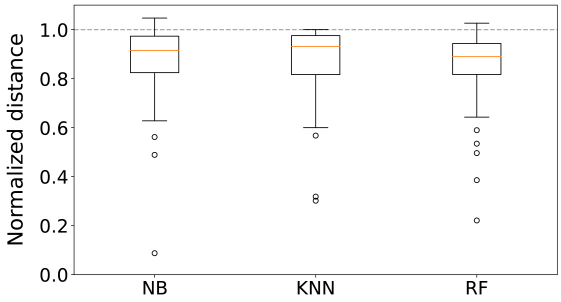

Figure 7: Normalized distances between the scores obtained by optimizing our effective prototypes and the ones obtained optimizing the exhaustive set.

algorithm. In order to evaluate how close the *effective prototypes* are to the *exhaustive ones*, we calculate the *normalized distance* from the result to the optimum:

$$normalized\ distance = \frac{Acc(d_{effective}, a^*) - Acc(d, a)}{Acc(d_{exhaustive}, a^*) - Acc(d, a)}$$

where, $Acc(d, a)$ is the baseline performance (i.e., predictive accuracy of the algorithm a with default hyper-parameters over the original dataset d). $Acc(d_{effective}, a^*)$ is the accuracy of the optimized algorithm a^* over the dataset $d_{effective}$ transformed using the optimized instantiation of the effective set of prototypes (i.e., our approach). Finally, $Acc(d_{exhaustive}, a^*)$ is the accuracy of the optimized algorithm a^* over the dataset $d_{exhaustive}$ transformed using the optimized pipeline instantiation of the exhaustive set of prototypes. The subtraction by $Acc(d, a)$ is done with the aim of weighting the difficulty of a dataset, hence allowing for comparisons in terms of the gain in accuracy. To this end, the bigger the potential gain (denominator) is, the bigger the obtained gain (numerator) must be, for the latter to be relevant.

The results obtained for every dataset and algorithm are shown as boxplots in Figure 7. Observe that, most of the cases are very close to the results obtained using the exhaustive set, the median distances being 91.51%, 93.13%, 88.97%, for NB, KNN, and RF, respectively. In general, in 75% of the cases the chosen pipelines are above 80%, and only few outliers are below 60%. Curiously, in some cases, we outperform the results over the exhaustive set of pipelines, but this is due to the randomness of the optimization algorithm, which unless it is given an unrealistically high budget of time, is not capable of finding the true optimal solution. We discarded the option of assigning a larger budget since this was not practical considering the huge search space and the lack of any guarantee of improvement.

To summarize, the experiment shows that with roughly 24 times less time budget, we can obtain results that are as good as 90% in the median compared to the optimal ones. The raw results (i.e., without the normalized distances) can be found on the aforementioned github page.

4.3 Complementing hyper-parameter optimization with pre-processing

We have just shown that our effective pipeline prototypes have similar impact as the exhaustive prototypes. Now we want to compare the impact of effective prototypes against optimizing only the hyper-parameters of the ML algorithm. That is, we want to examine whether dedicating a part of the optimization budget to the pre-processing pipeline impacts more (positively) the results of the analysis, than using the whole budget for the hyper-parameter optimization[7].

To this end, for the latter we now dedicate the total optimization budget (i.e., 400 seconds), and for the former, inspired by [19], we split the budget 50-50 between the pre-processing pipeline optimization and the hyper-parameter optimization (i.e., 200 seconds for the pre-processing, and 200 seconds for the hyper-parameter optimization). The time for the pre-processing is further split among the five different pipeline prototypes (i.e., 40 seconds each).

To compare the results, we calculate the impact using the formulas below, that correspond to the normalized distance from either pre-processing or hyper-parameter optimization to the maximum improvement that can be achieved, regardless of whether pre-processing is applied or not.

$$pp\ impact = \frac{Acc(d_{effective}, a^*) - Acc(d, a)}{max(Acc(d_{effective}, a^*), Acc(d, a^*)) - Acc(d, a)}$$

$$hp\ impact = \frac{Acc(d, a^*) - Acc(d, a)}{max(Acc(d_{effective}, a^*), Acc(d, a^*)) - Acc(d, a)}$$

where, $Acc(d, a)$ is the baseline accuracy (i.e., predictive accuracy of the algorithm a with default hyper-parameters over the original dataset d). $Acc(d_{effective}, a^*)$ is the accuracy of the optimized algorithm a^* over the dataset $d_{effective}$ transformed using the optimized instantiation of the effective set of prototypes obtained with the method above. Finally, $Acc(d, a^*)$ is the accuracy of the optimized algorithm a^* (i.e, using the entire budget) over the original dataset d.

To obtain relative values that sum to 1, we normalize the impacts dividing them by their sum. For instance, for the pre-processing score we calculate the following:

$$normalized\ pp\ impact = \frac{pp\ impact}{pp\ impact + hp\ impact}$$

We perform the same for the hyper-parameter impact and plot the results obtained for all the algorithms and datasets in Figure 8, where each bar represents the results obtained for a single dataset. The different colors represent the impact values of pre-processing and hyper-parameter optimization.

Observing the bar-charts one can see that (i) dedicating a portion of the budget to pre-processing, brings benefit to the analysis in most of the cases (i.e., 73% of the cases), and (ii) the impact of hyper-parameter optimization, increases with the increase of the number of hyper-parameters of the ML algorithm (e.g., hyper-parameter optimization impacts more RF than NB). Overall, we can conclude that pre-processing is a critical step that once effectively applied may have a high positive impact on the final result of the analysis.

5 CONCLUSIONS AND FUTURE WORK

In this work, we first studied the overall impact of transformations when combined into pre-processing pipelines and then delved into examining the impact of the precedence order between pairs of various transformations. As a result, we defined a method that allows to generate effective pre-processing pipelines. That is, pipelines that consist of,

[7]To enable the application of the ML algorithms on all the datasets, whenever required, we apply the necessary transformation (e.g, imputation or encoding).

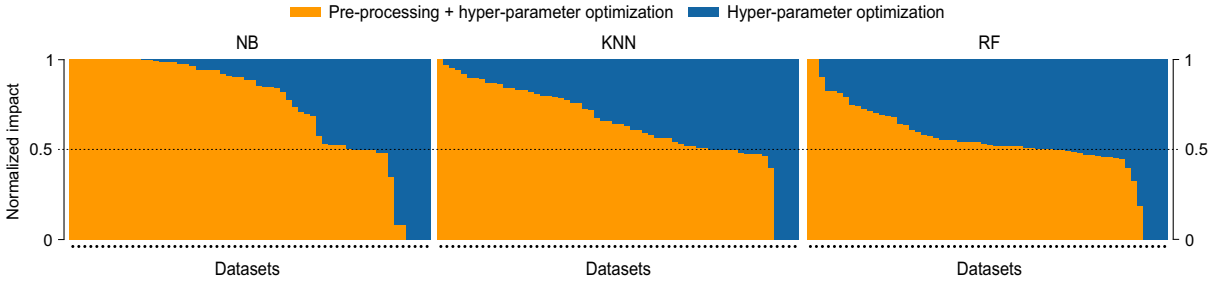

Figure 8: The impact of dedicating a portion of the optimization budget to pre-processing compared to using the whole optimization budget for the hyper-parameter optimization.

(i) compatible pairs of transformations with respect to the framework used, (ii) meaningful pairs of transformations in terms of general knowledge (best practices), and (iii) promising pairs of transformations that once applied are expected to provide higher overall impact (domain knowledge). An extensive evaluation on 60 datasets with heterogeneous characteristics, from sample size to feature types, and a set of classification algorithms (i.e., *NB*, *KNN*, *RF*), showed that our devised pipeline prototypes give promising results. More specifically, we were able to observe that:

- The overall impact of optimizing pre-processing is not negligible and it may boost the performance of the overall analytics (e.g., predictive accuracy).
- There is no universal pre-processing pipeline prototype that works best for every dataset and algorithm.
- With 24 times less time budget, our proposed pipeline prototypes were able to obtain results that were as good as 90% in the median of the optimal ones found through an exhaustive search.
- Dedicating a portion of the time to the pre-processing optimization, instead of dedicating it entirely to hyper-parameter optimization may boost the final result of the analysis. On average, in 73% of the cases including pre-processing in the optimization, outperformed the results of only optimizing hyper-parameters.

The results indicate that pre-processing can boost the performance of the ML algorithm. Hence, it must be considered as an integral part of the data analytics optimization process. As immediate future work, we intend to: (i) study the datasets that do not react well, or at all, to data pre-processing (see Figure 8), and find out the reasons why they do not do so, (ii) increase the number of datasets and algorithms used to further validate our approach, and (iii) extend our approach with a meta-learning module, that given some dataset characteristics [3] is capable of recommending the most useful prototype among the five effective ones.

ACKNOWLEDGMENTS

This work was supported by the GENESIS project, funded by the Spanish Ministerio de Ciencia e Innovación under project TIN2016-79269-R. We thank University of Bologna for issuing a grant for author's research stay at Universitat Politècnica de Catalunya. Finally, we thank Matteo Golfarelli for his comments and feedback on this work.

REFERENCES

[1] J. Bergstra, D. Yamins, and D. D. Cox. 2013. Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures *(ICML'13)*. 115–23.

[2] Laure Berti-Équille. 2019. Learn2Clean: Optimizing the Sequence of Tasks for Web Data Preparation *(WWW '19)*. 2580–2586.

[3] Besim Bilalli, Alberto Abelló, and Tomàs Aluja-Banet. 2017. On the predictive power of meta-features in OpenML. *Int. J. Appl. Math. Comput. Sci.* 27, 4 (2017), 697–712.

[4] Besim Bilalli, Alberto Abelló, Tomàs Aluja-Banet, Rana Faisal Munir, and Robert Wrembel. 2018. PRESISTANT: Data Pre-processing Assistant *(CAiSE Forum '18)*. 57–65.

[5] Besim Bilalli, Alberto Abelló, Tomàs Aluja-Banet, and Robert Wrembel. 2018. Intelligent assistance for data pre-processing. *Comput. Stand. Interfaces* 57 (2018), 101–109.

[6] Besim Bilalli, Alberto Abelló, Tomàs Aluja-Banet, and Robert Wrembel. 2019. PRESISTANT: Learning based assistant for data pre-processing. *Data Knowl. Eng.* 123 (2019).

[7] Xu Chu, John Morcos, Ihab F. Ilyas, Mourad Ouzzani, Paolo Papotti, Nan Tang, and Yin Ye. 2015. KATARA: A Data Cleaning System Powered by Knowledge Bases and Crowdsourcing *(SIGMOD '15)*. 1247–1261.

[8] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Tobias Springenberg, Manuel Blum, and Frank Hutter. 2015. Efficient and Robust Automated Machine Learning *(NeurIPS '15)*. 2962–2970.

[9] Matthias Feurer, Jost Tobias Springenberg, and Frank Hutter. 2015. Initializing Bayesian Hyperparameter Optimization via Meta-Learning *(AAAI '15)*. 1128–1135.

[10] Nicolo Fusi, Rishit Sheth, and Melih Elibol. 2018. Probabilistic Matrix Factorization for Automated Machine Learning *(NeurIPS'18)*. 3352–3361.

[11] Floris Geerts, Giansalvatore Mecca, Paolo Papotti, and Donatello Santoro. 2013. The LLUNATIC Data-cleaning Framework. *PVLDB End.* 6, 9 (July 2013), 625–636.

[12] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. 2011. Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization*. Springer, 507–523.

[13] Zhongjun Jin, Michael R. Anderson, Michael Cafarella, and H. V. Jagadish. 2017. Foofah: A Programming-By-Example System for Synthesizing Data Transformation Programs *(SIGMOD '17)*. 1607–1610.

[14] Zuhair Khayyat, Ihab F. Ilyas, Alekh Jindal, Samuel Madden, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Si Yin. 2015. BigDansing: A System for Big Data Cleansing *(SIGMOD '15)*. 1215–1230.

[15] L. Kotthoff, Colleen Thornton, Holger Hoos, F. Hutter, and Kevin Leyton-Brown. 2017. Auto-WEKA 2.0: Automatic model selection and hyperparameter optimization in WEKA. *Journal of Machine Learning Research* 18 (03 2017), 1–5.

[16] Sanjay Krishnan, Jiannan Wang, Eugene Wu, Michael J. Franklin, and Ken Goldberg. 2016. ActiveClean: Interactive Data Cleaning For Statistical Modeling. *PVLDB* 9, 12 (2016), 948–959.

[17] M. Arthur Munson. 2012. A Study on the Importance of and Time Spent on Different Modeling Steps. *SIGKDD Explor. Newsl.* 13, 2 (2012), 65–71.

[18] Randal S. Olson, Nathan Bartley, Ryan J. Urbanowicz, and Jason H. Moore. 2016. Evaluation of a Tree-based Pipeline Optimization Tool for Automating Data Science *(GECCO '16)*. 485–492.

[19] Alexandre Quemy. 2020. Two-stage optimization for machine learning workflow. *Information Systems* 92 (2020), 101483.

[20] Zeyuan Shang, Emanuel Zgraggen, Benedetto Buratti, Ferdinand Kossmann, Philipp Eichmann, Yeounoh Chung, Carsten Binnig, Eli Upfal, and Tim Kraska. 2019. Democratizing Data Science through Interactive Curation of ML Pipelines *(SIGMOD '19)*. 1171–1188.

[21] Chris Thornton, Frank Hutter, Holger H. Hoos, et al. 2013. Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms. In *KDD*. 847–855.

[22] Joaquin Vanschoren, Jan N. van Rijn, Bernd Bischl, and Luis Torgo. 2013. OpenML: Networked Science in Machine Learning. *SIGKDD Explorations* 15, 2 (2013), 49–60.