Data pre-processing pipeline generation for AutoETL

Joseph Giovanelli[1,*], Besim Bilalli[b], Alberto Abelló[b]

[a]*University of Bologna, Bologna, Italy*
[b]*Universitat Politècnica de Catalunya, BarcelonaTech, Barcelona, Spain*

Abstract

Data pre-processing plays a key role in a data analytics process (e.g., applying a classification algorithm on a predictive task). 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 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 prototypes can be instantiated and optimized e.g., using Bayesian Optimization. In this work, we study the impact of transformations when chained together into prototypes, and the impact of transformations when instantiated via various operators. We develop and scrutinize a generic method that allows to generate pre-processing pipelines, as a step towards AutoETL. We make use of rules that enable the construction of prototypes (i.e., define the order of transformations), and rules that guide the instantiation of the transformations inside the prototypes (i.e., define the operator for each transformation). The optimization of our effective pipeline prototypes provide results that compared to an exhaustive search, get 90% of the predictive accuracy in the median, but with a time cost that is 24 times smaller.

Keywords: data pre-processing pipelines, data analytics

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 are extracted in a raw format from different sources and then 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 [1]), 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

[*] Corresponding author

Email addresses: `j.giovanelli@unibo.it` (Joseph Giovanelli), `bbilalli@essi.upc.edu` (Besim Bilalli), `aabello@essi.upc.edu` (Alberto Abelló)

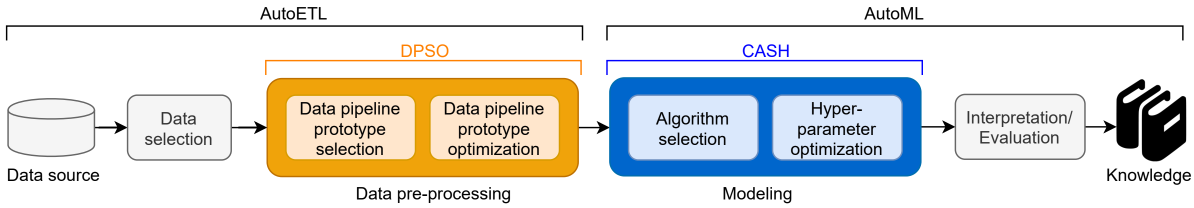
AutoETL
AutoML
DPSO
CASH
Data source
Data selection
Data pipeline prototype selection
Data pipeline prototype optimization
Data pre-processing
Algorithm selection
Hyper-parameter optimization
Modeling
Interpretation/ Evaluation
Knowledge

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

acceptable result is obtained — *ML model*. This whole process requires expertise and is particularly challenging for novice, inexperienced data scientists for whom hand-tuning is no longer an option.

Recent developments in algorithm configuration have raised the efficiency and effectiveness of automatic search, and therefore, for instance, AutoML is now considered a prominent technique for finding optimal models. Some AutoML frameworks [2, 3], mix-in the pre-processing during the optimization, but they are typically limited to very few transformations or do not consider all the data processing phases (e.g., extraction, selection, loading), thus in a way overlooking it. Inspired from [4], we contend that there is need for more generic AutoETL techniques, encompassing all the phases of the ETL process [5]; from its inception via *data extraction*, to the intermediate phase of *data transformation*, up to the final phase, when data reaches its destination, via *data loading*. Assistance is required in every phase [6]. Yet, in a data analytics context, as in the case of this work, the critical automation challenge lies more on the *data transformation* phase (see AutoETL and AutoML in Figure 1), which is related to the pre-processing of the data. In the literature, this particular problem has been often referred to as the Data Pipeline Selection and Optimization problem (DPSO) [7], where a *pipeline prototype* (sequence of transformations, e.g., missing value imputation followed by normalization) is fed to an optimizer 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 [8] 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 ratio of the change in terms 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 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 pipelines, that is, (i) how to order the transformations (i.e., prototype construction), and (ii) which pre-processing operators to consider in the prototype (i.e., prototype instantiation) such that when optimizing the parameters, better results are obtained. The aim of this work is to study these two questions in order to propose a

[1] `https://archive.ics.uci.edu/ml/datasets/Bank+Marketing`

[2] This order is used only for the sake of illustration.

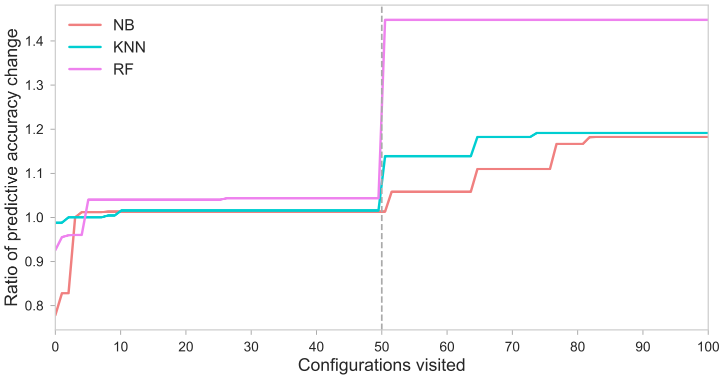

Figure 2: Evolution of predictive accuracy during the optimization process. The first 50 configurations optimize only the hyper-parameters and after the 50th configuration, the pre-processing pipeline is optimized instead.

method for generating effective pre-processing pipeline prototypes that, once instantiated through some optimization technique (e.g., Bayesian optimization), 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 (i.e., binary and multiclass), where the output to be predicted is of Categorical type. Furthermore, this work extends [9], among others with, (i) a new meta-learning step to guide the instantiation of transformations where a model is learned to predict the operators for a given transformation (see Section 3.6), (ii) a set of rules extracted through the meta learning process that mitigate the cold start problem (see Example 8), (iii) a new cross validation to confirm the initial results (see Section 3.4), (iv) a new background section for AutoML and AutoETL (see Section 2), and (v) experiments over new datasets (see Section 4).

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 at least one different pipeline works best for each dataset and algorithm considered, hence showing that there is no universal pipeline that works best for all of them.
- We define a method that given a classification algorithm and a set of pre-processing transformations, is capable of generating the right order between transformations, obtaining effective pre-processing pipeline prototypes, which are then instantiated and further optimized via Bayesian optimization.
- We suggest a meta-learning step, where the relationship between pre-processing operators and dataset characteristics is learned in order to create rules that help with the initial instantiation of the pipeline prototypes.
  - We exemplify our meta-learning study generating simple but not obvious and effective rules for two kinds of transformations, namely, Feature Engineering and Rebalancing.
- We perform a comprehensive evaluation by comparing the performance of optimizing the pipelines generated following our method, and find out that:
  1. with 24 times less time budget, our proposed pipelines obtain results whose median is above 90% of the ones generated via exhaustive search.
  2. 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 remaining of this paper is organized as follows. Section 2 provides a brief background on AutoETL and AutoML. Section 3 presents our method of generating effective pipelines. Section 4 provides an extensive evaluation of the pipelines created using our proposed method. Section 5 discusses the related work. Finally, Section 6 provides the conclusions and future work.

2. Background: AutoETL and AutoML

The abundance of data has led to data analytics being prevalent in many disciplines and domains, but since the number of its applications exceeds the number of qualified experts, more and more non-experts approach the task of data analytics. This has consequently led to the rise of off-the-shelf automated techniques that facilitate its application. AutoML is an umbrella term for automations mainly related to the ML algorithm, and it typically aims to tackle the challenge of Combined Algorithm Selection and Hyperparameter Optimization (CASH). Yet, there is also need for automation in the more generic aspects of ETL [4], which we coin as AutoETL. AutoETL encompasses various steps, however, in this work we focus on the phase that is related to the transformation (pre-processing) of the data, typically formalized as DPSO.

CASH and DPSO can be treated as a single optimization problem [10, 2]. However, we consider them separately because this allows to, (i) reduce the search space and, (ii) to explicitly assign different optimization budgets and/or optimization techniques, depending on their respective impact to the final result of the analysis. Since these problems are similar, the methods initially employed in CASH have been recently considered to solve the DPSO problem too. Therefore, in the following we first delve into more details about CASH and then DPSO. In particular, we formalize them and discuss the methods they employ.

2.1. Combined Algorithm Selection and Hyper-parameter optimization (CASH)

The algorithm selection problem is known to exist for a long time and many approaches have been proposed to solve it [11]. Recently, in the context of ML algorithms, the problem has been extended to include the optimization of the hyperparameters too, and thus has been formalized as follows [10].

Given:

- A data-set D divided into D_{train}, D_{test};
- A set of algorithms $\mathcal{A} = \{A^1, \dots, A^k\}$ with associated hyperparameter spaces $\Lambda^1, \dots, \Lambda^k$;
- And a loss function $\mathcal{L}(A^i_\lambda, D_{train}, D_{test})$;

we are searching for:

$$A^*_{\lambda^*} \in \text{argmin}_{A^i \in \mathcal{A}, \lambda \in \Lambda^i} \mathcal{L}(A^i_\lambda, D_{train}, D_{test}) \tag{CASH}$$

The dataset D is divided into D_{train} and D_{test}, to build and to evaluate the overall performance, calculated through the loss function $\mathcal{L}$. The problem is set up as an optimization problem and, as such, the configuration space is assumed to be known in advance (the set of algorithms $\{A^1, \dots, A^k\}$ and the related hyper-parameter spaces $\{\Lambda^1, \dots, \Lambda^k\}$). The goal is then to find the best algorithm A^* in the set of algorithms and its best hyperparameters λ^* in the related hyperparameter space.

Many optimization techniques have been employed to solve the CASH problem and some of them are: Grid search [12], Random search [13], Simulating annealing [14], Genetic algorithms [15], Bayesian techniques [16], Bandit-based algorithms [17]. However, due to their promising results, Bayesian techniques are perhaps the most popular ones [18, 19]. We explore their details and specifically focus on one of their incarnation, the Sequential Model-Based Optimization (SMBO) algorithm [20].

2.1.1. Sequential Model-Based Optimization (SMBO)

In an optimization problem, we are searching for the best solution among a set of feasible solutions. The latter can be formalized as follows:

$$\max_{x \in B} f(x)$$

, where B contains all the feasible solutions, or candidates, and it is typically d-dimensional ($B \subseteq \mathbb{R}^d$), where $d \in \mathbb{Z}$ and $d > 1$. A specific solution $x \in B$ is evaluated through the function $f : \mathbb{R}^d \longrightarrow \mathbb{R}$, also called the objective function. In general, in these kinds of problems, f has no special structure like concavity or linearity that would make the optimization easier. In fact, it is considered as a "black-box" function, without any knowledge about its behaviour; being that it maps certain inputs, $x \in B$, to certain outputs, $f(x) \in \mathbb{R}$. The goal is then finding an x that maximizes $f(x)$. The naive solution to the problem would be to systematically evaluate all possible candidates x and choosing the one leading to the highest value of $f(x)$, aka *exhaustive search*. Since this evaluates all the potential solutions, it guarantees that it always finds the best one. Nevertheless, generally, it cannot be applied to real problems due to the large number of candidate solutions to be explored, which dwell in a high-dimensional space, and too expensive objective functions. The result is that not all candidates can be evaluated and we have to find a way to wisely choose the most promising ones. Bayesian techniques are part of the family of "surrogate methods", which create a surrogate model to approximate the objective function and thus, choose a point in the search space where to evaluate the objective function [21]. In contrast to the other methods, they build such surrogate models through Bayesian statistics.

In short, Bayesian techniques start by evaluating the objective function on an initial observation point of the search space, then the process becomes iterative: the surrogate model is constructed on the basis of the visited points and through an acquisition function — the Bayesian interpretation of the surrogate, the candidate for the next observation is decided. The process ends when a termination condition is reached, generally expressed through a *budget* represented in terms of the *number of iterations* or *execution time*. Given its iterative nature and the fundamental role of the model, this algorithm is called Sequential Model-Based Optimization [20, 22]. Variations of SMBO exist, depending on the method used to build the surrogate model (e.g., Gaussian Processes, Random Forest Regressions) [23, 24].

2.2. Data Pipeline Selection and Optimization (DPSO)

DPSO was formalized for the first time in [25], where some new concepts were introduced. For instance, a pre-processing *pipeline prototype* or *logical pipeline* is defined as a sequence of kinds of transformations, where each represents a logical concept that can be implemented/instantiated by one or more operators. The prototype thus, defines only the order between kinds of transformations, without specifying the concrete operators nor their parameters. Yet, the potential operators of each kind and their corresponding parameter search spaces need to be known in advance. Solving the DPSO problem translates to finding the right instantiation and configuration for each kind of transformation in the pipeline prototype (i.e., optimal operator and optimal parameter values), which is called pre-processing *pipeline* or *physical pipeline*.

Formally, given:

- A data-set D divided into D_{train}, D_{test};
- A data pipeline prototype P with a configuration space $\mathcal{P}$;
- The algorithm A, for which the given pipeline P transforms the data;
- And a loss function $\mathcal{L}(P, A, D_{train}, D_{test})$;

we are searching for:

$$P^* \in \operatorname{argmin}_{P \in \mathcal{P}} \mathcal{L}(P, A, D_{train}, D_{test}) \tag{DPSO}$$

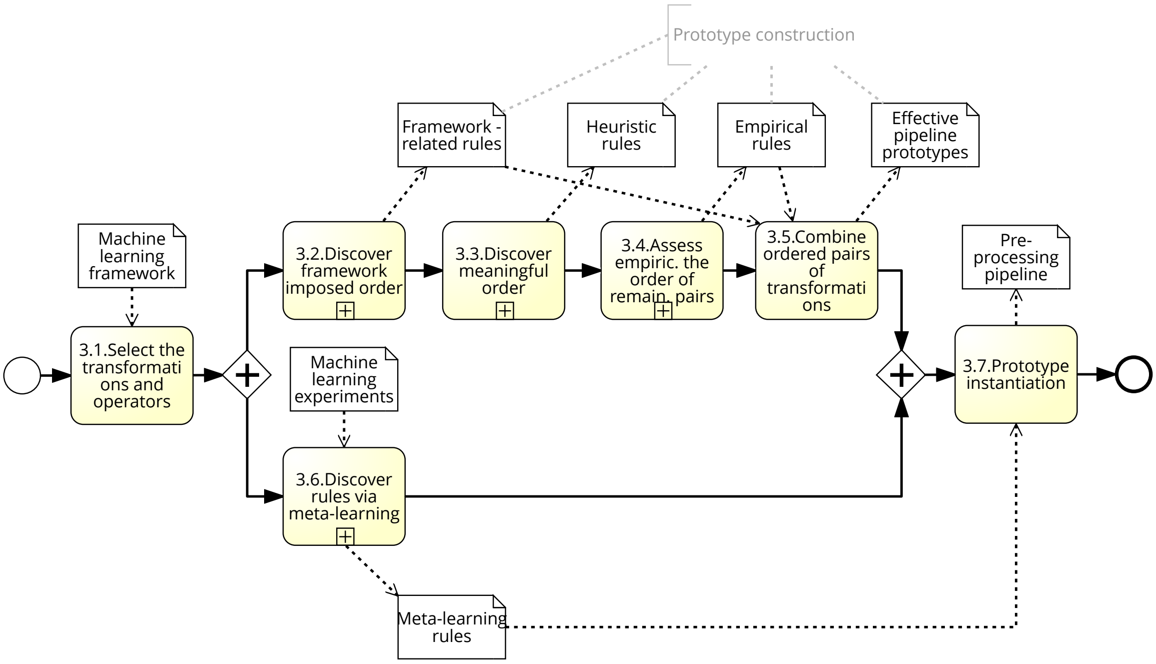
Prototype construction
Framework - related rules
Heuristic rules
Empirical rules
Effective pipeline prototypes
Machine learning framework
3.2.Discover framework imposed order
3.3.Discover meaningful order
3.4.Assess empiric. the order of remain. pairs
3.5.Combine ordered pairs of transformati ons
Pre- processing pipeline
3.1.Select the transformati ons and operators
Machine learning experiments
3.7.Prototype instantiation
3.6.Discover rules via meta-learning
Meta-learning rules

Figure 3: A method for generating pre-processing pipelines.

Notice that a prototype imposes an order between the kinds of transformations, but this is an additional problem that is not dealt within DPSO, since it is assumed to be given as part of the input. This is in fact a limitation of the current approaches in DPSO, in that the order of kinds of transformations are fixed a priori without sufficiently studying the potential effectiveness of different alternatives.

2.2.1. SMBO as solver for DPSO

Since DPSO is formalized as an optimization problem, SMBO has been proposed as a valid solver [20]. In the previous section, we saw the application of SMBO to CASH, but in fact the process of selecting the best algorithm, and its hyperparameters configuration, is identical to selecting the best operator for a transformation, and its parameter configuration. Yet, DPSO requires one more layer, in that transformations need to be chained together into a pipeline. To this end, given a pipeline prototype as input and a budget either in terms of time or number of iterations, SMBO can be configured to iterate over different configurations until a near to optimal physical pipeline is found. The objective function of the pipeline is however measured in the context of a given parametrised ML algorithm by applying the pipeline on a dataset, and measuring the performance of the ML algorithm (e.g., predictive accuracy) on the transformed output. In this context, by fixing the hyper-parameters of the ML algorithm to the default ones, the performance of the learner is set to measure the effectiveness of the considered data pipeline.

3. Data pre-processing pipeline generation

Following the notation from [25], we also distinguish between a fixed, ordered sequence of kinds of pre-processing transformations, known as *pipeline prototypes*, and a fixed, ordered sequence of operators (i.e., instantiations of transformations) known as (executable) *pipelines*. 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, which are then fed to an optimizer (e.g., we use the SMBO [8] variant), to be instantiated and further optimized. 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., ML framework used or dataset characteristics).

The method consists of two flows running in parallel. The first flow is responsible for the pipeline prototype construction and the second flow allows to generate rules that guide the instantiation of the transformations inside the pipeline prototype. The output from the two flows is fed to the final step where the instantiation and optimization happens. The result is an executable pipeline. Notice that what we propose is a generic method, however for the sake of an example, we use the OpenML repository [26], the Scikit-learn library, and the HyperOpt tool, that internally uses SMBO, to provide a use case.

The proposed method starts with the selection of the ML library and optimization framework to be used. On the one hand, this allows to choose the potential kinds of transformations and their available instantiations, and on the other it allows 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 flow on top continues 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. Afterwards, for the pairs for which an order cannot be clearly devised, an additional empirical study is proposed. This study may rely on a testbed of dataset representatives, and thus it may implicitly correspond to *domain knowledge*. The output of this step is a set of *empirically learned rules* that determines promising precedences of transformations (i.e. an order that would potentially positively impact the final result of the analysis). However, even after this phase, for some pairs of transformations a precedence order may not be found. These are pairs for whom 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 *empirically learned*), transformations are composed into a set of valid and potentially effective pipeline prototypes.

Once the prototype is constructed, the flow running in parallel is proposed to help with its instantiation. It consists of a meta-learning step, where a set of ML experiments (e.g., pre-processing and classification algorithm runs) are used as training data, to predict the initial operator for the transformations inside the pipeline prototype. These rules extract knowledge from past experiments and are complementary to the rules obtained in the first flow. They would be used, for example, to ease the cold start problem in the prototype instantiation phase.

3.1. Transformations and Operators

The first task in the process consists of selecting the kinds of transformations and their available operators.

When combining two different kinds of transformations, it is important to check if, (i) the input and output types of transformations are compatible, (ii) the combination makes sense, and (iii) 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).

Transf. Kind	Input	Output	Operator	Parameters
Encoding (E)	CA	CO	Ordinal	–
			One Hot	–
Normalization (N)	CO	CO	Standard Scaler	with_mean:[True,False] with_std:[True,False]
			Power Transform	–
			MinMax Scaler	–
			Robust Scaler	quantile_range:[(25,75),(10,90),(5,95)] with_centering:[True,False] with_scaling:[True,False]
Discretization (D)	CO	CA	KBins	n_bins:[3,5,7] encode:[‘onehot’,‘onehot-dense’,‘ord.’] strategy:[‘uniform’,‘quant.’,‘kmeans’]
			Binarization	threshold: [0, 0.5, 2, 5]
Imputation (I)	CA/CO	CA/CO	Univariate	strategy:[‘most_freq.’,’constant’]
			Multivariate	initial_strategy:[‘most_freq’,‘const.’] order:[‘asc’,‘dsc’,‘rom’,‘arab’,‘rand’]
Rebalancing (R)*	CA/CO	CA/CO	Near Miss	n_neighbors:[1,2,3]
			SMOTE	k_neighbors:[5,6,7]
Feat. Eng. (F)	CA/CO	CA/CO	PCA	n_components:[1,2,3,4]
			Select K Best	k:[1,2,3,4]
			PCA + Select K Best	n_components:[1,2,3,4] k:[1,2,3,4]

CA - Categorical, CO - Continuous.
*All transformations except Rebalancing are taken from scikit-learn.

Table 1: List of transformations applicable to Categorical or Continuous data types.

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).

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*.

EXAMPLE 1. The kinds of transformations selected for the sake of our use case 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

	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	

(a) Compatible 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	

(b) Meaningful 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	

(c) Promising precedence.

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.

Table 2: Precedence order between pairs of transformations, represented independently for each phase.

our experiments, we selected the operators and parameters from those available in the Scikit-learn[3] library, and they are listed in Table 1. *Input* denotes the compatible feature type for a given kind of transformation and can be Continuous (CO) — when it represents measurements on some continuous scale, or Categorical (CA) — when it represents information about some categorical or discrete characteristics. Similarly, *Output* denotes the type of the features after a kind of transformation is applied. Finally, *Operator* denotes the physical instantiation for the kind of transformation, and it can be parametrised using its *Parameters*.

3.2. Framework-related rules

Once the implementation framework is selected, one needs to study it and see if there exist constraints that limit the interaction between transformations. For instance, applying a transformation may actually invalidate the application of another transformation, because the compatibility of transformations is dependent on the selected ML framework.

EXAMPLE 2. 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 2a. 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, whereas `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[4], 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 is imposed when considering Rebalancing and Feature Engineering, since these transformations do not accept inputs containing strings (i.e., representing

[3] `https://scikit-learn.org`

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

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 2a. 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 proposed to derive 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, domain-independent knowledge (i.e., literature) about transformations and their applicability. To this end, some of the constraints imposed by the framework may be contradicted here, but this is resolved in the last step of the proposed method, when we take the union of the rules and hence construct the final pipeline prototypes (see Section 3.5). Briefly, in a combination where Imputation is involved, it is advised to apply Imputation first. Next, an Encoding transformation makes sense to be combined in any order with the rest of transformations, except Imputation. 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, a 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.

EXAMPLE 3. For our use case, Table 2b shows the heuristic rules obtained considering domain-independent knowledge about transformations [27]. In comparison with the results from Table 2a, 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. This time even when combining it with Normalization — note the difference with Table 2a. The constraints of Encoding are however not present in Table 2a, hence not considering the framework, 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.

3.4. Empirically learned rules

The two previously proposed steps (i.e., *framework-related* and *heuristic rules*), do not guarantee that for each pair of transformations we will obtain a precedence order. Therefore, for the cases where they are not sufficient to determine the precedence, a third viewpoint can be considered. That of learning a promising order by empirically studying the impact of the combinations on the final result of the

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
$acc_{baseline} = Acc(d, a)$;                     # get baseline performance of algorithm on d
$[pipeline_{T_1 \rightarrow T_2}, acc_{T_1 \rightarrow T_2}] = SMBO(T_1 \rightarrow T_2, d, a)$
                                                  # get pipeline and accuracy for $T_1 \rightarrow T_2$
$[pipeline_{T_2 \rightarrow T_1}, acc_{T_2 \rightarrow T_1}] = SMBO(T_2 \rightarrow T_1, d, a)$
                                                  # get pipeline and accuracy for $T_2 \rightarrow T_1$
if $IsValid(acc_{T_1 \rightarrow T_2}, acc_{T_2 \rightarrow T_1}, acc_{baseline})$ then   # see Table 3 for the rules applied
    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 3
else
    return $\varnothing$
end if
```

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

$\varnothing$ - 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.

Table 3: Validation rules.

analysis, using different classification problems in the training. For every selected pair of transformations, for a given classification algorithm, we propose to check which order of the pair improves most the performance (e.g., predictive accuracy) of the algorithm over a set of datasets (preferably from different domains). Like this, for each dataset we can get a precedence order that gives better results (i.e., promising precedence) in terms of predictive accuracy (other metrics can be used as well).

3.4.1. Algorithm

To find a promising precedence order between a given pair of transformations, we propose Algorithm 1. To compute the impact of transformations, we first get the accuracy of the ML algorithm over the original non-transformed dataset (see line 1). Afterwards, for each precedence order between the pairs of transformations, we find both their optimized executable pipelines (i.e., using SMBO), and the accuracies of the ML algorithm (with default parametrisation) over the datasets transformed using the respective pipelines (see lines 2-3). Based on the comparison between the respective optimized pipelines, we get the winner in line 5. However, beforehand, in line 4, we perform a validity check. This is because when optimizing a pre-processing pipeline, SMBO may not instantiate a transformation with an operator at all (i.e., represented with a $\varnothing$ symbol). Hence, given a pair of transformations, where one or both of them may not be instantiated, SMBO may generate 16 possible scenarios. They are listed in Table 3, and make up the validation rules for Algorithm 1 (see line 4).

Briefly, if among the optimized pairs of transformations (same transformations but in reverse order) obtained from SMBO, one or both of them contain a $\varnothing$ operator, their results are considered valid, only if they have equal scores (i.e., a draw). This is because, if one has a higher score, it means that during the optimization phase it was more advantageous than the other, since it could find a configuration that should have been found by both of the pairs, given enough budget. In our SMBO runs, such invalid results account for less than 10% and in those cases datasets are discarded from the study (see line 7).

In particular, in Table 3, the first two columns denote the pipeline instantiations for the respective pairs of transformations (i.e., $T_1 \rightarrow T_2$ and $T_2 \rightarrow T_1$). Next, *Valid result* denotes the expected result when comparing the results of the pipelines in the same row. For instance, in the first row, if both transformations 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, and the *Winner prototype* is the prototype that is in accordance with the expected result, which in this case is the Baseline (i.e., prototype consisting of only the ML algorithm, where no transformations are applied).

For the sake of another example, let us check row 2 in Table 3. Running SMBO, the best result for the first pair is the pipeline $\varnothing \rightarrow \varnothing$, and for the second pair, the pipeline $T_2 \rightarrow \varnothing$. In this case, the comparison between the results of these pipelines should be equal (i.e., draw), and the score should be that of the baseline. Otherwise, if say, the score of the second pipeline was higher, it would mean that for the first pair, SMBO was not given enough time to find the pipeline with higher score (i.e., $T_2 \rightarrow \varnothing$). The same logic applies also for the other rows where a $\varnothing$ operator is involved.

EXAMPLE 4. For the sake of this work, we considered three classification algorithms (i.e., *NB*, *RF*, *KNN*) and 80 datasets from the OpenML repository. The datasets, were compiled from three OpenML benchmarks, namely, the OpenMLCC18 benchmark[5], the AutoML benchmark[6], and the Classification algorithms benchmark[7]. For the final set, we filtered out datasets with more than 10% of missing values — not to include bias due to the heavy pre-processing we need to perform on top of them, and we filtered out the datasets with more than 5 million instances — because of the computation time required to process them. As a result we obtained 60 datasets from the first benchmark, 17 from the second, and 3 more from the third to reach a total of 80 datasets.

[5] https://www.openml.org/s/99/data

[6] https://www.openml.org/s/271/data

[7] https://www.openml.org/s/1/data

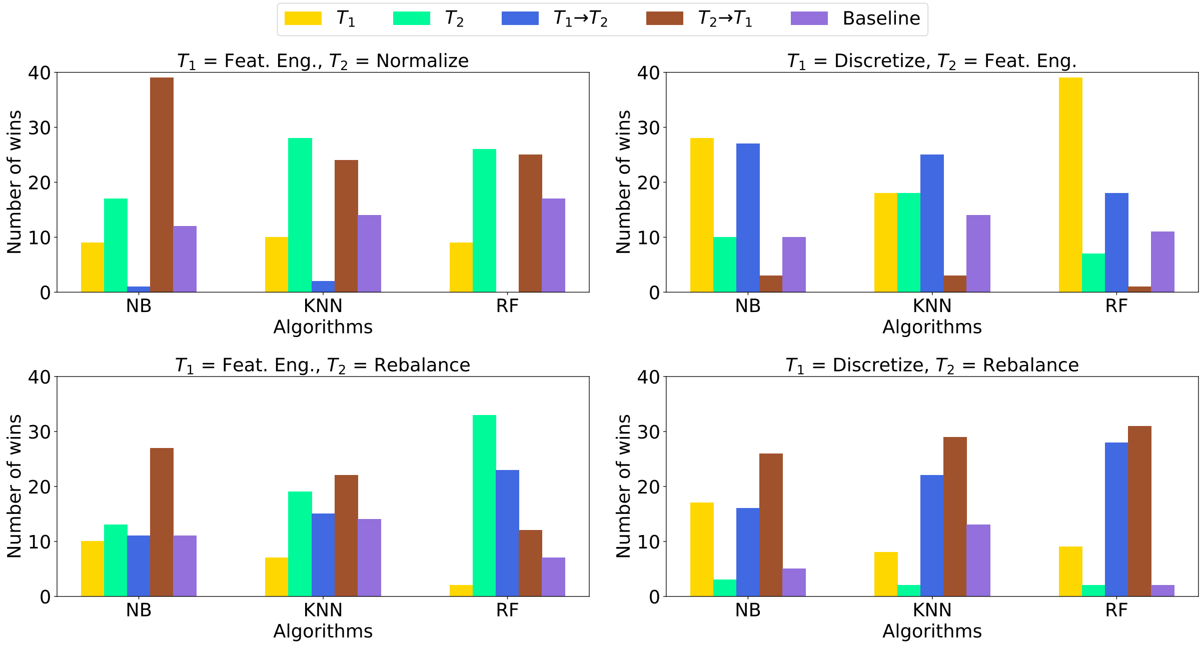

Figure 4: Number of datasets for which a given pipeline prototype is declared the winner.

T_1	T_2	$T_1 \to T_2$	$T_2 \to T_1$	alpha	p-value
F	N	3	88	0.05	**0**
D	F	70	7	0.05	**0**
F	R	49	61	0.05	8.53e-01
D	R	66	86	0.05	9.38e-01

N - Normalization; D - Discretization; R - Rebalancing; F - Feature Engineering.

Table 4: Binomial test for determining the order between pairs of transformations.

Given the proposed algorithm (i.e. Algorithm 1), 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 2a and 2b, 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 2a and 2b) 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 2a). To this end, the pairs of transformations we consider for the third precedence graph include only $\{F, N\}$, $\{F, D\}$, $\{F, R\}$, and $\{R, D\}$.

Applying Algorithm 1, we obtain a promising order for each pair of transformations considered. Since SMBO is a randomized algorithm we experimented with (i) running it several times splitting the budget, and (ii) running it only once with the entire budget. For the experiments considered, no significant differences where observed, therefore we opted for running it once with the entire 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 prototype (see Table 3, 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 prototype winning in more datasets for *KNN* and *NB* is $N \to F$. This means that in general, better results are obtained if Normalization is applied before Feature Engineering.

Next, only N appears as first for *RF* and second best for *KNN* and *NB*, which means that for many datasets, considering different algorithms, it results better to apply only Normalization without combining it with Feature Engineering. The third position is for $\varnothing \to \varnothing$, which means that for some datasets it is better not to apply any of the transformations (in any combination). The remaining prototypes winning in some datasets are F (only Feature Engineering), and $F \to N$ (Feature Engineering preceding Normalization). Finally, for three datasets, that are omitted from the figure, there were no winning pipelines (i.e., pipelines resulted in a draw).

Since our goal is to find the best order for a pair of transformations, we focus on the performances of the pipelines where both of the transformations are instantiated (i.e., $T_1 \to T_2$ versus $T_2 \to T_1$). To do this, we check whether the difference between the number of datasets where they each appear to win are statistically significant by running a binomial test assuming a theoretical probability of 0.5. The results are shown in Table 4. In summary, the results from Table 4 indicate that, with 95% confidence we can assume that for the pair $\{F, N\}$, $N \to F$ performs better than $F \to N$, hence Normalization should precede Feature Engineering. On the other hand, for $\{D, F\}$, $D \to F$ performs better than $F \to D$, hence Discretization should precede Feature Engineering. Finally, for the remaining transformations, $\{F, R\}$ and $\{R, D\}$, a precedence order can not be pre-assumed since the results obtained are not significant. Using these results, we create the *Promising precedence* adjacency matrix shown in Table 2c, 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 $\{R, D\}$.

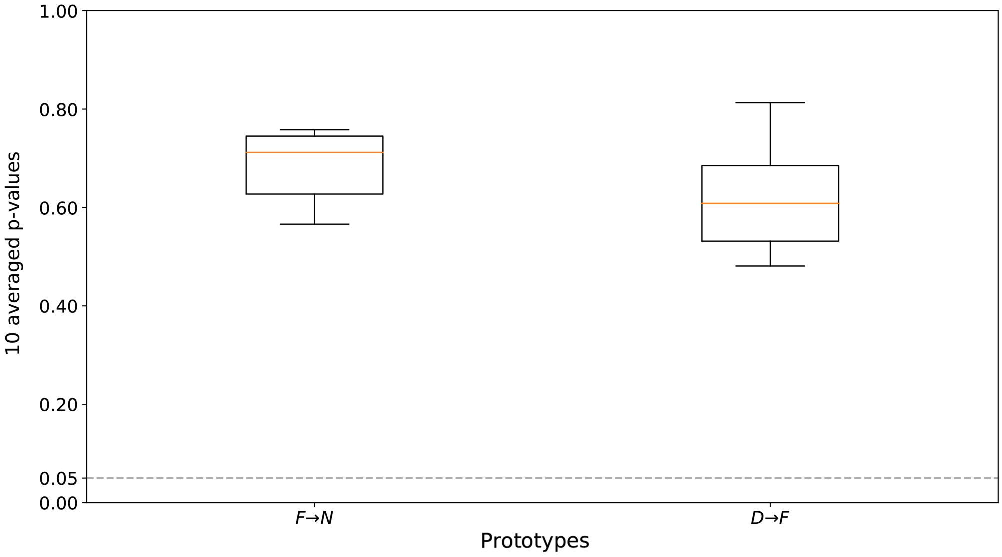

Figure 5: The distribution of the p-values obtained after repeating the chi-square test for 10 times, for the 10 times 4-fold cross-validation.

3.4.2. Cross-validation

After running Algorithm 1 to empirically find a winner between two pairs of transformations, we may obtain a different distribution of the number of wins for the pairs, depending on the datasets considered. To show that the results obtained with the initial set of datasets are generalizable, we propose to perform an additional cross-validated experiment, where the set of datasets considered can be randomly split into many folds. Then, for each fold, the results can be compared to the rest, with the aim of checking whether the distributions are similar. This check can be done via a significance test (e.g., chi-square). To this end, if the distributions between the folds are similar, it means that the obtained results are independent of the datasets considered, since no matter the combination of the datasets, the results are the same and thus generalizable.

EXAMPLE 5. In our use case, to show that the results do not depend on the datasests selected, we re-run the experiments (i.e., 10-times each), but this time splitting the datasets into 4-folds. The goal was to check if the results of the precedence orders from the different folds (i.e., for each experiment considering a randomly different set of datasets) are similar between them (i.e., follow the same distributions). To confirm this hypothesis, we perform a chi-square test between the results (precedence orders) obtained in a single fold in comparison to the three remaining folds, hence comparing 25% of the datasets to the rest. In particular, to confirm the hypothesis, we need to find results that accept the null hypothesis of the chi-square test which states that "there is no significant difference between the distributions". To do that, sticking to the 95% confidence interval, we need to look for p-values greater than 0.05. That is, the higher the p-values, the more we accept the null hypothesis, the more similar the distributions. Looking at the p-values we found out that they were all much higher than 0.05. Specifically, the scores of the chi-square tests of the folds (one fold compared to the rest) are averaged and, after having repeated this procedure 10 times, instead of using a table we depict the 10 averaged p-values using box-plots in Figure 5. We conclude that, for both of the rules (i.e., $F \to N$ and $F \to D$), the significance test indicates a compliance between the new results (Figure 5) and those illustrated above (Table 4).

3.5. Effective pipeline prototypes

In this task we foresee the composition of the previously defined rules (i.e., for the pairs of transformations), to generate the final set of rules that would allow to compose longer chains — consisting of

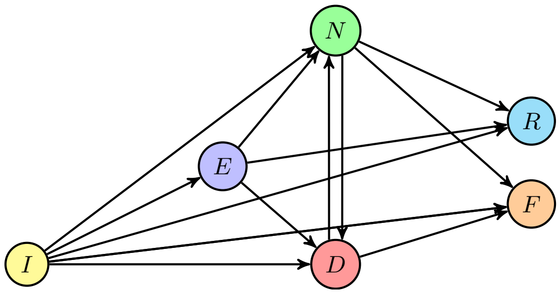

Figure 6: Precedence graph generated from Table 5. E - Encoding; N - Normalization; D - Discretization; I - Imputation; R - Rebalancing; F - Feature Engineering.

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

Table 6: Effective pipeline prototypes generated from Figure 6. E - Encoding; N - Normalization; D - Discretization; I - Imputation; R - Rebalancing; F - Feature Engineering.

more than two transformations. This is when we resolve the inconsistencies and also define precedences for the pairs of transformations that may not have any precedence defined already — in that case, we basically take into account all the permutations. This step allows to finally generate the possible effective pipeline prototypes.

EXAMPLE 6. 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 2 (a,b,c), and create a new final adjacency matrix, shown in Table 5. This is the matrix that will allow us to generate the final effective pipeline prototypes.

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 6, we visualize Table 5 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 6. This set consisting of *compatible*, *meaningful* and *promising* pairs of transformations is the set of recommended *effective pipeline prototypes*.

3.6. Meta-learning rules

Once the pipeline prototype is constructed, that is, the order between the kinds of transformations is defined, what follows is the instantiation of transformations with the physical operators. For that, one can rely completely on the optimization algorithm, and let the algorithm choose the right operators.

	$\boldsymbol{E}$	$\boldsymbol{N}$	$\boldsymbol{D}$	$\boldsymbol{I}$	$\boldsymbol{R}$	$\boldsymbol{F}$
$\boldsymbol{E}$		1	1	0	1	1
$\boldsymbol{N}$	0		X	0	1	1
$\boldsymbol{D}$	0	X		0	0	1
$\boldsymbol{I}$	1	1	1		1	1
$\boldsymbol{R}$	0	0	0	0		0
$\boldsymbol{F}$	0	0	0	0	0	

Table 5: Union of rules from Table 2. $\boldsymbol{E}$ - Encoding; $\boldsymbol{N}$ - Normalization; $\boldsymbol{D}$ - Discretization; $\boldsymbol{I}$ - Imputation; $\boldsymbol{R}$ - Rebalancing; $\boldsymbol{F}$ - Feature Engineering `1` - an edge exists, `0` - edge does not exist, `X` - the combination is meaningless.

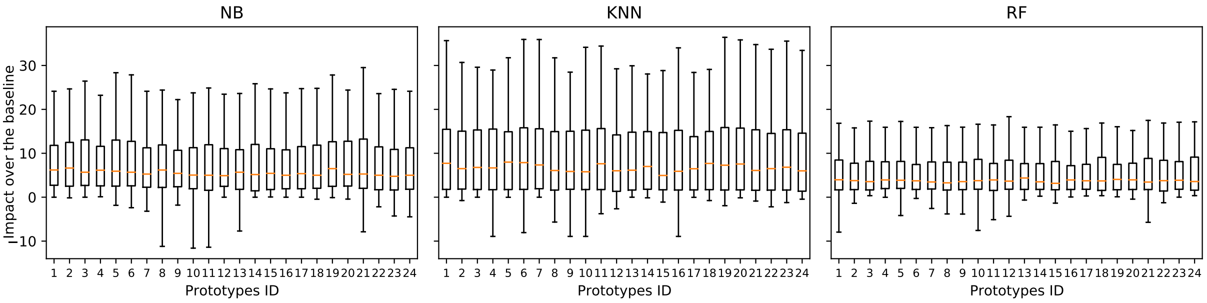

Figure 7: The impact of the different pipeline prototypes over the baseline (i.e., when no transformation is applied).

However, given the way optimization algorithms work (e.g., SMBO) — successively finding better and better instantiations, there is a cold-start problem, where in the beginning, the algorithm does not have enough information in order to come up with the most promising initial instantiations, and a wrong choice may affect the optimization process.

3.6.1. Exploratory analysis

Given the availability of the experimental SMBO executions (executed in an exhaustive manner, considering all the pipeline prototypes), one can perform an exploratory analysis with the aim of removing useless prototypes, pipelines or operators. Hence, further tweaking the search space. In particular, starting from the highest level, that of prototypes, then going to the physical pipelines, and finally to the actual operators inside the pipeline, one can analyze if:

- there exist some combination of transformations in the form of prototypes (see Table 7 for the exhaustive list of prototypes), that are generally useless (i.e., in terms of their impact to the final accuracy), and thus can be discarded a priori in order to reduce the search space,
- there are some physical pipelines that are consistently chosen more often than others by the optimization algorithm, meaning that they are more useful than others,
- within the physical pipelines, some transformations are chosen more often than others, meaning that they provide more positive impact.

EXAMPLE 7. We performed the above-mentioned analysis to our use case, but it did not lead to any conclusive or significant results. In particular, as shown in Figure 7, we could not find any useless prototypes — not positively impacting the final accuracy, that could be discarded a priori from the potential list of prototypes. Actually, as we will show in Section 4.1, all of them lead to the best in one case or another, which does not mean the epsilon improvement some provide is worth the search cost you incur in considering them (but this more in-depth analysis is done later). Next, as shown in Figure 8, there were no physical pipelines shown to be more useful — hence more often selected, than others. Even if $N \rightarrow R$ is clearly above, it barely reaches 30% in KNN. Finally, observing Figure 9, it is clear that some kinds of transformations are chosen more often, but looking closely (i.e., the shaded bars), it is not clear which operator brings more benefit. For instance, Normalization is present in 90% of the pipelines, but it is not easy to distinguish which kind of Normalization (i.e., actual operator) is more beneficial. For this, we need more complex rules or guidelines that may help in finding the right operator to use.

3.6.2. Meta-learning

To mitigate the red cold-start problem, we propose to perform meta-learning (shown in Figure 3), where we intend to use the knowledge extracted from historical data in order to devise rules that may help the optimization algorithm in its initial phase. Meta-learning is the process of 'learning on top of

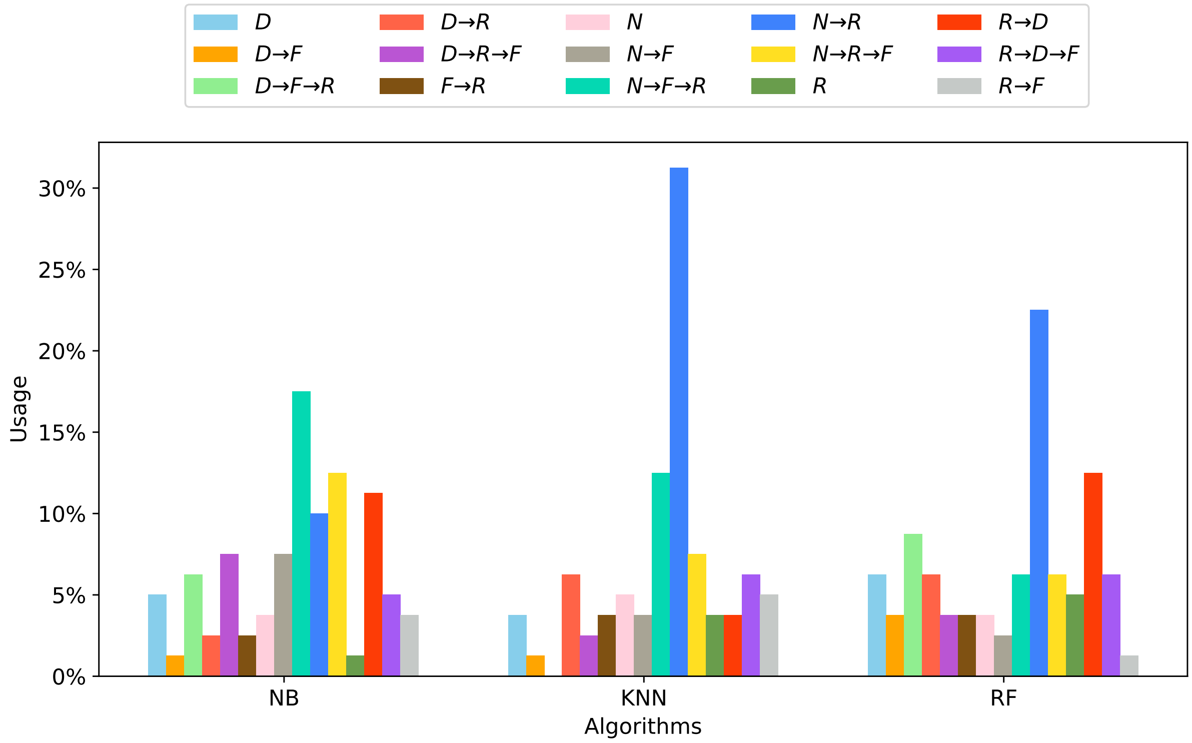

Figure 8: Percentage of use of the different physical pipelines.

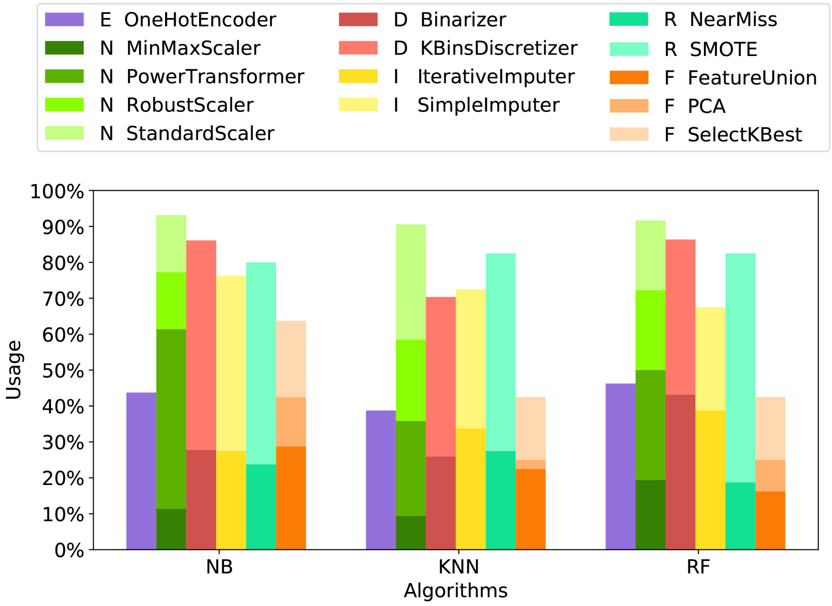

Figure 9: Percentage of use of a transformation in a physical pipeline.

learning', or learning a model using historical data from ML experiments. Traditionally, it has been used for predicting the performance (e.g., predictive accuracy) of an algorithm on a given dataset. That is, given some historical runs of the performance of classification algorithms over various datasets (i.e., meta-database: consisting of datasets characteristics as predictive variables and the performance of the classification algorithm as the response variable), one can learn a model (i.e., meta-model), that is able to predict the performance of a given classification algorithm on a new dataset [28]. Lately, this technique has been extended in order to predict the impact of transformations over the performance of classification algorithms and thus rank transformations based on their impact [29, 30, 31]. The same idea can be applied for learning the best operator for a given transformation. That is, through meta-learning one can learn the intrinsic relationship between dataset characteristics and the operator performance, and thus come up with rules that are not obvious and are effective at the time of instantiating a transformation. The main idea is to build a model, that is able to predict the operator for a certain kind of transformation, given the meta-features extracted from the dataset considered for the optimization. This translates to answering the following question: "given that we know the dataset characteristics and having selected a certain kind of transformation (e.g., missing value imputation), what is the optimal physical algorithm (see Table 1) we need to select, to obtain the highest improvement possible in terms of classification accuracy (i.e., when the classification algorithm is applied over the transformed dataset)?". In particular, the model can generate a set of complementary rules that help in the optimization, providing a good starting instantiation for some of the transformations in the prototype.

To train the model we need a meta-dataset that can be (i) generated through optimization algorithms (e.g., SMBO executions), (ii) generated manually through simple evaluations of classification algorithms over transformed datasets, or (iii) assumed already given (e.g., OpenML).

Given a meta-dataset, we propose to learn to predict the best instantiation (operator) for a given transformation, where among the classes we can include the class `None` too. This means that one of the possible predictions is to not instantiate a transformation at all, hence remove it from the pipeline.

EXAMPLE 8. Our training dataset (sometimes referred to as 'meta-database' or 'meta-dataset') for the meta-learning is compiled through SMBO runs on the OpenML datasets (see Section 3.4.1). That is, we first extract the dataset characteristics/profiles (i.e., number of features, number of instances, number of missing values, etc), and then by applying SMBO optimization, on classification algorithms and pre-processing pipelines (as explained in Section 3.5), for each dataset, we retrieve the evaluations (i.e., predictive accuracy) of the algorithms over the optimized pipelines. This gives us the presumably optimal physical pipelines and their impact on the accuracy of the learning algorithms for each dataset at hand. Given such information, our aim is to now save time and improve the instantiation of the operators for each transformation considered in the prototype.

We trained several different Conditional Inference Trees [32] because they produce models that can be easily read and interpreted. Specifically, the independence of each variable (meta-features in our case) with the class (operator of a specific transformation) is tested through a statistical test. The split is made on the variable with the lowest p-value. We report the p-value too, so that it can be seen how strong the association is (i.e., why that variable was chosen). We stick with the p-value threshold of 0.05, and devise a rule from any branch of the tree that is within the threshold. In the following, we describe the rules obtained within the selected significance threshold.

Rules for Feature Engineering. The available operators in Scikit-learn for Feature Engineering (see Table 1) are: `PCA` (Principal Component Analysis), `Feature Selection` (Select K Best), `Both` (PCA + Select K Best), and `None`. The tree generated for the Feature Engineering transformation is shown in Figure 10. The leaves show the selected operator frequency. For the sake of simplicity, we do not consider the union of PCA and Select K Best as an operator per se, instead we distribute that contribution to the two operators that compose it. Observe that there is a strong correlation between the Feature Engineering operator and the entropy of the class attribute. Indeed, such a meta-feature achieved a p-value smaller than 0.001. We can clearly read that if the Class Entropy is low, then `Feature Selection` is way more chosen than the other options (see Node 2). Recall that the entropy of an attribute is a measure of how much disorder there is among its instances. The less is that value, the easier is the classification problem. As a consequence, it is reasonable to think that the easier the classification problem is, the more likely is the fact that the class can be described by a low number of features. Hence, the `Feature Selection` technique can be successfully applied. Conversely, Node 5 shows that, when the Class Entropy is high, it is better to not apply any Feature Engineering operator. As a matter of fact, a high value of Class Entropy involves a high number of classes and/or few instances per class, hence a really difficult problem. In such cases, reducing the dimensionality of the dataset does not lead to any improvement. Finally, when the Class Entropy is in between, there is no clear winner, and thus other non-obvious factors may affect the choice of the operator.

Rules for Rebalancing. As for Rebalancing, the operators considered from the imblearn[8] library are: `Near Miss`, `SMOTE`, and `None`. The first is an undersampling algorithm which randomly eliminates the samples from the larger class. Instead, the second is an oversampling technique that creates samples of the minority class, as a linear combination of them. As shown in Figure 11, the meta-feature Majority Class Percentage has a p-value of 0.014. This can be read as, in case of an unbalanced class problem (i.e., Node 3: Majority Class Percentage greater than 56), an oversampling of the minority class(es) is preferred to a downsampling of the majority one(s). However, when the Majority Class Percentage is smaller than 56%, the situation is not that clear, and there is no technique that is applied significantly more often than the rest; they are close to each other. Therefore, it is difficult to understand which problems (which dataset characteristics do they have) belong to Node 2. In summary, when the majority class has no more than 56% , it implies that it is an unbalanced class, and as mentioned above, SMBO tends to choose the same operator. However, when the majority class has less than 56%, it may imply that: (i) there are just two classes and the problem counts as a balanced problem, so no operator needs to be applied, or (ii) it is a multi-class problem, and thus there is no clear winner in terms of operators.

3.7. Prototype instantiation

The prototypes from the top flow and the meta-lerning rules from the bottom flow (if the optimization framework permits), are finally fed to the final step which deals with the instantiation and optimization of the prototypes. In this task we run an optimization algorithm that is executed until an optimal pipeline is found.

EXAMPLE 9. In our final execution, we run SMBO to find a suitable instantiation for the suggested prototypes. The simple but not obvious meta-learning rules, even though not included in our final execution, because of the implementation considered (i.e., HyperOpt), can potentially be used to ease the cold-start problem.

4. Evaluation

The aim of our experimental study is three-fold:

1. Check whether there exists a universal pipeline prototype that works best for any classification problem considered (i.e., dataset and ML algorithm) (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 repository (see Section 3.4.1). Finally, the classification algorithms considered are *NB*, *KNN*, and *RF*. All the experiments for a single algorithm, on average took approximately two weeks[9].

[8] `https://pypi.org/project/imbalanced-learn`

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

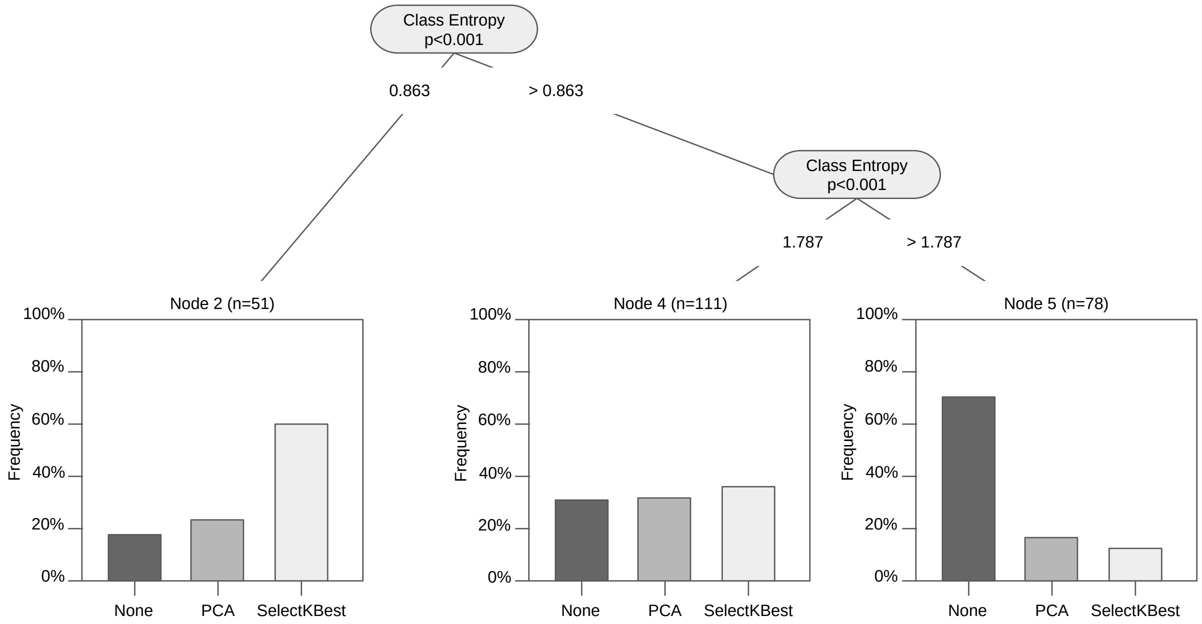

Figure 10: Conditional Inference Tree built for the *Features Engineering* transformation.

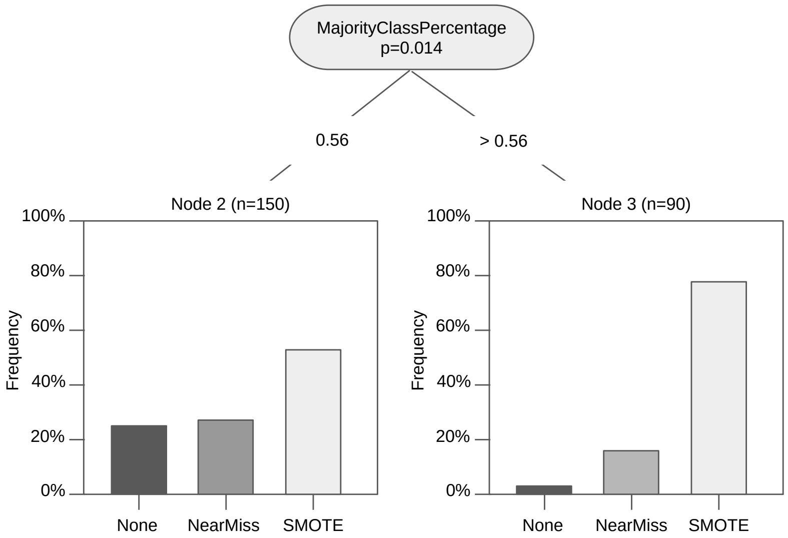

Figure 11: Conditional Inference Tree built for the *Rebalancing* transformation.

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 7, we list the exhaustive set of pipeline prototypes generated considering the compatible precedence graph in Table 2a (i.e., 24 compatible permutations). In a real scenario, this number would be 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.

The results obtained are shown in Figure 12. The enumerated prototypes are listed in the ordinate

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	$I \rightarrow E \rightarrow R \rightarrow F \rightarrow D \rightarrow N$

Table 7: Exhaustive set of pipeline prototypes generated using the compatible precedence graph of Table 2a. E - Encoding; N - Normalization; D - Discretization; I - Imputation; R - Rebalancing; F - Feature Engineering.

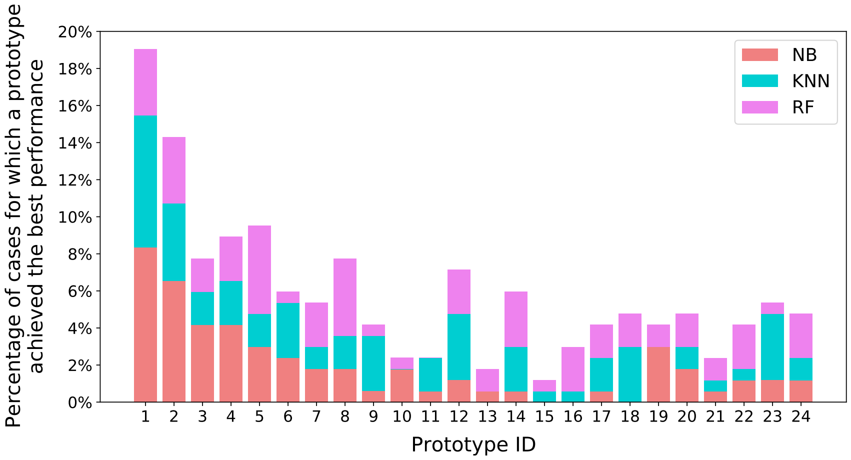

Figure 12: Comparison of the goodness of the exhaustive set of prototypes.

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 19% 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 better solutions.

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 7) to the optimization algorithm in order to get the best solutions out of them. 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 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:

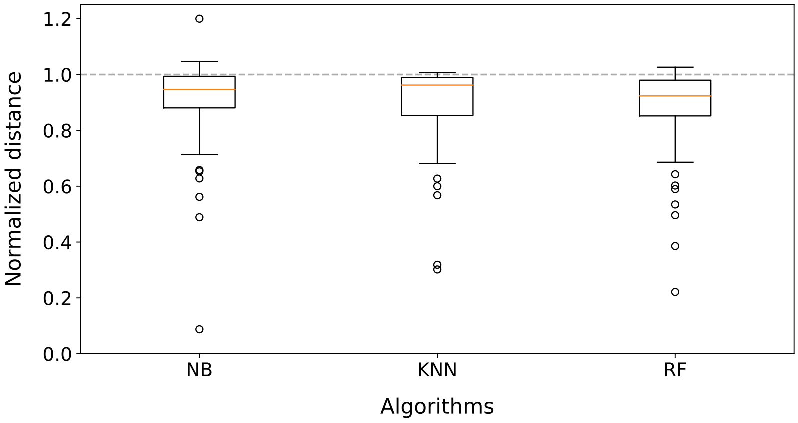

Figure 13: Normalized distances between the scores obtained by optimizing our effective prototypes and the ones obtained optimizing the exhaustive set.

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

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 13. 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 exhaustive 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[10].

To this end, for the latter we now dedicate the total optimization budget (i.e., 400 seconds), and for the former, inspired by [25], we split the budget 50-50 between the pre-processing pipeline optimization

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

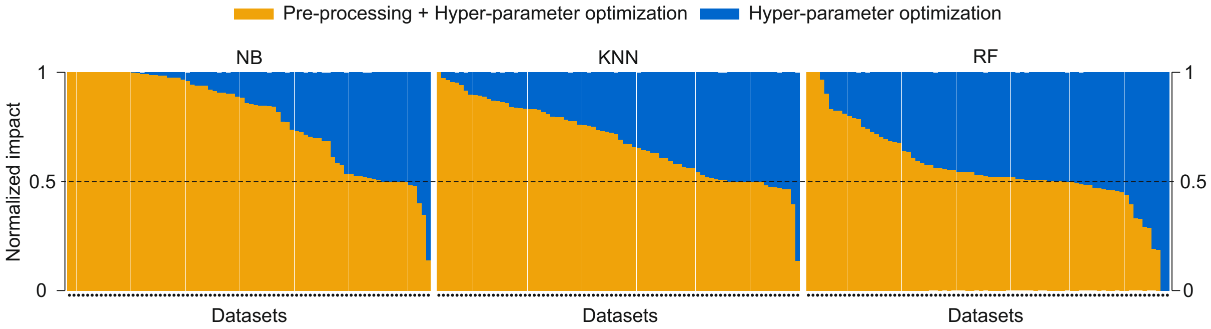

Figure 14: The impact of dedicating a portion of the optimization budget to pre-processing compared to using the whole optimization budget for the hyper-parameter optimization.

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.

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

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

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). $Acc(d_{\textit{effective}}, a^*)$ is the accuracy of the optimized algorithm a^* over the dataset $d_{\textit{effective}}$ transformed using the optimized instantiation of the effective set of prototypes obtained using our method. 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:

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

We perform the same for the hyper-parameter impact and plot the results obtained for all the algorithms and datasets in Figure 14, 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. Related work

A lot of ongoing research aims at addressing the problem of providing user assistance for the data analytics process. Specifically, they can be classified into three main categories [33]: distributed, cloud-based, and centralized. The first two try to address the problem of Big Data. Thus, clusters of several machines are employed to distribute the workload. On the contrary, this is not a fundamental requirement for centralized solutions. Indeed, the overhead of using a cluster is not worth for relatively small datasets. Since our work belongs to the category of centralized solutions, in the following, we provide examples of them.

As already mentioned before, the data analytics process consists of different steps. 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 [34, 35, 36, 37]. 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.

5.1. DPSO

In DPD [25], 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 have already seen (Section 4.1), a single fixed prototype cannot perform best for every dataset. Therefore, we build on top of [25], and instead of relying on a fixed prototype, we define a method to generate the right pipeline prototypes to be optimized.

In PRESISTANT [30, 38, 31], 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 [39], 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 [40], 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 [25], 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 [41], 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 [25] and if some further adjustment from the user is required.

5.2. 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 [42, 43, 44]. Since Bayesian optimization is randomized, meta-learning has been used to find a good seed for the search [45]. Most of these works however, only minimally consider the data pre-processing step. Auto-WEKA [43], based on the Java machine learning library Weka, is the pioneer of the field. The authors formalized the problem of algorithm selection and their associated hyper-parameter optimization, and solved it in a combined search space. Sequential Model-based Algorithm Configuration (SMAC) is used to explore the large search space.

Autostacker [46] combines a hierarchical stacking architecture and an evolutionary algorithm (EA). Stacking is an ensemble method that involves the concatenation of several classifiers, so that the later layers can learn the mistakes that classifiers in the previous layers make. Even if it brings some benefits, this approach affects the search space: way larger than that of a single classifier. In a nutshell, such concatenations are randomly generated and then optimized. The one that achieves the higher predictive accuracy is chosen. Rather than Bayesian Optimization, to find suitable hyper-parameters, the authors utilize a basic Evolutionary Algorithm.

OBoe [47] exploits collaborative filtering for AutoML, choosing models that have performed well on similar datasets. It collects a large number of datasets and applies different ML algorithms (with different hyper-parameters configurations). In this way, a matrix of cross-validated errors is built. Common approaches typically compute dataset meta-features and use them to predict the error of a particular machine learning model, but OBoe works exactly the other way around. PCA is applied on such a matrix in order to find latent meta-features. Given a new dataset, some basic algorithms are applied to infer a feature vector (i.e., the value of the latent meta-features). Finally, the feature vector is leveraged to estimate the cross-validated error of more complex algorithms.

5.3. DPSO + CASH

Auto-sklearn [42] is based on the popular Python library scikit-learn. The authors, inspired by Auto-Weka, address the problem with the Sequential Model-based Algorithm Configuration (SMAC). Furthermore, they improve the approach by adding a meta-learning phase at the beginning (to warm-start the Bayesian Optimization) and an ensemble technique at the end (to suggests multi-classifiers). Such a system considers 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 in a way is complementary to this, since instead of a priori fixing the prototype, we can construct a potentially optimal one (or a set), and then provide it to the tool for it to be instantiated and further optimized. TPOT [44] 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).

ML-Plan [3] uses hierarchical planning, a particular form of AI planning, to propose a solution to both the pre-processing and the modeling phases. As in context-free grammars, there are complex tasks (non-terminal symbols) that are derived as long as primitive tasks (terminal symbols) are not obtained. Typically, standard graph search algorithms (e.g., depth-first search, best-first search, etc.) are employed to solve such problems. ML-Plan successively creates solutions in a global search instead of changing given solutions in a local search. However, due to the problem constraints, they adopt a randomized best-first search, randomly choosing the solution path.

AutoBazaar [48] is a Python open-source tool. Like in ML-Plan [3], both pre-processing and modeling phases are covered. Here the last step of a prototype is the machine learning algorithm. The approach involves two different steps. Firstly, a *catalog* proposes a collection of prototypes (with an ML algorithm as last step) based on the task and the dataset itself. Secondly, the optimization process starts tuning the prototypes until either the time budget is expired or the prototypes are all optimized. In particular, a *selector* and a *tuner* work in synergy. The former decides which prototype should be optimized next. Such a task is treated as a multi-armed bandit problem. As to the tuner, Bayesian Optimization is chosen. At the end, the prototype that achieved the higher predictive accuracy is elected. However, AutoBazaar strictly depends on the catalog. Such a component memorizes all the possible primitives and supported tasks. The prototypes are hard-coded for each task. Thus, it is neither flexible nor maintainable. If a task is not implemented, the approach cannot suggest a solution.

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. 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 space and guiding the search, hence reducing their cost.

6. Conclusions and future work

In this work, we first studied the overall impact of transformations when chained together inside pre-processing prototypes and then delved into examining the impact of instantiating transformations via various operators. As a result, we defined a method that allows to generate effective pre-processing pipelines. That is, pipelines that consist of, (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). In addition, via the meta-learning step proposed, we aim to guide the instantiation of transformations in order to facilitate finding better instantiations.

An extensive evaluation on 80 datasets with heterogeneous characteristics, from sample size to feature types, and a set of classification algorithms (i.e., Naive Bayes, Random Forest, K-Nearest Neighbours), 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.

Finally, previous works have shown the effectiveness of meta-learning for solving the cold start problem [45], hence as immediate future work, we intend to extend an optimization framework (i.e., HyperOpt) with a complementary meta-learning module that can ease the cold-start problem, facilitating the search for optimal instantiations.

Acknowledgments. This work was supported by the DOGO4ML project, funded by the Spanish Ministerio de Ciencia e Innovación under project/funding scheme PID2020-117191RB-I00 / AEI / 10.13039/501100011033. 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] M. A. Munson, A study on the importance of and time spent on different modeling steps, SIGKDD Explor. Newsl. 13 (2) (2012) 65–71.

[2] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, F. Hutter, Efficient and robust automated machine learning (2015) 2962–2970.

[3] F. Mohr, M. Wever, E. Hüllermeier, Ml-plan: Automated machine learning via hierarchical planning, Machine Learning 107 (8) (2018) 1495–1515.

[4] L. Muñoz, J.-N. Mazón, J. Trujillo, Automatic generation of etl processes from conceptual models, in: Proceedings of the ACM Twelfth International Workshop on Data Warehousing and OLAP, DOLAP '09, 2009, p. 33–40.

[5] A. A. Vaisman, E. Zimányi, Data Warehouse Systems - Design and Implementation, Data-Centric Systems and Applications, Springer, 2014.

[6] B. Bilalli, A. Abelló, T. Aluja-Banet, R. Wrembel, Towards intelligent data analysis: The metadata challenge, in: M. Ramachandran, G. B. Wills, R. J. Walters, V. M. Muñoz, V. Chang (Eds.), Proceedings of the International Conference on Internet of Things and Big Data, IoTBD 2016, Rome, Italy, April 23-25, 2016, pp. 331–338.

[7] A. Quemy, Data pipeline selection and optimization, in: Proceedings of the 21st International Workshop on Design, Optimization, Languages and Analytical Processing of Big Data, co-located with EDBT/ICDT Joint Conference, DOLAP@EDBT/ICDT 2019, Lisbon, Portugal, March 26, 2019, 2019.

[8] J. Bergstra, D. Yamins, D. D. Cox, Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures, ICML'13, 2013, pp. 115–23.

[9] J. Giovanelli, B. Bilalli, A. Abelló, Effective data pre-processing for automl, in: K. Stefanidis, P. Marcel (Eds.), Proceedings of the 23rd International Workshop on Design, Optimization, Languages and Analytical Processing of Big Data (DOLAP), Vol. 2840 of CEUR Workshop Proceedings, CEUR-WS.org, 2021, pp. 1–10.

[10] L. Kotthoff, C. Thornton, H. Hoos, F. Hutter, K. Leyton-Brown, Auto-weka 2.0: Automatic model selection and hyperparameter optimization in weka, Journal of Machine Learning Research 18 (2017) 1–5.

[11] F. Serban, J. Vanschoren, J. Kietz, A. Bernstein, A survey of intelligent assistants for data analysis, ACM Computing Surveys 45 (3) (2013) 1–35.

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

[13] J. Bergstra, Y. Bengio, Random search for hyper-parameter optimization., Journal of machine learning research 13 (2) (2012).

[14] P. J. Van Laarhoven, E. H. Aarts, Simulated annealing, in: Simulated annealing: Theory and applications, Springer, 1987, pp. 7–15.

[15] O. Kramer, Genetic algorithms, in: Genetic algorithm essentials, Springer, 2017, pp. 11–19.

[16] P. I. Frazier, A tutorial on bayesian optimization (2018). `arXiv:1807.02811`.

[17] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, A. Talwalkar, Hyperband: A novel bandit-based approach to hyperparameter optimization, The Journal of Machine Learning Research 18 (1) (2017) 6765–6816.

[18] M.-A. Zöller, M. F. Huber, Survey on automated machine learning, arXiv preprint arXiv:1904.12054 9 (2019).

[19] Q. Yao, M. Wang, Y. Chen, W. Dai, Y.-F. Li, W.-W. Tu, Q. Yang, Y. Yu, Taking human out of learning applications: A survey on automated machine learning (2018). `arXiv:1810.13306`.

[20] F. Hutter, H. H. Hoos, K. Leyton-Brown, Sequential model-based optimization for general algorithm configuration, in: International conference on learning and intelligent optimization, Springer, 2011, pp. 507–523.

[21] F. Archetti, A. Candelieri, Bayesian Optimization and Data Science, 1st Edition, Springer International Publishing, 2019.

[22] F. Hutter, H. H. Hoos, K. Leyton-Brown, K. P. Murphy, An experimental investigation of model-based parameter optimisation: Spo and beyond, in: Proceedings of the 11th Annual conference on Genetic and evolutionary computation, 2009, pp. 271–278.

[23] J. Snoek, H. Larochelle, R. P. Adams, Practical bayesian optimization of machine learning algorithms, NeurIPS '12, 2012, pp. 2960–2968.

[24] M. Wistuba, N. Schilling, L. Schmidt-Thieme, Scalable gaussian process-based transfer surrogates for hyperparameter optimization, Mach. Learn. 107 (1) (2018) 43–78.

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

[26] J. Vanschoren, J. N. van Rijn, B. Bischl, L. Torgo, Openml: Networked science in machine learning, SIGKDD Explorations 15 (2) (2013) 49–60.

[27] T. Dasu, T. Johnson, Exploratory Data Mining and Data Cleaning, 1st Edition, John Wiley & Sons, Inc., New York, NY, USA, 2003.

[28] P. Brazdil, C. G. Giraud-Carrier, C. Soares, R. Vilalta, Metalearning - Applications to Data Mining, Cognitive Technologies, Springer, 2009.

[29] 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.

[30] B. Bilalli, A. Abelló, T. Aluja-Banet, R. Wrembel, Intelligent assistance for data pre-processing, Comput. Stand. Interfaces 57 (2018) 101–109.

[31] B. Bilalli, A. Abelló, T. Aluja-Banet, R. Wrembel, PRESISTANT: learning based assistant for data pre-processing, Data Knowl. Eng. 123 (2019).

[32] T. Hothorn, K. Hornik, A. Zeileis, Unbiased recursive partitioning: A conditional inference framework, Journal of Computational and Graphical Statistics 15 (3) (2006) 651–674.

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

[34] F. Geerts, G. Mecca, P. Papotti, D. Santoro, The llunatic data-cleaning framework, PVLDB End. 6 (9) (2013) 625–636.

[35] Z. Khayyat, I. F. Ilyas, A. Jindal, S. Madden, M. Ouzzani, P. Papotti, J.-A. Quiané-Ruiz, N. Tang, S. Yin, Bigdansing: A system for big data cleansing, SIGMOD '15, 2015, pp. 1215–1230.

[36] X. Chu, J. Morcos, I. F. Ilyas, M. Ouzzani, P. Papotti, N. Tang, Y. Ye, Katara: A data cleaning system powered by knowledge bases and crowdsourcing, SIGMOD '15, 2015, pp. 1247–1261.

[37] Z. Jin, M. R. Anderson, M. Cafarella, H. V. Jagadish, Foofah: A programming-by-example system for synthesizing data transformation programs, SIGMOD '17, 2017, pp. 1607–1610.

[38] B. Bilalli, A. Abelló, T. Aluja-Banet, R. F. Munir, R. Wrembel, PRESISTANT: data pre-processing assistant, CAiSE Forum '18, 2018, pp. 57–65.

[39] S. Krishnan, J. Wang, E. Wu, M. J. Franklin, K. Goldberg, Activeclean: Interactive data cleaning for statistical modeling, PVLDB 9 (12) (2016) 948–959.

[40] L. Berti-Équille, Learn2clean: Optimizing the sequence of tasks for web data preparation, WWW '19, 2019, pp. 2580–2586.

[41] Z. Shang, E. Zgraggen, B. Buratti, F. Kossmann, P. Eichmann, Y. Chung, C. Binnig, E. Upfal, T. Kraska, Democratizing data science through interactive curation of ML pipelines, SIGMOD '19, 2019, pp. 1171–1188.

[42] M. Feurer, A. Klein, K. Eggensperger, J. T. Springenberg, M. Blum, F. Hutter, Efficient and robust automated machine learning, NeurIPS '15, 2015, pp. 2962–2970.

[43] C. Thornton, F. Hutter, H. H. Hoos, et al., Auto-weka: Combined selection and hyperparameter optimization of classification algorithms, in: KDD, 2013, pp. 847–855.

[44] R. S. Olson, N. Bartley, R. J. Urbanowicz, J. H. Moore, Evaluation of a tree-based pipeline optimization tool for automating data science, GECCO '16, 2016, pp. 485–492.

[45] M. Feurer, J. T. Springenberg, F. Hutter, Initializing bayesian hyperparameter optimization via meta-learning, AAAI '15, 2015, pp. 1128–1135.

[46] B. Chen, H. Wu, W. Mo, I. Chattopadhyay, H. Lipson, Autostacker: A compositional evolutionary learning system, in: Proceedings of the Genetic and Evolutionary Computation Conference, 2018, pp. 402–409.

[47] C. Yang, Y. Akimoto, D. W. Kim, M. Udell, Oboe: Collaborative filtering for automl model selection, in: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, 2019, pp. 1173–1183.

[48] M. J. Smith, C. Sala, J. M. Kanter, K. Veeramachaneni, The machine learning bazaar: Harnessing the ML ecosystem for effective system development, CoRR abs/1905.08942 (2019).