mathematics

MDPI

Article

EvoPreprocess—Data Preprocessing Framework with Nature-Inspired Optimization Algorithms

Sašo Karakatič

Faculty of Electrical Engineering and Computer Science, University of Maribor, Maribor 2000, Slovenia; saso.karakatic@um.si

Received: 29 April 2020; Accepted: 27 May 2020; Published: 2 June 2020

Abstract: The quality of machine learning models can suffer when inappropriate data is used, which is especially prevalent in high-dimensional and imbalanced data sets. Data preparation and preprocessing can mitigate some problems and can thus result in better models. The use of meta-heuristic and nature-inspired methods for data preprocessing has become common, but these approaches are still not readily available to practitioners with a simple and extendable application programming interface (API). In this paper the EvoPreprocess open-source Python framework, that preprocesses data with the use of evolutionary and nature-inspired optimization algorithms, is presented. The main problems addressed by the framework are *data sampling* (simultaneous over- and under-sampling data instances), *feature selection* and *data weighting* for supervised machine learning problems. EvoPreprocess framework provides a simple object-oriented and parallelized API of the preprocessing tasks and can be used with scikit-learn and imbalanced-learn Python machine learning libraries. The framework uses self-adaptive well-known nature-inspired meta-heuristic algorithms and can easily be extended with custom optimization and evaluation strategies. The paper presents the architecture of the framework, its use, experiment results and comparison to other common preprocessing approaches.

Keywords: data sampling; feature selection; instance weighting; nature-inspired algorithms; meta-heuristic algorithms

1. Introduction

Data preprocessing is one of the standard procedures in data mining, which can greatly improve the performance of machine learning models or statistical analysis [1]. Three common data preprocessing tasks that are addressed by the presented EvoPreprocess framework are feature selection, data sampling, and data weighting. This paper presents the EvoPreprocess framework, which addresses the listed preprocessing tasks with the use of supervised machine learning based evaluation. All three tasks deal with inappropriate and high-dimensional data, which can result in either over-fitted and non-generalizable machine learning models [2,3].

Many different techniques have been proposed and applied to these of preprocessing data tasks [1,4]; from various feature selection methods based on statistics (information gain, covariance, Gini index, χ^2 etc.) [5,6]; under-sampling data with neighborhood cleaning [7], prototype selection [8], over-sampling data with SMOTE [9], and other SMOTE data over-sampling variants [10]. These methods are mainly deterministic and have limited variability in the resulting solutions. Due to this, researchers also extensively focused on preprocessing with meta-heuristic optimization methods [11,12].

Meta-heuristic optimization methods provide sufficiently good solutions to NP-hard problems while not guaranteeing that the solutions are globally optimal. Since feature selection, data sampling and data weighting are NP-hard problems [6], meta-heuristics present a valid approach, which is

also supported by a wide body of research presented in the following sections. Applications of nature-inspired algorithms in data preprocessing have already been used. Genetic algorithms were used for feature selection in high-dimensional datasets to select six biomarker genes that linked with colon cancer [13,14]. Evolution strategy was used for data sampling for the in the early software defect detection [15]. Also, particle swarm optimization was used for the data sampling for classification of hyperspectral images [16].

While research on the topic is wide and prolific, the standard libraries, packages and frameworks are sparse. There are some well maintained and well-documented data preprocessing libraries [17–22], but none of them provide the ability to use nature-inspired approaches in Python programming language. The EvoPreprocess framework aims to fill in the gap between the data preprocessing and nature-inspired meta-heuristics and provide easy to use and extend Python API that can be used by practitioners in their data mining pipelines, or by researchers developing novel nature-inspired methods on the problem of data preprocessing.

This paper presents the implementation details and examples of use of the EvoPreprocess framework, which offers API for solving the three mentioned data preprocessing tasks with nature-inspired meta-heuristic optimization methods. While the data preprocessing approaches for classification tasks are already available, there is a lack of resources for data preprocessing on regression tasks. The presented framework works with both tasks of supervised learning (i.e., classification and regression) and therefore fills this gap.

The novelty of the framework is the following:

1. The framework provides simple Python object-oriented and parallel implementation of three common data preprocessing tasks with nature-inspired optimization methods.
2. The framework is compatible with the well-established Python machine learning and data analysis libraries, scikit-learn, imbalanced-learn, pandas and NumPy. It is also compatible with the nature-inspired optimization framework NiaPy.
3. The framework provides an easily extendable and customizable API, that can be customized with any scikit-learn compatible decision model or NiaPy compatible optimization method.
4. The framework provides data preprocessing for regression supervised learning problems.

The implementation of preprocessing tasks in the provided framework is on-par or better in comparison to other available approaches. Thus, the framework can be used as-is, but its main strength is that it provides a framework on which others can build upon to provide various specialized preprocessing approaches. The framework handles the parallelization and the evaluation of the optimization process and addresses the data leakage problem without any additional input needed.

The rest of the paper is organized as follows. The next section contains the problem formulation of the three preprocessing tasks and the literature overview of data preprocessing with the nature-inspired method. Section 3 contains the implementation details of the presented framework, which is followed by the fourth section with examples of use. Finally, concluding remarks are provided in the fifth section of the paper.

2. Problem Formulation

Let $X \in \mathbb{R}^{n \times m}$ be a data matrix (data set) where n denotes the number of instances (samples) in the data set, and m is the number of features. The data set X consists of instances $x_1, x_2, \ldots, x_n$, where $x_i \in \mathbb{R}^m$ and is written as $X = [x_1, x_2, \ldots, x_n]$. Also, the data set X consists of features $f_1, f_2, \ldots f_m$, where $f_i \in \mathbb{R}^n$ and is denoted as $X = [f_1, f_2, \ldots, f_m]$. As we are dealing with a supervised data mining problem, there is also Y, which is a vector of target values which are to be predicted. If the problem is in a type of classification, the values in Y are nominal $Y \in \{c_1, c_2, \ldots, c_k\}$, where k is the number of predefined discreet categories or classes. On the contrary, if we are dealing with regressing, the target values in Y are continuous $Y \in \mathbb{R}^n$. The goal of supervised learning is to construct the model

M, which can map X to $\hat{Y}$ while minimizing the difference between the predicted target values $\hat{Y}$ and true target values Y.

Feature selection handles the curse of dimensionality when machine learning models tend to over-fit on data with a too large set of features [23]. It is a technique where most relevant features are selected, while the redundant, noisy or irrelevant features are discarded. The result of using feature selection is improved learning performance, increased computational efficiency, decreased memory storage space needed and more generalizable models [5]. In mathematical terms, the feature selection transforms the original data X to the new X_{FS} (Equation (1)), with a potentially smaller set of features f'_i than in the original set.

$$\begin{aligned} X &= [f_1, f_2, \ldots, f_m] \\ X_{FS} &= [f'_1, f'_2, \ldots, f'_l] \\ l &\leq m \\ l &> 0 \end{aligned} . \tag{1}$$

Feature selection has already been addressed with meta-heuristic and nature-inspired optimization methods, as has been demonstrated in review papers [24,25]. Lately, the research topic has gained extensive focus from the nature-inspired optimization research community, with the application of every type of nature-inspired method to the given problem—whale optimization algorithm [26], dragonfly [27] and chaotic dragonfly algorithm [28], grasshopper algorithm [29], grey wolf optimizer [30,31], differential evolution and artificial bee colony [32], crow search [33], swarm optimization [34], genetic algorithm [35] and many others.

On the other hand, **data sampling** and **data weighting** (sometimes *instance weighting*) address the problem of improper ratios of instances in the learning data set [36]. The problem is two-fold—some types of instances can be over-represented (the majority), and other instances can be under-represented (the minority).

Imbalanced data sets can form for various reasons. Either there is a natural imbalance in real-life cases, or certain instances are more difficult to collect. For example, some diseases are not common and therefore patients with similar symptoms without the rare disease are much more common than the patients with symptoms that have the rare condition [37,38]. In other cases, the balanced and representative collection of data that reflects the population is sometimes problematic or even impossible. One major cause of this problem in social domains is the well-documented self-selection bias, where only a non-representative group of individuals select themselves into the group [39]. Convenience sampling is the next reason for over-representation of some and under-representation of other samples [40]. For example, the cost-effectiveness of data collection can also contribute to the emergence of majority and minority cases, when minorities are expensive (be it time- or financial cost-wise) to obtain.

Furthermore, machine learning models require an appropriate representation of all types of instances to be able to extract the signal and not confuse it with noise [41]. Data sampling [42–44] and cost-sensitive instance weighting [45] have already been tackled with meta-heuristic methods, evolutionary algorithms, and nature-inspired methods.

In the **data sampling** task, we transform the original data set X to X_S, with a potentially different distribution of instances. Note, that we can (1) *under-sample* the data set–only select the most relevant instances, (2) *over-sample* the data set–introduce copies or new instances to the original set X, or (3) *simultaneously under- and over-sample* the data set. The latter removes redundant instances and introduces new ones in the new data set X_S. With the EvoPreprocess framework, the simultaneous under- and over-sampling is used, where instances can be removed, and copies of existing instances can be introduced. The size n' of the new set X_S can be different or equal to the size n of the original

set X, but the distribution of the instances in the X_S should be different from the X. This is shown in Equation (2).

$$\begin{gathered} n = |X| \\ n' = |X_S| \\ \\ n' > 0 \\ X_S \neq X \\ X_S \cap X \neq \emptyset. \end{gathered} \tag{2}$$

On the other hand, **data weighting** does not alter the original data set X, but introduces the importance factor of instances in the X, called weights. The greater the importance of the instance, the bigger the weight for that instance, and vice versa—the lesser the importance, the smaller the instance weight. Fitting the machine learning model on weighted instances is called *cost-sensitive* learning, and can be utilized in several different machine learning models [46]. Let us denote the vector of weights as W, which consists of individual weights $w_1, w_2, \ldots, w_n$, where $w_i \in \mathbb{R}^+$ as is presented in Equation (3).

$$\begin{gathered} W = [w_1, w_2, \ldots, w_n] \\ |W| = |X| \end{gathered}. \tag{3}$$

While there are some rudimentary feature selection, data weighting, and data sampling methods in the Python machine learning framework scikit-learn, again, there is a lack of evolutionary and nature-inspired methods either included in this framework or, as independent open-source libraries, compatible with it. EvoPreprocess intends to fill this gap by providing a scikit-learn compatible toolkit in Python, which can be extended easily with the custom nature-inspired algorithms.

2.1. Nature-Inspired Preprocessing Optimization

Nature-inspired optimization algorithms is a broad term for meta-heuristic optimization techniques inspired by nature, more specifically by biological systems, swarm intelligence, physical and chemical systems [47,48]. In essence, these algorithms look for good enough solutions to any optimization problem with the formulation [47,49] in Equation (4).

$$\begin{aligned} S_t &= [s_1, s_2, \ldots, s_k] \\ S_{t+1} &= A\left\{S_t; (p_1, p_2, \ldots, p_h); (w_1, w_2, \ldots, w_g)\right\}. \end{aligned} \tag{4}$$

The set S_t is a set of solutions $[s_1, s_2, \ldots, s_k]$ in the iteration t. The next iteration of solutions S_{t+1} is generated using an algorithm A in accordance to the solution set in the previous iteration S_t, the parameters $(p_1, p_2, \ldots, p_h)$ of the algorithm A, and random variables $(w_1, w_2, \ldots, w_g)$.

These algorithms have already been successfully applied to the preprocessing tasks [11,12,25], proving the validity and the efficacy of these methods.

2.1.1. Solution Encoding for Preprocessing Tasks

The base of any optimization method is the solution s to the problem, which is encoded in a way that changing operators can be applied on it to minimize the distance between the solution s and the optimal solution. A broad set of nature-inspired algorithms work with the encoding of one solution in the form of an array of values. If the values in the array are continuous numbers, we are dealing with a continuous optimization problem. Alternatively, we also have a discrete optimization problem where values in the array are discrete. Even though some problems need a discrete solution, one can use continuous optimization techniques that can be mapped to the discrete search space [50]. Figure 1

shows the common solution encoding s for feature selection, data sampling, and data weighting and the transformation from a continuous search space to a discrete one, where this is applicable (data sampling and feature selection).

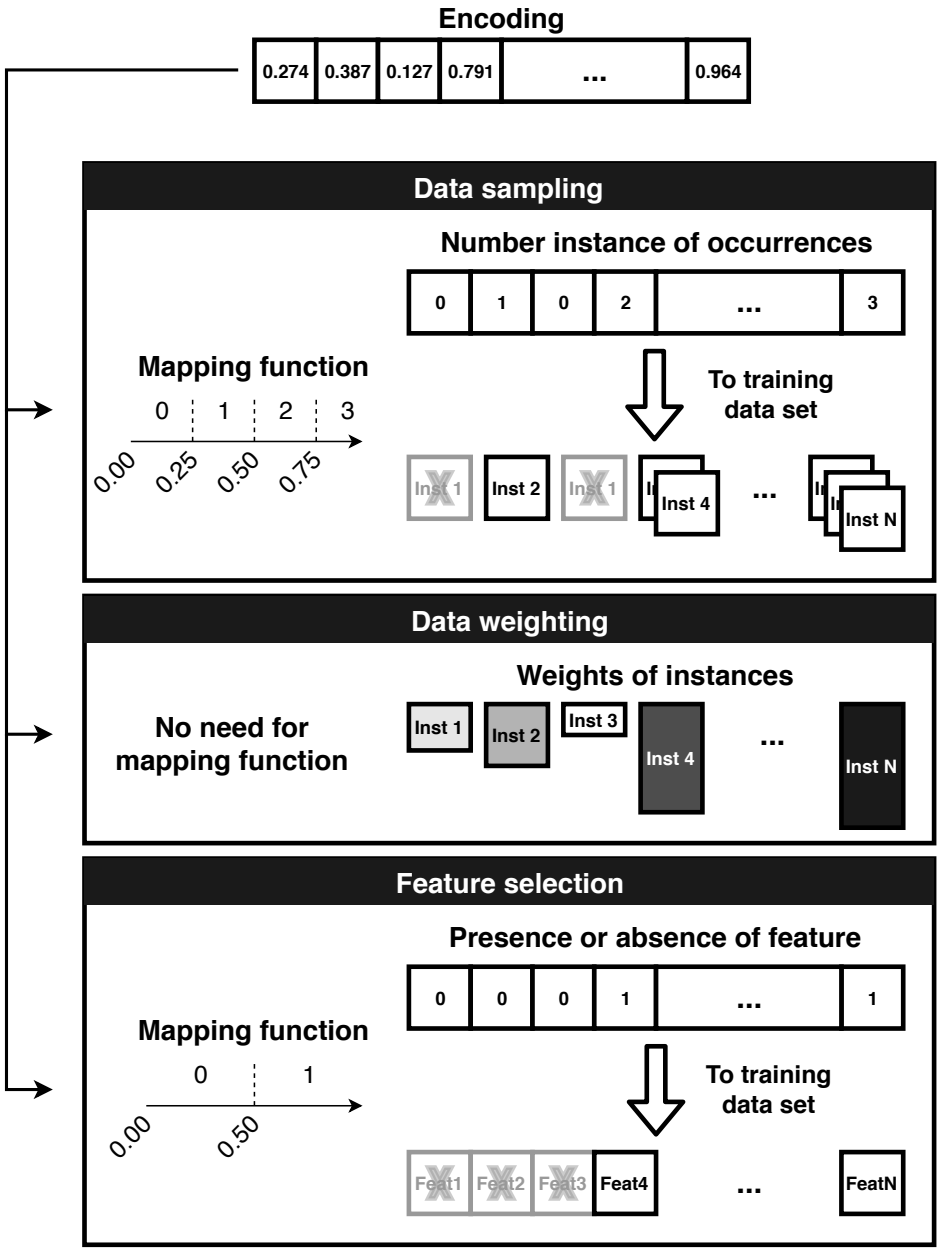

Figure 1. Encoding and decoding of solutions for data preprocessing tasks.

As Figure 1 shows, the encoding of *data sampling* with discrete values is straightforward—one value in the encoding array corresponds to one instance from the original data set X, and the scalar value represents the number of occurrences in the sampled data set X_S. When using continuous optimization, one can use mapping function m, which splits the continuous search space into bins, each with its corresponding discrete value. Note that discrete values of occurrences can take any of the non-negative integers, but solution encoding can be any non-negative real value. This is reflected in Equation (5).

$$\begin{aligned} s_S &= [e_1, e_2, \ldots, e_k] \\ e_i &\in \mathbb{R}_{\geq 0} \\ \\ G_S &= m(s_S) \\ m &: \mathbb{R}_{\geq 0} \longrightarrow \mathbb{N}_0 \\ \\ G_S &= [g_1, g_2, \ldots, g_k] \\ g_i &\in \mathbb{N}_0 \end{aligned} \tag{5}$$

The encoding for *data weighting* is even more straightforward. Again, each value in the array corresponds to one instance in the data set X, and the scalar values represent the actual weights W of the instances. There is no need for the mapping function, as long as we limit the interval of allowed values for scalars in solution e_i to $[0, max\_weight]$ as shown in Equation (6). Some implementations of the machine learning algorithms accept only weights up to 1, thus limiting the search space to $[0, 1]$; others have no such limit, broadening the search space to $[0, \infty)$.

$$\begin{aligned} s_W &= [e_1, e_2, \ldots, e_k] \\ e_i &\in \mathbb{R}_{\geq 0}. \end{aligned} \tag{6}$$

Encoding for *feature selection* is in the form of an array of binary values—the feature is either present (value 1) or absent (value 0) from the changed data set X_{FS} (see Equation (7)). Using the continuous solution encoding, one should again use the mapping function m, which splits the search space into two bins (with arbitrary limits), one for the feature being present and one for the feature being absent.

$$\begin{aligned} s_{FS} &= [e_1, e_2, \ldots, e_k] \\ e_i &\in \mathbb{R}_{\geq 0} \\ \\ G_{FS} &= m(s_{FS}) \\ m &: \mathbb{R}_{\geq 0} \longrightarrow \{0, 1\} \\ \\ G_{FS} &= [g_1, g_2, \ldots, g_k] \\ g_i &\in \{0, 1\}. \end{aligned} \tag{7}$$

2.1.2. Self-Adaptive Solutions

The presented solution encoding shows that there are different options for mapping the encoding to the actual data set. These mapping settings are the following:

- In *data weighting* the maximum weight can be set,
- in *data sampling* the mapping from the continuous value to the appearance count can be set, and
- in *feature selection* the mapping from the continuous value to presence or absence of the feature can be set.

In general, these values can all be set arbitrary, but could also be one of the objectives of the optimization process itself. The values of these parameters can seriously influence the quality of the results in preprocessing tasks [51–53], as it can guide the evolution of the optimization to the global optima, rather than to the local one. The implementation of the preprocessing tasks in EvoPreprocess uses this self-adaptive approach with the additional genes in the genotype [54].

In the *data weighting* task, there is one additional gene, which corresponds to the maximum weight that can be assigned. All of the genes corresponding to the weights are normalized to this maximum value while leaving the minimum at 0. When using the imbalanced data set it is preferred that bigger differences in weights are possible.

In *data sampling* the mapping of the interval $[0, 1]$ to the instance occurrence count is set. Here, n_{max} setting genes are added to the genotype, where n_{max} is the maximum number of occurrences on the individual instance after it is over-sampled. Each setting gene presents the size of the mapping interval of an individual occurrence count. Figure 2 presents the process of splitting the encoding genome to the sampling and the self-adaptation parts and mapping it to the solution. Note, that in Figure 2 the mapping intervals are just an example of one such self-adaptation and are not set as the final values used in all mappings. These values differ from solutions and are also data set dependent.

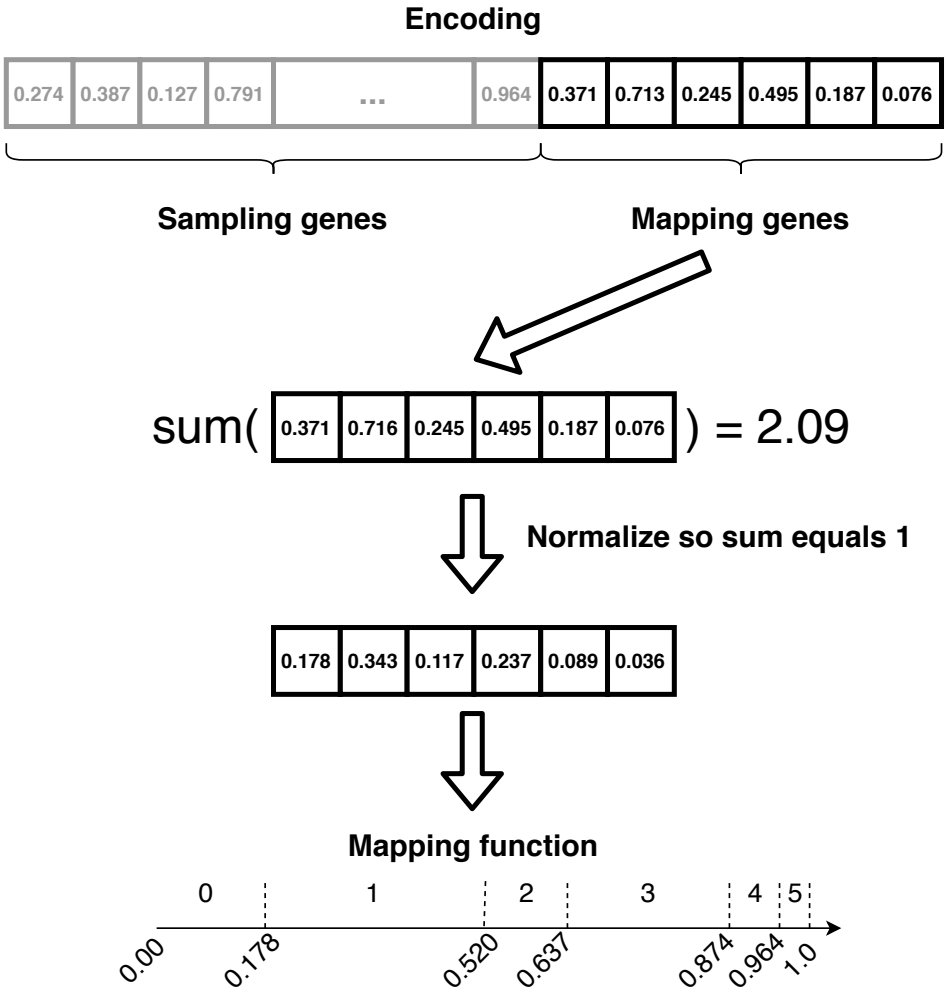

Figure 2. The self-adaptation with mapping genes in the encoding for data sampling task. The mapping values presented in this example are determined by the genotype and are not fixed, but adapt during the evolution process.

Some heavily imbalanced data sets could be better analysed if the emphasis is on under-sampling—the interval for 0 occurrences (the absence of the instance) would become bigger in the process. Other data sets could be more suitable if there are more minority instances, and therefore intervals for many occurrences become bigger within the optimization process.

With *feature selection* where is only one setting gene added to the genotype. This gene represents the size of the first interval, which maps other genes to the absence of the feature. Wider the data sets with more features could be analyses with better results if more features are removed from the data

sets. Therefore, the larger the number in the setting gene is preferred, when the more emphasis is given to the shrinkage of the data set as more features are not selected. Again, the mapping interval in Figure 3 are just an example for that particular mapping gene in the encoded genotype.

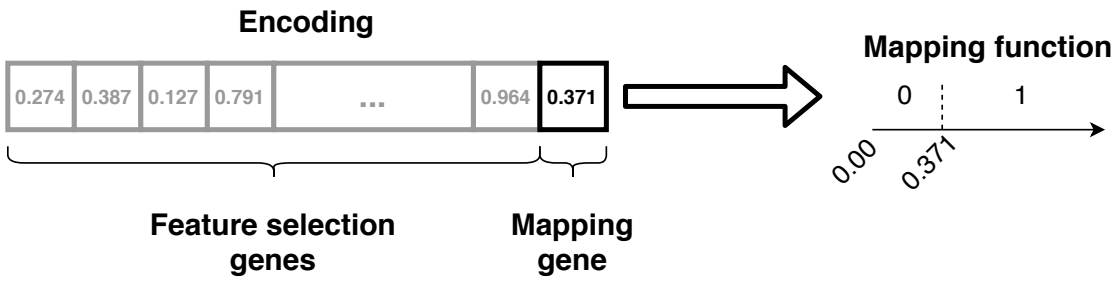

Figure 3. The self-adaptation with mapping gene in the encoding for feature selection task. The mapping value presented in this example are determined by the genotype and are not fixed, but adapt during the evolution process.

The self-adaptation is implemented in the framework but could be removed in the extension or customization of the individual optimization process. It is included in the framework as it is an integral part of the optimization process in recent state-of-the-art papers [53,55–57] on data preprocessing with nature-inspired methods.

2.1.3. Optimization Process for Preprocessing Tasks

All nature-inspired methods run in iterations during which the optimization process is applied. The broad overview of the nature-inspired optimization algorithm is shown in Figure 4 and is based on the rudimentary framework for nature-inspired optimization of feature selection by [25].

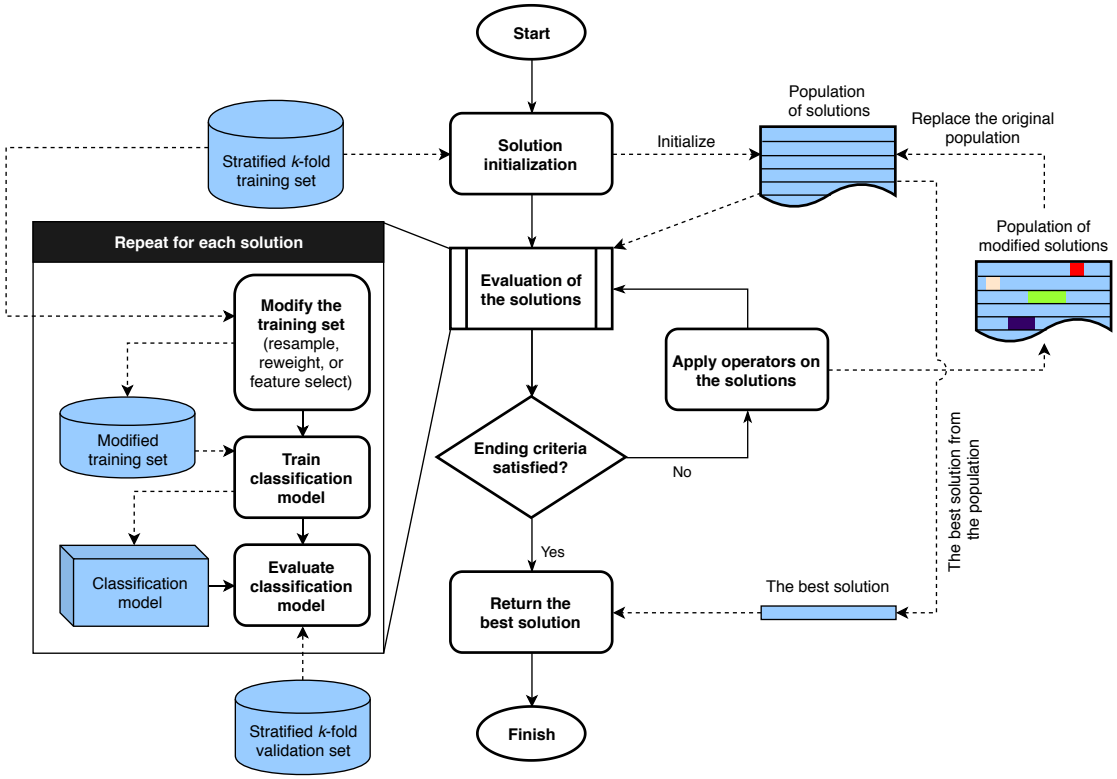

Figure 4. The optimization process of preprocessing data with nature-inspired optimization methods in EvoPreprocess.

The optimization process starts with the initialization of solutions, which is either random or with some heuristic methodology. After that, the iteration loop starts. First, every solution is evaluated, which will be discussed later in this section. Next, the optimization operators are applied to the solutions, which are specific to each the nature-inspired algorithm. The general goal of the operators is to select, repair, change, combine, improve, mutate, and so forth,. solutions in the direction of a perceived (either local or, hopefully, global) optimum. The iteration loop is stopped when a predefined limit of ending criteria is reached, be it the maximum number of iterations, the maximum number of iterations when the solutions stagnate, the quality that solutions reach, and others.

Nature-inspired algorithms strive to keep good solutions and discard the bad ones. Here, the method of evaluating solutions plays an important role. One should define one or more objectives that solutions should meet, and the evaluation function grades the quality of the solution based on these objectives. As solutions are evaluated every iteration, this is usually the most computationally time-consuming process of all steps in the optimization process.

When dealing with the optimization of the data set for the machine learning process, multiple objectives have been considered in the literature [25,58–61]. Usually, one of the most important objectives is the quality of the fitted model from the given data set. If the classification task is used the standard classification metrics could be used: accuracy, error rate, F-measure, G-measure, area under the ROC curve (AUC). If we are dealing with the regression task, the following regression tasks can be used—mean squared error, mean absolute error, or explained variance. By default, the EvoPreprocess framework uses a single-objective optimization to obtain either the error rate and F-score for the classification and the mean squared error for the regression tasks. As the later sections will show, the framework is easily extendable to be used for multiple objectives, be it the size of the data set or others.

It is important to note that not all researchers consider the problem of data leakage. *Data leakage* occurs when information from outside the training data set is used in the fitting of the machine learning models. This manifests in over-fitted models that perform exceptionally well on the training set, but poorly on the hold-out testing set. A common mistake that leads to data leakage is not using the validation set when optimizing the model fitting. Applying the same logic to nature-inspired preprocessing, data leakage occurs when the same data are used in the evaluation process and the final testing process. The EvoPreprocess framework automatically holds-out the separate validation sets and thus, prevents data leakage.

2.2. Nature-Inspired Algorithms

A large number of different nature-inspired algorithms were proposed in recent years. The recent meta-heuristic research of nature-inspired algorithm was reviewed by Lones [62] in 2020, where he concluded that most recent innovations are usually small variations of already existing optimization operators. Still, there are some novel algorithms worth further investigation: polar bear optimization algorithm [63], bison algorithm optimization [64], butterfly optimization [65], cheetah based optimization algorithm [66], coyote optimization algorithm [67] and squirrel search algorithm [68]. Due to the amount of nature-inspired algorithms, a broad overview is beyond the scope of this paper. Consequently, this section provides the formulation for the well-established and most used nature-inspired algorithms.

First, *Genetic algorithm* (GA) is one of the cornerstones of nature-inspired algorithms, presented by Sampson [69] in 1976, but research efforts on its the variations and novel applications are still numerous. The basic operators here are the selection of the solutions (called individuals) that are then used for the crossover (the mixing of genotype) which forms new individuals, which have a chance to go through a mutation procedure (random changing of genotype). The crossover is an exploitation operator where the solutions are varied to find their best variants. On the other hand, the mutation is a prime example of an exploration operator, which prevent the optimization to get stuck in the local optima. Each iteration is called a generation and can repeat until predefined criteria, be it in a form

of maximum generations or stagnation limit. Most of the following optimization algorithms use a variation of the presented operators.

Next, *Differential evolution* (DE) [70] includes the same operators as GA, selection, crossover and mutation, but multiple solutions (called agents) can be used. In its basic form, the crossover of three agents is not done with the simple mixing of genes (like in GA), but the calculation from Equation (8) for each gene is used. Here $x_{parent1}$, $x_{parent2}$ and $x_{parent3}$ are gene values from three parents, x_{new} is the new value of the gene, and F is the differential weight parameter.

$$x_{new} = x_{parent1} + F * (x_{parent2} - x_{parent3}). \tag{8}$$

Evolution strategy (ES) [71] is an optimization algorithm, which has similar operators to GA, but the emphasis is given to the selection and mutation of the individuals. The most common variants are the following: $(\mu/\rho, \lambda)$, where only new solutions form the next generation and $(\mu/\rho + \lambda)$ where the old solutions compete with the new ones. The parameter μ presents the number of selected solutions for the crossover and mutation from where ρ are derived. Value λ denotes the size of the generation.

A variation of Evolution strategy is the *Harmony search* (HS) optimization algorithm [72] which mimics the improvisation of a musician. The main search operator is the pitch adjustment, which adds random noise to existing solutions (called harmony) or creates a new random harmony. Creation of new harmony is shown in Equation (9). where x_{old} denotes the old gene from the old genotype, x_{new} is the new gene solution, b_{range} denotes the range of maximal improvisation (change of solution) and ϵ is a random number in the interval $[-1, 1]$.

$$x_{new} = x_{old} + b_{range} * \epsilon. \tag{9}$$

Next group of nature-inspired algorithms mimic the behaviour of swarms and are called *swarm optimization* algorithms. *Particle swarm optimization* (PSO) [73] is a prime example of swarm algorithms. Here each solution (called particle) is supplemented with its velocity. This velocity represents the difference of change from its current position (genotype encoding) in the new iteration for this solution. After each iteration, the velocities are recalculated to direct the particle to the best position. The moving of particles in the search space represents both, the exploitation (moving around the best solution) and the exploration (moving towards the best solution) parts of optimization. The iterative moving of particles stops one of the following criteria is satisfied—the maximum number of iterations is reached, the stagnation limit is reached, or particles converge to one best position.

One variation of PSO is *Artificial bee colony* (ABC) algorithm [74] which imitates the foraging process of a honey bee swarm. This optimization algorithm consists of three operators which modify existing solutions. First, employed bees use local search to exploit already existing solutions. Next, the onlooker bees, which serve as a selection operator, search for new sources in their vicinity of existing ones (exploitation). And last, the scout bees are used for the exploration where they use a random search to find new food sources (new solutions).

Next, the *Bat algorithm* (BA) [75] which is a variant of PSO which imitates swarms of microbats, where every solution (called bat) still has a velocity, but also emits pulses with varying levels of loudness and frequency. The velocity of bats changes in consideration to the pulses from other bats and the pulses are determined by the quality of the solution. Equation (10) shows the procedure for updating the genes of individual bats. Here x_{old} and x_{new} denote the old and the new genes respectively, v_{old} and v_{new} are the old and the new velocities of the bats, and f_i, f_{min} and f_{max} are current, minimal and maximal frequencies. β is a random number in the interval $[0, 1]$.

$$\begin{aligned} f_i &= f_{min} + (f_{max} - f_{min}) * \beta \\ v_{new} &= v_{old} + (x_{old} - x_{best}) * f_i \\ x_{new} &= x_{old} + v_{new}. \end{aligned} \tag{10}$$

Finally, one of the widely used swarm algorithms is *Cuckoo search* (CS) [76], which imitates laying of eggs in the foreign nest by cuckoo birds. The solutions are eggs in the nests and those are repositioned to new nests every iteration (exploitation), while the worst ones are abandoned. The modification of the solutions is done with the optimization operator called the Lévy flight, which is in the form of long random flights (exploration) or short random flights (exploitation). The migration of the eggs with Lévy flight is shown in Equation (11), where α denotes the size of the maximal flight and $Levi(\lambda)$ represents a random number from Lévi distribution.

$$x_{new} = x_{old} + \alpha * Levi(\lambda). \tag{11}$$

2.3. Computation Complexity

In general, most nature-inspired algorithms have time complexity of $O(m * p * C_{operators} + p * C_{fitness_eval})$, where m is the size of the solution, p is the number of modified and evaluated solutions during the whole process, $C_{operators}$ is the complexity of the operators (i.e., crossover, recombination, mutation, selection, random jumps...), and $C_{fitness_eval}$ is the complexity of the evaluation.

The evaluation of solutions is the most computationally expensive part. One of the classification/regression algorithms must be used in order to (1) build the model, (2) make the predictions, and (3) evaluate the predictions. This part is heavily reliant on the chosen solution evaluator (the classification algorithm used). The evaluation of the prediction is a fixed $O(n)$ complexity, dependent on n samples in the data set. Training of the models and making predictions vary: decision tree with $O_{training}(n^2p)$ and $O_{prediction}(p)$, linear and logistic regression with $O_{training}(p^2n + p^3)$ and $O_{prediction}(p)$, and naive Bayes with $O_{training}(np)$ and $O_{prediction}(p)$. Some of the ensemble methods have and additional factor of the number of models in the ensembles (i.e., Random forest and AdaBoost), and neural networks depend on the net architecture. Usually, the more complex the training process, the more complex patterns can be extracted from the data and consequently, the predictions are better.

If the evaluator and its use are fixed (as is in the experimental part of the paper), optimization time varies in relation to the nature-inspired optimization algorithms used. Considering only the array style solution encoding, the basic exploration and exploitation type operators (i.e., crossover, recombination, mutation and random jumps) have $O(m * r * C_{operation})$ complexity, where m is the size of the encoded solution, r is the number of solutions used in the operator, and $C_{operation}$ is the complexity of the operation (i.e., linear combination, sum, distance calculation...). As the m size of the solution is usually fixed, researchers must optimize other aspects of the algorithms to get shorter computation times. Thus, the fastest optimization algorithms are the ones with the fewest operators, the operators with the fewest solutions participating in the optimization, or the lest complex operators. For example, the genetic algorithm has multiple relatively simple operators: crossover, mutation, elitism, selection (i.e., tournament where the fitness values are compared) and thus is one of the more time-consuming ones.

Furthermore, algorithms, where operators can be vectorized, can take advantage of computational speed-ups when low-level massive computation calls can be used. This is especially prevalent in swarm algorithms, where calculating distances or velocities and then moving the solutions can be done with matrix multiplications for all of the solution set at once, instead of individually for every solution. If the speed of the data preprocessing is of the essence, this should be taken into account in the implementation of the optimization algorithms. EvoPreprocess framework already provides parallel runs of the optimization process data set folds, but its speed is still reliant on the nature-inspired algorithms and evaluators used in the process.

3. EvoPreprocess Framework

The present section provides a detailed architecture description of the EvoPreprocess framework, which is meant to serve as a basis for further third party additions and extensions.

The EvoPreprocess framework includes three main modules:

- `data_sampling`
- `data_weighting`
- `feature_selection`

Each of the modules contains two files:

1. Main **task class** is to be used for running tasks data sampling, feature selection or data weighting;
2. Standard **benchmark class**, which is a default class used in the evaluation of the task, and can be replaced or extended by custom evaluation class.

Figure 5 shows a UML class diagram for the EvoPreprocess framework and its relation to the Python packages scikit-learn, imbalanced-learn and NiaPy. The custom implementation of benchmark classes are in the classes `CustomSamplingBenchmark`, `CustomWeightingBenchmark` and `CustomFeatureSelectionBenchmark`, which are denoted with dark background color.

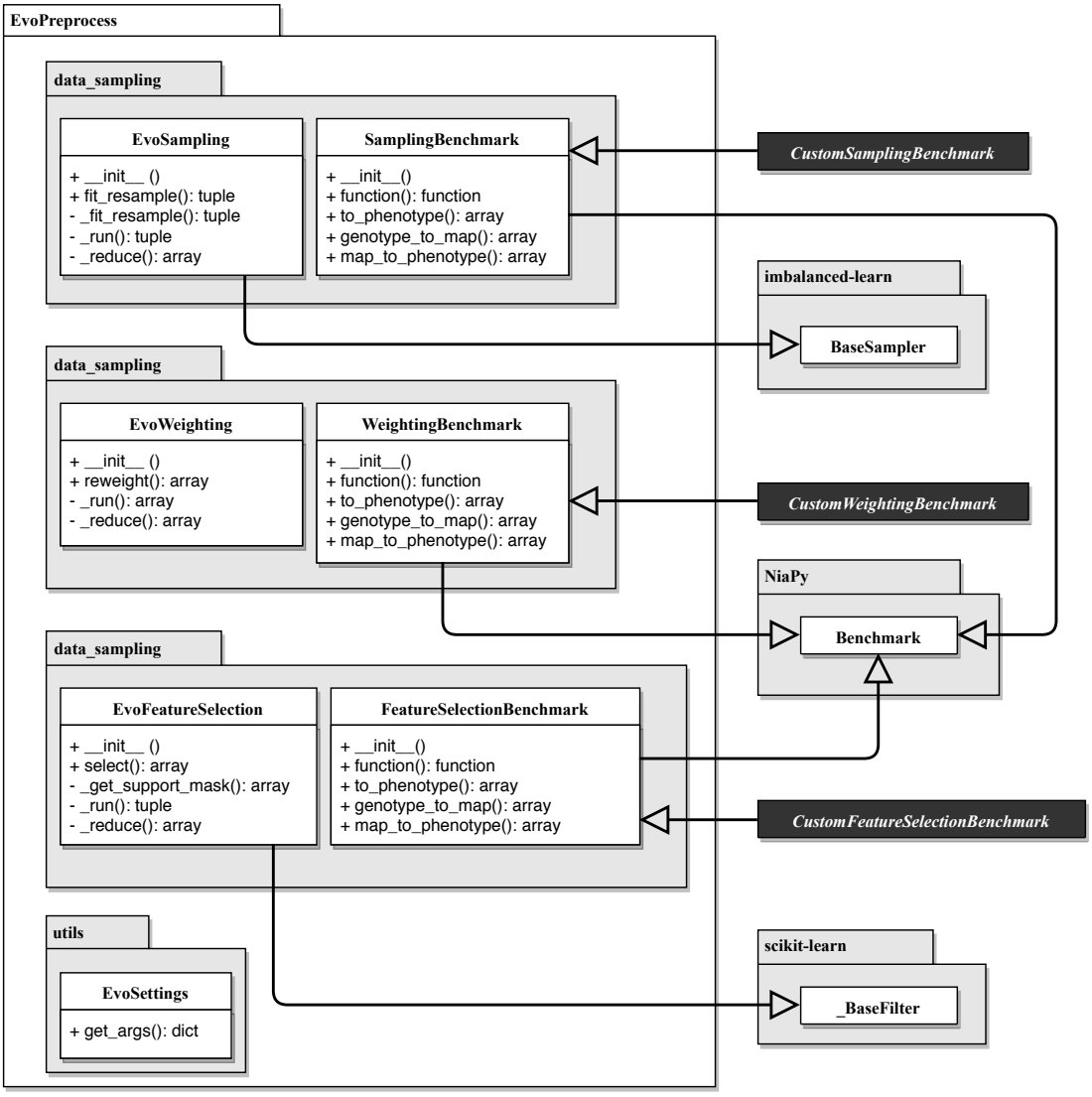

Figure 5. UML class diagram of the EvoPreprocess framework.

3.1. Task Classes

The task class for data sampling is `EvoSampling`, which extends the imbalanced-learn class for sampling data `BaseOverSampler`. The task class for feature selection, `EvoFeatureSelection`, extends `_BaseFilter` class from scikit-learn. The main task class for data weighting does not use any parent class. All three task classes are initialized with the following parameters.

- `random_seed` ensures reproducibility. The default value is the current system time in milliseconds.
- `evaluator` is the machine learning supervised approach used for the evaluation of preprocessed data. Here, scikit-learn compatible classifier or regressor should be used. See description of the **benchmark** classes for more details.
- `optimizer` is the optimization method used to get the preprocessed data. Here, the NiaPy compatible optimization method is expected, with the function `run` and the usage of the evaluation benchmark function. The default optimization method is the genetic algorithm.
- `n_folds` is the number of folds for the cross-validation split into the training and the validation sets. To prevent data leakage, the evaluations of optimized data samplings should be done on the hold-out validation sets. The default number of folds is set to 2 folds.
- `n_runs` is the number of independent runs of the optimizer on each fold. If the optimizer used is deterministic, just one run should be sufficient, otherwise more runs are suggested. The default number of runs is set to 10.
- `benchmark` is the evaluation class which contains the function that returns the quality of data sampling. The custom benchmark classes should be used here if the data preprocessing objective is different from the singular objective of optimizing error rate and F-score (for classification) or mean squared error (for regression).
- `n_jobs` is the number of optimizers to be run in parallel. The default number of jobs is set to the number of CPU cores.

The base optimization procedure pseudo-code is demonstrated in Algorithm 1, where the train-validation split (lines 3 and 4) and multiple parallel runs (for loop in lines 6, 7 and 8) are shown. Note that the X_T data set given to the procedure should already be the training set and not the whole data set X. The further splitting of X_T into training and validation data sets ensures that data leakage does not happen. To ensure that random splitting into training and validation sets does not produce split dependent results, the stratified k-fold splitting is applied (into n_folds folds). As the nature of nature-inspired optimization methods is non-deterministic, optimization is done multiple times (n_runs parameter) and is run in parallel—every optimization on the separate CPU core. The reduction (aggregation) of the results to one final results is done in different ways, dependent on the preprocessing task.

- In *data sampling* the best performing instance occurrences in every fold are aggregated with mode.
- In *data weighting* the best weights in every fold are averaged.
- In *feature selection* the best performing selected features in every fold are aggregated with mode.

The preprocessing procedure from Algorithm 1 is called in different way in every task, which is the consequence of different inheritance for every task: data sampling inherits from scikit-learn `_BaseFilter`, feature selection inherits from imbalanced-learn `BaseSampler` and data weighting does not inherit from any class.

All three task classes contain the following functions.

- `_run` is a private static function to create and run the `optimizer` with the provided evaluation `benchmark` function. Multiple calls of this function can be run in parallel.
- `_reduce` is a private static function used to aggregate (reduce) the results of individual runs on multiple folds in one final sampling.

Algorithm 1: Base procedure for running preprocessing optimization.

input : X_T training data set
n_folds number of folds for k-fold validation
n_runs number of individual parallel runs for each fold
$optimizer$ nature-inspired optimization algorithm
$evaluator$ classifier/regressor for evaluation processed data set
$benchmark$ function to evaluate preprocessed data with $evaluator$
output: $X_{processed}$ processed data set

```
results ← ∅
for i ← 1 to n_folds do
    training_set ← getTrainingData(X_T)
    validation_set ← getValidationData(X_T)
    split_results ← ∅
    // This for loop is run in parallel
    for r ← 1 to n_runs do
        X' ← _run(training_set, validation_set, optimizer, evaluator, benchmark)
        split_results ← split_results ∪ X'
    end
    best_result ← best (split_results)
    results ← results ∪ best_result
end
X_processed ←_reduce(results)
```

In addition to those functions, the class `EvoSampling` also contains the function which is used for sampling data.

- `_fit_resample` private function which gets the data to be sampled `X` and the corresponding target values `y`. This is an implementation of the abstract function from imbalanced-learn package, is called from the public `fit_resample` function from the `BaseOverSampler` class, and provides the possibility to be included in imbalanced-learn pipelines. It contains the main logic of the sampling process. The function returns a tuple with two values: `X_S` which is a sampled `X` and `y_S` which is a sampled `y`.

The class `EvoWeighting` also contains the following function.

- `reweight`, which gets the data to be reweighted `X` and the corresponding target values `y`. This function returns the array `weights` with weights for every instance in `X`.

The class `EvoFeatureSelection` contains the following functions.

- `_get_support_mask` private function, which checks if features are already selected. It overrides the function from the `_BaseFilter` class from scikit-learn.
- `select` is a score function that gets the data `X` and the corresponding target values `y` and selects the features from `X`. This function is provided as the scoring function for the `_BaseFilter` class so it can be used in scikit-learn pipelines as the feature selection function. This function returns the `X_FS`, which is derived from `X` with potentially some features removed.

3.2. Evaluation of Solutions with Benchmark Classes

The second part of classes in all the modules are **benchmark** classes, which are helper classes meant to be used in the evaluation of the samplings, weighting or feature selection tasks. The implementation of custom fitness evaluation function should be done by replacing or extending of these classes. The benchmark classes are initialized with the following parameters.

- `X` data to be preprocessed.
- `y` target values for each data instance from `X`.
- `train_indices` array of indices of which instances from `X` should be used to train the evaluator.
- `valid_indices` array of indices of which instances from `X` should be used for validation.
- `random_seed` is the random seed for the evaluator and ensures reproducibility. The default value is 1234 to prevent different results from evaluators initialized at different times.
- `evaluator` is the machine learning supervised approach used for the evaluation of the results of the task. Here, scikit-learn compatible classifier or regressor should be used: evaluator's function `fit` is used to construct the model, and function `predict` is used to get the predictions. If the target of the data set is numerical (regression task) the regression method should be provided, otherwise the classification method is needed. The default evaluator is `None`, which sets the evaluator to either linear regression if the data set target is a number, or Gaussian naive Bayes classifier if the target is nominal.

The benchmark classes all must provide one function—`function` which returns the evaluation function of the tasks. This architecture is in accordance with NiaPy benchmark classes for their optimization methods and was used in EvoPreprocess for compatibility reasons.

Three benchmark classes are provided, each for its task to be evaluated:

- `SamplingBenchmark` for data sampling,
- `WeightingBenchmark` for data weighting, and
- `FeatureSelectionBenchmark` for feature selection.

All provided benchmarks classes evaluate the task in the same way: evaluator is trained on the training set selected with `train_indices` and evaluated on the validation set selected with `valid_indices`. If the evaluator provided is a classifier, the sampled, weighted or feature selected data are evaluated with *error rate*, which is defined in the Equation (12) and presents the ratio of misclassified instances [77].

$$Error\ rate = \frac{1}{n}\sum_{n}^{i-1} z \begin{cases} 0 & \text{if } y_i = \widehat{y_i} \\ 1 & \text{if } y_i \neq \widehat{y_i} \end{cases} \tag{12}$$

The *F-score* can be used to evaluate the classification quality, where the balance between precision and recall is considered [77]. As the optimization strives to minimize the solution values, $1 -$ F-score is used then. If the evaluator is a regressor, the new data set is evaluated using *mean squared error* presented in Equation 13 and calculates the average of absolute error in predicting the outcome variable [77].

$$MSE = \frac{1}{n}\sum_{n}^{i-1} (y_i - \widehat{y_i})^2 . \tag{13}$$

4. Examples of Use

This section presents examples of using EvoPreprocess in all three supported data preprocessing tasks: first, the problem of data sampling, next, we cover data weighting, and finally, the feature selection problem. Before using the framework, the following requirements must be met.

- Python 3.6 must be installed,
- The NumPy package,
- The scikit-learn package, at least version 0.19.0,
- The imbalanced-learn package, at least version 0.3.1,
- The NiaPy package, at least version 2.0.0*rc*5, and
- The EvoPreprocess package can be accessed at https://github.com/karakatic/EvoPreprocess.

4.1. Data Sampling

One of the problems addressed by the EvoPreprocess is data sampling; or more specifically, simultaneous under- and over-sampling of data with class `EvoSampling` in the module `data_sampling` and its function `fit_resample()`. This function returns a two arrays; a two-dimensional array of sampled instances (rows) and features (columns), and a one-dimensional array of sampled target values from the corresponding instances from the first array. The code below shows the basic use of the framework for resampling data for the classification problem, with default parameter values.

```
>>> from sklearn.datasets import load_breast_cancer
>>> from EvoPreprocess.data_sampling import EvoSampling
>>>
>>> dataset = load_breast_cancer()
>>> print(dataset.data.shape, len(dataset.target))
[569,30] 569

>>> X_resampled, y_resampled = EvoSampling().fit_resample(dataset.data, dataset.target)
>>> print(X_resampled.shape, len(y_resampled))
[341,30] 341
```

The results of the run show that there were 569 instances in the data set before sampling and there are 341 instances in the data set after the sampling, which shows that more samples were removed from the data set than there were added.

The following code shows the usage of data sampling for the data set for the regression problem. The code also demonstrates the setting of parameter values: a non-default optimizer method of evolution strategy, 5 folds for validation, 5 numbers of individual runs on each fold split and 4 parallel executions of individual runs.

```
>>> from sklearn.datasets import load_boston
>>> from EvoPreprocess.data_sampling import EvoSampling, SamplingBenchmark
>>>
>>> dataset = load_boston()
>>> print(dataset.data.shape, len(dataset.target))
(506, 13) 506

>>> X_resampled, y_resampled = EvoSampling(
evaluator=DecisionTreeRegressor(),
optimizer=nia.Evolution strategy,
n_folds=5,
n_runs=5,
n_jobs=4,
benchmark=SamplingBenchmark
).fit_resample(dataset.data, dataset.target)
>>> print(X_resampled.shape, len(y_resampled))
(703, 13) 703
```

Note that, in this case, the optimized resampled set is bigger (703 instances) than the original non-resampled data set (506 instances). This is a clear example of when more instances are added than removed from the set. If only under-sampling is preferred instead of simultaneous under- and over-sampling, the appropriate under-sampling benchmark class would be provided as the value for the `benchmark` parameter. In this example, a CART regression decision tree is used as the evaluator for the resampled data sets (`DecisionTreeRegressor`), but any scikit-learn regressor could take its place.

4.2. Data Weighting

Some scikit-learn models can handle weighted instances. Class `EvoWeighting` from the module `data_weighting` optimizes weights of individual instances with the call of `reweight()` function. This function serves to find instance weights that can lead to better classification or regression results—it returns the array of the real numbers, which are weights of instances in the order of those instances in the given data set. The following code shows the basic example of reweighting and the resulting array of weights.

```
>>> from sklearn.datasets import load_breast_cancer
>>> from EvoPreprocess.data_weighting import EvoWeighting
>>>
>>> dataset = load_breast_cancer()
>>> instance_weights = EvoWeighting().reweight(dataset.data,
dataset.target)
>>> print(instance_weights)
[1.568983893273244 1.2899430717992133 ... 0.7248390003761751]
```

The following code shows the example of combining the data weight optimization with the scikit-learn classifier of the decision tree, which supports weighted instances. The accuracies of both classifiers, the one fitted with unweighted data and the one built with weighted data, are outputted.

```
>>> from sklearn.datasets import load_breast_cancer
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.tree import DecisionTreeClassifier
>>> from EvoPreprocess.data_weighting import EvoWeighting
>>>
>>> random_seed = 1234
>>> dataset = load_breast_cancer()
>>> X_train, X_test, y_train, y_test = train_test_split(
dataset.data, dataset.target,
test_size=0.33,
random_state=random_seed)
>>> cls = DecisionTreeClassifier(random_state=random_seed)
>>> cls.fit(X_train, y_train)
>>>
>>> print(X_train.shape,
accuracy_score(y_test, cls.predict(X_test)),
sep=':')
(381, 30): 0.8936170212765957

>>> instance_weights = EvoWeighting(random_seed=random_seed).reweight(X_train,
y_train)
>>> cls.fit(X_train, y_train, sample_weight=instance_weights)
>>> print(X_train.shape,
accuracy_score(y_test, cls.predict(X_test)),
sep=':')
(381, 30): 0.9042553191489362
```

The example shows that the number of instances stays 381 and the number of feature stays at 30, but the accuracy rises from 89.36% with unweighted data to 90.43% with weighted instances.

4.3. Feature Selection

Another task performed by the EvoPreprocess framework is feature selection. This is done with nature-inspired algorithms from the NiaPy framework, and can be used independently, or as one of the steps in the scikit-learn pipeline. The feature selection is used with the construction of the `EvoFeatureSelection` class from the module `feature_selection` and calling the function `fit_transform()`, which returns the new data set with only selected features.

```
>>> from sklearn.datasets import load_breast_cancer
>>> from EvoPreprocess.feature_selection import EvoFeatureSelection
>>>
>>> dataset = load_breast_cancer()
>>> print(dataset.data.shape)
(569, 30)

>>> X_new = EvoFeatureSelection().fit_transform(dataset.data,
dataset.target)
>>> print(X_new.shape)
(569, 17)
```

The results of the example demonstrate, that the original data set contains 30 features, and the modified data set after the feature selection contains only 17 features.

Again, numerous settings can be changed: the nature-inspired algorithm used for the optimization process, the classifier/regressor used for the evaluation, the number of folds for validation, the number of repeated runs for each fold, the number of parallel runs and the random seed. The following code shows the example of combining `EvoFeatureSelection` with the regressor from scikit-learn, and the evaluation of the quality of both approaches – the regressor built with the original data set, and the regressor built with the modified data set with only some features selected.

```
>>> from sklearn.datasets import load_boston
>>> from sklearn.metrics import mean_squared_error
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.tree import DecisionTreeRegressor
>>> from EvoPreprocess.feature_selection import EvoFeatureSelection
>>>
>>> random_seed = 654
>>> dataset = load_boston()
>>> X_train, X_test, y_train, y_test = train_test_split(
dataset.data,
dataset.target,
test_size=0.33,
random_state=random_seed)
>>> model = DecisionTreeRegressor(random_state=random_seed)
>>> model.fit(X_train, y_train)
>>> print(X_train.shape,
mean_squared_error(y_test, model.predict(X_test)),
sep=':')
(339, 13): 24.475748502994012

>>> evo = EvoFeatureSelection(evaluator=model, random_seed=random_seed)
>>> X_train_new = evo.fit_transform(X_train, y_train)
>>>
>>> model.fit(X_train_new, y_train)
>>> X_test_new = evo.transform(X_test)
>>> print(X_train_new.shape,
mean_squared_error(y_test, model.predict(X_test_new)),
sep=':')
(339, 6): 18.03443113772455
```

The results show that, using the new data set with only 6 features selected during the fitting of the decision tree regressor, outperforms the decision tree regressor built with the original data set with all of the 13 features—MSE of 18.03 for the regressor with feature selected data set vs. MSE of 24.48 for the regressor with the original data set.

4.4. Compatibility and Extendability

The compatibility with existing well-established data analysis machine learning libraries is one of the main features of EvoPreprocess. For this reason, all the modules accept the data in the form of extensively used NumPy array [78] and the pandas `DataFrame` [79]. The following examples demonstrate the further compatibility capacity of the EvoPreprocess with scikit-learn and imbalanced-learn Python machine learning packages.

The scikit-learn already includes various feature selection methods and supports their usage with the provided machine learning pipelines. `EvoFeatureSelection` extends scikit-learn's feature selection base class, and so it can be included in the pipeline, as the following code demonstrates.

```
>>> from sklearn.linear_model import LinearRegression
>>> from sklearn.pipeline import Pipeline
>>> from sklearn.datasets import load_boston
>>> from sklearn.metrics import mean_squared_error
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.tree import DecisionTreeRegressor
>>> from EvoPreprocess.feature_selection import EvoFeatureSelection
>>>
>>> random_seed = 987
>>> dataset = load_boston()
>>>
>>> X_train, X_test, y_train, y_test = train_test_split(
dataset.data,
dataset.target,
test_size=0.33,
random_state=random_seed)

>>> model = DecisionTreeRegressor(random_state=random_seed)
>>> model.fit(X_train, y_train)

>>> print(mean_squared_error(y_test, cls.predict(X_test)))
20.227544910179642

>>> pipeline = Pipeline(steps=[
('feature_selection', EvoFeatureSelection(
evaluator=LinearRegression(),
n_folds=4,
n_runs=8,
random_seed=random_seed)),
('regressor', DecisionTreeRegressor(random_state=random_seed))
])
>>> pipeline.fit(X_train, y_train)

>>> print(mean_squared_error(y_test, pipeline.predict(X_test)))
19.073532934131734
```

As the example demonstrates, the pipeline with `EvoFeatureSelection` builds a better regressor than the model without the feature selection, as the MSE is 19.07 vs. the original regressor's MSE of 20.23. The example also shows that one can choose different evaluators in any of the preprocessing tasks from the final classifier or regressor. Here, the linear regression is chosen as the evaluator, but the decision tree regressor is used in the final model fitting. Using a different, lightweight, evaluation model can be useful when fitting the decision models that can be computationally expensive.

All three EvoPreprocess tasks are compatible and can be used in imbalanced-learn pipelines [17], as is demonstrated in the following code. Note that both tasks in the example are parallelized and the number of simultaneous runs on different CPU cores is set with the `n_jobs` parameter (default value of `None` utilizes all cores available). The reproducibility of the results is guaranteed with setting the `random_seed` parameter.

```
>>> from sklearn.datasets import load_breast_cancer
>>> from sklearn.tree import DecisionTreeClassifier
>>> from imblearn.pipeline import Pipeline
>>> from EvoPreprocess.feature_selection import EvoFeatureSelection
>>> from EvoPreprocess.data_sampling import EvoSampling
>>> random_seed = 1111
>>> dataset = load_breast_cancer()
>>> X_train, X_test, y_train, y_test = train_test_split(
dataset.data,
dataset.target,
test_size=0.33,
random_state=random_seed)
>>>
>>> cls = DecisionTreeClassifier(random_state=random_seed)
>>> cls.fit(X_train, y_train)
>>> print(accuracy_score(y_test, cls.predict(X_test)))
0.8829787234042553

>>> pipeline = Pipeline(steps=[
('feature_selection', EvoFeatureSelection(n_folds=10,
random_seed=random_seed)),
('data_sampling', EvoSampling(n_folds=10,
random_seed=random_seed)),
('classifier', DecisionTreeClassifier(random_state=random_seed))])
])
>>> pipeline.fit(X_train, y_train)
>>> print(accuracy_score(y_test, pipeline.predict(X_test)))
0.9148936170212766
```

The results show that the pipeline with both feature selection and data sampling results in a superior classifier than the one without these preprocessing steps, as the accuracy of the pipeline is 91.49 vs. 88.30 for the classifier without preprocessing steps.

Using any of the EvoPreprocess tasks can be further optimized with the custom settings of the optimizer (the nature-inspired algorithm used to optimize the task). This can be done with the parameter `optimizer_settings`, which is in the form of a Python dictionary. The code listing below shows one such example, where the bat algorithm [80] is used for the optimization process, and samples the new data set with only 335 instances instead of the original 569. One should refer to the NiaPy package for the available nature-inspired optimization methods and their settings.

```
>>> from sklearn.datasets import load_breast_cancer
>>> from EvoPreprocess.data_sampling import EvoSampling
>>> import NiaPy.algorithms.basic as nia
>>>
>>> dataset = load_breast_cancer()
>>> print(dataset.data.shape, len(dataset.target))
[569,30] 569

>>> settings = {'NP': 1000, 'A': 0.5, 'r': 0.5, 'Qmin': 0.0, 'Qmax': 2.0}
>>> X_resampled, y_resampled = EvoSampling(optimizer=nia.Bat algorithm,
optimizer_settings=settings
).fit_resample(dataset.data,
dataset.target)
>>> print(X_resampled.shape, len(y_resampled))
(335, 30) 335
```

As EvoPreprocess uses any NiaPy compatible continuous optimization algorithm, one can implement their own, or customize and extend the existing ones. The customization is also possible on the evaluation functions (i.e., favoring or limiting to under-sampling with more emphasis on a smaller data set than on the quality of the model). This can be done with the extension or replacement of benchmark classes (`SamplingBenchmark`, `FeatureSelectionBenchmark` and `WeightingBenchmark`). The following code listing shows the usage of the custom optimizer (random search) and custom benchmark function for data sampling (a mixture of both model quality and size of the data set).

```
>>> import numpy as np
>>> from NiaPy.algorithms import Algorithm
>>> from numpy import apply_along_axis, math
>>> from sklearn.datasets import load_breast_cancer
>>> from sklearn.utils import safe_indexing
>>> from EvoPreprocess.data_sampling import EvoSampling
>>> from EvoPreprocess.data_sampling.SamplingBenchmark import SamplingBenchmark
>>>
>>> class RandomSearch(Algorithm):
Name = ['RandomSearch', 'RS']

def runIteration(self, task, pop, fpop, xb, fxb, **dparams):
pop = task.Lower + self.Rand.rand(self.NP, task.D) * task.bRange
fpop = apply_along_axis(task.eval, 1, pop)
return pop, fpop, {}
>>>
>>> class CustomSamplingBenchmark(SamplingBenchmark):
# ________________0___1_____2______3_______4___
mapping = np.array([0.5, 0.75, 0.875, 0.9375, 1])

def function(self):
def evaluate(D, sol):
phenotype = SamplingBenchmark.map_to_phenotype(
CustomSamplingBenchmark.to_phenotype(sol))
X_sampled = safe_indexing(self.X_train, phenotype)
y_sampled = safe_indexing(self.y_train, phenotype)

if X_sampled.shape[0] > 0:
cls = self.evaluator.fit(X_sampled, y_sampled)
y_predicted = cls.predict(self.X_valid)
quality = accuracy_score(self.y_valid, y_predicted)
size_percentage = len(y_sampled) / len(sol)

return (1 - quality) * size_percentage
else:
return math.inf

return evaluate

@staticmethod
def to_phenotype(genotype):
return np.digitize(genotype[:-5], CustomSamplingBenchmark.mapping)
>>>
>>> dataset = load_breast_cancer()
>>> print(dataset.data.shape, len(dataset.target))
(569, 30) 569

>>> X_resampled, y_resampled = EvoSampling(optimizer=RandomSearch,
benchmark=CustomSamplingBenchmark
).fit_resample(dataset.data, dataset.target)
>>> print(X_resampled.shape, len(y_resampled))
(311, 30) 311
```

5. Experiments

In this section, the results of the experiments of all three data preprocessing tasks with EvoPreprocess framework are presented. The results of the EvoPreprocess framework are compared to other publicly available preprocessing approaches in Python programming language. The classification data set *Libras Movement* [81] was selected, as is the prototype of heavily imbalanced data set with many features. Thus, data sampling, feature selection and data weighting could all be used. The data set contains 360 instances of two classes with imbalance ration 14:1 (334 vs. 24 instances) and 90 continuous features. All tests were run with five-fold cross-validation. The classifier used was scikit-learn implementation of CART decision tree `DecisionTreeClassifier` with default setting. The settings for all EvoPreprocess experiments were the following:

- 2 folds for internal cross-validation,
- 10 independent runs of meta-heuristic algorithm on each fold,
- `Genetic algorithm` meta-heuristic optimizer from NiaPy with the population size of 200, with 0.8 chance of crossover, 0.5 chance of mutation and 20,000 total evaluations,
- `DecisionTreeClassifier` for the evaluation of the solutions (denoted as CART),
- random seed was set to 1111 for all algorithms, and
- all other settings were left at their default values.

The results of weighting the data set instances are shown in Table 1. Classification of the weighted data set (denoted as EvoPreprocess) produced better results, in terms of overall accuracy and F-score, than the classification of the original non-weighted set. This was evident by looking at the average and median score as the average rank (the smaller ranks are preferred). There are no other weighting approaches available in either scikit-learn or imbalanced-learn packages, so the results of EvoPreprocess weighting was compared to non-weighted instances.

Table 1. Accuracy and F-score results of weighting. The best values are bolded.

	CART		**EvoPreprocess**	
Fold	**Acc**	**Fsc**	**Acc**	**Fsc**
1	83.56	33.33	**87.67**	**47.06**
2	90.28	36.36	**95.83**	**57.14**
3	88.89	20.00	**91.67**	**57.14**
4	**91.67**	25.00	88.89	**42.86**
5	85.92	0.00	**88.73**	**0.00**
Average	88.06	22.94	**90.56**	**40.84**
Median	**88.89**	25.00	**88.89**	**47.06**
Average rank	1.8	1.8	**1.2**	**1.0**

Next, the feature selection EvoPreprocess method was experimented with. All of the scikit-learn feature selection methods were included in this comparison: `chi2`, `ANOVA F` and `Mutual information`. Also, one additional implementation of feature election with genetic algorithm was included in the experiment—`sklearn-genetic` [82]. This implementation was included in the experiment as it is freely available for use when applying the genetic algorithm to feature selection problems and it is compatible with scikit-learn framework (aligning it with of EvoPreprocess). The settings of `sklearn-genetic` were left to their default values, as with EvoPreprocess, to prevent over-fitting to the provided data set.

The results in Table 2 show that, again, in both classification metrics, the classification results were superior when feature selection with EvoPreprocess was used in comparisons to other available methods. Final classification with features selected by EvoPreprocess resulted in the best accuracy and F-score (according to average, median and average rank) Even when compared to the already existing implementation of feature selection with genetic algorithm `sklearn-genetic`, it is evident, that EvoPreprocess framework results in data sets which tend to perform better in classification. Even if

this superiority of EvoPreprocess over `sklearn-genetic` is as a result of chance, due to No Free Lunch Theorem, `sklearn-genetic` does not provide self-adaptation and parallelization as is readily available with EvoPreprocess. Furthermore, `sklearn-genetic` provides feature selection only with genetic algorithms, while EvoPreprocess provides preprocessing approaches with many more nature-inspired algorithms and is easily extendable with any future new methods.

Table 2. Accuracy and F-score results of feature selection. The best values are bolded.

	CART		**chi2**		**ANOVA F**		**Mutual Information**		**Sklearn Genetic**		**EvoPreprocess**	
Fold	**Acc**	**Fsc**	**Acc**	**Fsc**	**Acc**	**Fsc**	**Acc**	**Fsc**	**Acc**	**Fsc**	**Acc**	**Fsc**
1	84.93	35.29	97.26	80.00	**98.63**	**88.89**	90.41	53.33	82.19	31.58	97.26	80.00
2	83.33	33.33	94.44	33.33	94.44	33.33	**95.83**	**57.14**	94.44	60.00	**95.83**	**57.14**
3	93.06	28.57	91.67	0.00	87.50	0.00	86.11	16.67	91.67	25.00	**95.83**	**57.14**
4	86.11	0.00	93.06	**61.54**	91.67	40.00	93.06	54.55	91.67	50.00	**94.44**	50.00
5	81.69	0.00	83.10	14.29	**90.14**	0.00	84.51	**35.29**	87.32	0.00	**90.14**	0.00
Average	85.82	19.44	91.91	37.83	92.48	32.44	89.98	43.40	89.46	33.32	**94.70**	**48.86**
Median	84.93	28.57	93.06	33.33	91.67	33.33	90.41	53.33	90.56	32.45	**95.83**	**57.14**
Avg. rank	5.0	4.0	3.0	2.8	2.8	3.6	3.4	2.6	3.8	3.2	**1.2**	**2.2**

Lastly, the experiment with data sampling was conducted. Here, several imbalanced-learn implementation for under- and over-sampling were included: `under-sampling with Tomek Links`, `over-sampling with SMOTE`, and `over- and under-sampling with SMOTE and Tomek Links`. The results in Table 3 show the mixed performance of EvoPreprocess for data sampling. In general, both over-sampling methods (SMOTE and SMOTE with Tomek Links) performed better than EvoPreprocess. SMOTE over-sampling does not over-sample instance with duplication but constructs synthetic new instances. Regular over-sampling with duplication can only go so far, while the creation of new knowledge with new instances can further diversify the data set and prevent over-fitting of the model. Also, the implementation of data sampling is such that the search space is positively correlated with the number of instances in the original data set. Here, the data set contains 558 instances, which means, the length of the genotype is at least as large.

Table 3. Accuracy and F-score results of data sampling (under- and over- sampling). The best values are bolded.

	CART		**Under-Sampling Tomek Links**		**Over-Sampling SMOTE**		**Over-under- Sampling SMOTE with Tomek**		**EvoPreprocess**	
Fold	**Acc**	**Fsc**	**Acc**	**Fsc**	**Acc**	**Fsc**	**Acc**	**Fsc**	**Acc**	**Fsc**
1	84.93	35.29	84.93	35.29	**94.52**	**66.67**	**94.52**	**66.67**	84.93	0.00
2	83.33	33.33	83.33	33.33	**97.22**	**75.00**	**97.22**	**75.00**	94.44	33.33
3	93.06	28.57	93.06	28.57	88.89	**33.33**	88.89	**33.33**	**94.44**	**33.33**
4	86.11	0.00	86.11	0.00	**93.06**	**54.55**	**93.06**	**54.55**	87.50	40.00
5	81.69	0.00	81.69	0.00	92.96	**61.54**	92.96	**61.54**	**94.37**	0.00
Average	85.82	19.44	85.82	19.44	**93.33**	**58.22**	**93.33**	**58.22**	91.14	21.33
Median	84.93	28.57	84.93	28.57	93.06	**61.54**	93.06	**61.54**	**94.37**	33.33
Avg. rank	3.4	3.4	3.4	3.4	**1.8**	**1.0**	**1.8**	**1.0**	2.2	3

It was shown [83] that larger search spaces tend to be harder to solve for meta-heuristic algorithms, making them stuck in the local optima. Numerous techniques tackle large problem solving for meta-heuristic and nature-inspired methods [84–86], but the scope of this paper was to demonstrate the usability of the EvoPreprocess framework in general. The framework could easily be extended and modified with any large-scale mitigation approaches. On the other hand, the data sampling with EvoPreprocess still outperforms the included under-sampling with Tomek Links by a great margin.

As the results of the experiment demonstrate, the EvoPreprocess framework is a viable and competitive alternative to already existing data preprocessing approaches. Even though it did not provide the best results in all cases on the one data set used in the experiment, its effectiveness is

not questionable. Because of the No Free Lunch Theorem, there is no perfect setting for the used methods that would perform the best in all of the tasks and different data sets. Naturally, with the right parameter setting, all of the methods used in the experiment would probably perform in better. Tuning the EvoPreprocess setting parameters would change the results, but the optimization to the one used data set is not the goal of this paper. This experiment serves to demonstrate the straightforward use and general effectiveness of the framework.

Comparison of Nature-Inspired Algorithms

In this section, the results of different nature-inspired algorithms in their ability to preprocess data with feature selection, data sampling and data weighting, are compared. A synthetic classification data set was constructed, which contained 400 instances, 10 features and 2 overlapping classes. Out of all 10 features, 3 of them were informative, 3 were redundant of informative features, 2 were transformed copies of informative and redundant features and 2 were random noise. The experiment was done with 5-fold cross-validation and the final results were averaged across all folds. The overall accuracy and F-score were measured, as well as computation time of nature-inspired algorithms. All three preprocessing tasks were applied individually before the preprocessed data was used in training of the CART classification decision tree. The experiment was done on a computer with Intel i5-6500 CPU @ 3.2 GHz with four cores, 32GB of RAM and Windows 10.

The nature-inspired algorithms used in this experiment were the following: *Artificial cee colony*, *Bat algorithm*, *Cuckoo search*, *Differential evolution*, *Evolution strategy* $(\mu + \lambda)$, *Genetic algorithm*, *Harmony search*, and *Particle swarm optimization*. The settings of the optimization algorithms were left at their default values, only the size of the solution set (individual solutions in one generation/iteration) was set to 100, and the maximal number of fitness evaluations was set to 100,000.

Table 4. Comparison of different nature-inspired algorithms on three data preprocessing task. Classification accuracy, F-score and computation time (in seconds) are presented. The best results are bolded.

Task	Algorithm	Acc	Fsc	Computation Time (s) with Four Parallel Runs	Computation Time (s) with no Parallel Runs
Feature selection	Artificial bee colony	86.24	66.02	15.0062	42.1964
	Bat algorithm	85.24	64.12	15.2828	47.6856
	Cuckoo search	84.49	60.80	**13.8600**	**35.9927**
	Differential evolution	86.52	66.98	16.2162	61.5314
	Evolution strategy	**87.01**	**68.72**	15.2992	50.4924
	Genetic algorithm	86.51	67.50	15.0095	45.0393
	Harmony search	85.76	65.99	15.9159	50.2762
	Particle swarm optimization	85.25	63.66	14.8623	52.1369
Data sampling	Artificial bee colony	85.50	64.27	19.4990	61.4075
	Bat algorithm	**87.01**	**67.85**	19.0993	58.1645
	Cuckoo search	85.50	63.95	18.9420	53.1541
	Differential evolution	86.25	65.33	23.4585	67.7465
	Evolution strategy	84.00	61.56	19.2668	60.3277
	Genetic algorithm	85.02	62.89	24.8603	73.1325
	Harmony search	86.01	65.59	39.1134	105.1702
	Particle swarm optimization	85.25	64.30	**18.7993**	**52.8433**
Data weighting	Artificial bee colony	87.76	**69.98**	16.7115	43.9391
	Bat algorithm	86.25	66.47	**14.7816**	**37.3560**
	Cuckoo search	86.74	67.55	15.9384	42.0680
	Differential evolution	85.52	65.64	18.8684	51.4843
	Evolution strategy	86.00	67.16	15.6373	72.5965
	Genetic algorithm	86.24	64.35	22.0804	62.2260
	Harmony search	87.01	67.04	37.2622	109.2218
	Particle swarm optimization	**88.00**	68.76	15.5451	40.0671

The results of classification metrics and computation times for all preprocessing tasks are in Table 4 and show no clear winner. Evolution strategy is the best algorithm in feature selection (acc = 87.01%, F-score = 68.72%), and Bat algorithm is the best in data sampling (acc = 87.01%,

F-score = 67.85%). Artificial bee colony (acc = 87.76%, F-score = 69.98%) and Particle swarm optimization (acc = 88%, F-score = 68.76%) perform best in data weighting. Again, this is due to No Free Lunch Theorem so there is no setting or algorithm that could perform the best in all situations (preprocessing tasks or data sets).

Computation times are more consistent for different algorithms. Harmony search is clearly among the slowest optimization algorithms in all three tasks, while Particle swarm optimization, Cuckoo search and Bat algorithm rank among the fastest one in all three tasks. Table 4 also shows the computation times of the same algorithms when no parallelization is used. The computation time advantage of the parallelized implementation of the framework is evident in every nature-inspired algorithm in every data preprocessing tasks. The parallelization is done by the framework and is optimization algorithm independent, which makes custom implementations of optimization algorithms straightforward. The speed differences are due to the differences in the operators (see Section 2.3) and implementation differences (i.e., vectorized operators tend to compute faster, than non-vectorized ones).

6. Conclusions

This paper presents the EvoPreprocess framework for preprocessing data with nature-inspired optimization algorithms for three different tasks: data sampling, data weighting, and feature selection. The architectural structure and the documentation of the application programming interface of the framework are overviewed in detail. The EvoPreprocess framework offers a simple object-oriented implementation, which is easily extendable with custom metrics and custom optimization methods. The presented framework is compatible with the established Python data mining libraries pandas, scikit-learn and imbalanced-learn.

The framework effectiveness is demonstrated in the provided results of the experiment, where it was compared to the well-established data preprocessing packages. As the results suggest, the EvoPreprocess framework already provides competitive results in all preprocessing tasks (in some more than in others) in its current form. However, its true potential is in the customization ability with other variants of data preprocessing tasks.

In the future, unit tests are planned to be included in the framework modules, which could make customization of the provided tasks and classes easier. There are numerous preprocessing tasks which could still benefit from the easy to use package (feature extraction, missing value imputation, data normalization, etc.). Ultimately, the framework may serve the community by providing a simple and customizable tool for use in practical applications and future research endeavors.

Funding: This research was funded by the Slovenian Research Agency (research core funding No. P2-0057).

Conflicts of Interest: The author declares no conflict of interest.

References

1. García, S.; Luengo, J.; Herrera, F. *Data Preprocessing in Data Mining*; Springer: Berlin/Heidelberg, Germany, 2015.
2. Japkowicz, N.; Stephen, S. The class imbalance problem: A systematic study. *Intell. Data Anal.* **2002**, *6*, 429–449. [CrossRef]
3. García, V.; Sánchez, J.S.; Mollineda, R.A. On the effectiveness of preprocessing methods when dealing with different levels of class imbalance. *Knowl.-Based Syst.* **2012**, *25*, 13–21. [CrossRef]
4. Kotsiantis, S.; Kanellopoulos, D.; Pintelas, P. Data preprocessing for supervised leaning. *Int. J. Comput. Sci.* **2006**, *1*, 111–117.
5. Li, J.; Cheng, K.; Wang, S.; Morstatter, F.; Trevino, R.P.; Tang, J.; Liu, H. Feature selection: A data perspective. *ACM Comput. Surv. (CSUR)* **2018**, *50*, 94. [CrossRef]
6. Guyon, I.; Elisseeff, A. An introduction to variable and feature selection. *J. Mach. Learn. Res.* **2003**, *3*, 1157–1182.

7. Laurikkala, J. Improving Identification of Difficult Small Classes by Balancing Class Distribution. In Proceedings of the 8th Conference on AI in Medicine in Europe: Artificial Intelligence Medicine, Cascais, Portugal, 1–4 July 2001; Springer: London, UK, 2001; pp. 63–66.
8. Liu, H.; Motoda, H. *Instance Selection and Construction for Data Mining*; Springer Science & Business Media: Berlin/Heidelberg, Germany, 2013; Volume 608.
9. Chawla, N.V.; Bowyer, K.W.; Hall, L.O.; Kegelmeyer, W.P. SMOTE: Synthetic minority over-sampling technique. *J. Artif. Intell. Res.* **2002**, *16*, 321–357. [CrossRef]
10. Fernández, A.; Garcia, S.; Herrera, F.; Chawla, N.V. SMOTE for learning from imbalanced data: Progress and challenges, marking the 15-year anniversary. *J. Artif. Intell. Res.* **2018**, *61*, 863–905. [CrossRef]
11. Diao, R.; Shen, Q. Nature inspired feature selection meta-heuristics. *Artif. Intell. Rev.* **2015**, *44*, 311–340. [CrossRef]
12. Galar, M.; Fernández, A.; Barrenechea, E.; Herrera, F. EUSBoost: Enhancing ensembles for highly imbalanced data-sets by evolutionary undersampling. *Pattern Recognit.* **2013**, *46*, 3460–3471. [CrossRef]
13. Sayed, S.; Nassef, M.; Badr, A.; Farag, I. A nested genetic algorithm for feature selection in high-dimensional cancer microarray datasets. *Expert Syst. Appl.* **2019**, *121*, 233–243. [CrossRef]
14. Ghosh, M.; Adhikary, S.; Ghosh, K.K.; Sardar, A.; Begum, S.; Sarkar, R. Genetic algorithm based cancerous gene identification from microarray data using ensemble of filter methods. *Med. Biol. Eng. Comput.* **2019**, *57*, 159–176. [CrossRef] [PubMed]
15. Rao, K.N.; Reddy, C.S. A novel under sampling strategy for efficient software defect analysis of skewed distributed data. *Evol. Syst.* **2019**, *11*, 119–131. [CrossRef]
16. Subudhi, S.; Patro, R.N.; Biswal, P.K. Pso-based synthetic minority oversampling technique for classification of reduced hyperspectral image. In *Soft Computing for Problem Solving*; Springer: Berlin/Heidelberg, Germany, 2019; pp. 617–625.
17. Lemaître, G.; Nogueira, F.; Aridas, C.K. Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning. *J. Mach. Learn. Res.* **2017**, *18*, 559–563.
18. Kursa, M.B.; Rudnicki, W.R. Feature selection with the Boruta package. *J. Stat. Softw.* **2010**, *36*, 1–13. [CrossRef]
19. Lagani, V.; Athineou, G.; Farcomeni, A.; Tsagris, M.; Tsamardinos, I. Feature selection with the r package mxm: Discovering statistically-equivalent feature subsets. *arXiv* **2016**, arXiv:1611.03227.
20. Scrucca, L.; Raftery, A.E. clustvarsel: A Package Implementing Variable Selection for Gaussian Model-based Clustering in R. *J. Stat. Softw.* **2018**, *84*. [CrossRef]
21. Dramiński, M.; Koronacki, J. rmcfs: An R Package for Monte Carlo Feature Selection and Interdependency Discovery. *J. Stat. Softw.* **2018**, *85*, 1–28. [CrossRef]
22. Lunardon, N.; Menardi, G.; Torelli, N. ROSE: A Package for Binary Imbalanced Learning. *R J.* **2014**, *6*, 79–89. [CrossRef]
23. Liu, H.; Motoda, H. *Computational Methods of Feature Selection*; CRC Press: Boca Raton, FL, USA, 2007.
24. Xue, B.; Zhang, M.; Browne, W.N.; Yao, X. A survey on evolutionary computation approaches to feature selection. *IEEE Trans. Evol. Comput.* **2015**, *20*, 606–626. [CrossRef]
25. Brezočnik, L.; Fister, I.; Podgorelec, V. Swarm intelligence algorithms for feature selection: A review. *Appl. Sci.* **2018**, *8*, 1521. [CrossRef]
26. Mafarja, M.M.; Mirjalili, S. Hybrid whale optimization algorithm with simulated annealing for feature selection. *Neurocomputing* **2017**, *260*, 302–312. [CrossRef]
27. Mafarja, M.; Aljarah, I.; Heidari, A.A.; Faris, H.; Fournier-Viger, P.; Li, X.; Mirjalili, S. Binary dragonfly optimization for feature selection using time-varying transfer functions. *Knowl.-Based Syst.* **2018**, *161*, 185–204. [CrossRef]
28. Sayed, G.I.; Tharwat, A.; Hassanien, A.E. Chaotic dragonfly algorithm: An improved metaheuristic algorithm for feature selection. *Appl. Intell.* **2019**, *49*, 188–205. [CrossRef]
29. Aljarah, I.; Ala'M, A.Z.; Faris, H.; Hassonah, M.A.; Mirjalili, S.; Saadeh, H. Simultaneous feature selection and support vector machine optimization using the grasshopper optimization algorithm. *Cogn. Comput.* **2018**, *10*, 478–495. [CrossRef]
30. Abdel-Basset, M.; El-Shahat, D.; El-henawy, I.; de Albuquerque, V.H.C.; Mirjalili, S. A new fusion of grey wolf optimizer algorithm with a two-phase mutation for feature selection. *Expert Syst. Appl.* **2020**, *139*, 112824. [CrossRef]

31. Al-Tashi, Q.; Kadir, S.J.A.; Rais, H.M.; Mirjalili, S.; Alhussian, H. Binary Optimization Using Hybrid Grey Wolf Optimization for Feature Selection. *IEEE Access* **2019**, *7*, 39496–39508. [CrossRef]
32. Zorarpacı, E.; Özel, S.A. A hybrid approach of differential evolution and artificial bee colony for feature selection. *Expert Syst. Appl.* **2016**, *62*, 91–103. [CrossRef]
33. Sayed, G.I.; Hassanien, A.E.; Azar, A.T. Feature selection via a novel chaotic crow search algorithm. *Neural Comput. Appl.* **2019**, *31*, 171–188. [CrossRef]
34. Gu, S.; Cheng, R.; Jin, Y. Feature selection for high-dimensional classification using a competitive swarm optimizer. *Soft Comput.* **2018**, *22*, 811–822. [CrossRef]
35. Dong, H.; Li, T.; Ding, R.; Sun, J. A novel hybrid genetic algorithm with granular information for feature selection and optimization. *Appl. Soft Comput.* **2018**, *65*, 33–46. [CrossRef]
36. Ali, A.; Shamsuddin, S.M.; Ralescu, A.L. Classification with class imbalance problem: A review. *Int. J. Adv. Soft. Comput. Appl.* **2015**, *7*, 176–204.
37. Dragusin, R.; Petcu, P.; Lioma, C.; Larsen, B.; Jørgensen, H.; Winther, O. Rare disease diagnosis as an information retrieval task. In Proceedings of the Conference on the Theory of Information Retrieval, Bertinoro, Italy, 12–14 September 2011; pp. 356–359.
38. Griggs, R.C.; Batshaw, M.; Dunkle, M.; Gopal-Srivastava, R.; Kaye, E.; Krischer, J.; Nguyen, T.; Paulus, K.; Merkel, P.A. Clinical research for rare disease: Opportunities, challenges, and solutions. *Mol. Genet. Metab.* **2009**, *96*, 20–26. [CrossRef]
39. Weigold, A.; Weigold, I.K.; Russell, E.J. Examination of the equivalence of self-report survey-based paper-and-pencil and internet data collection methods. *Psychol. Methods* **2013**, *18*, 53. [CrossRef]
40. Etikan, I.; Musa, S.A.; Alkassim, R.S. Comparison of convenience sampling and purposive sampling. *Am. J. Theor. Appl. Stat.* **2016**, *5*, 1–4. [CrossRef]
41. Haixiang, G.; Yijing, L.; Shang, J.; Mingyun, G.; Yuanyue, H.; Bing, G. Learning from class-imbalanced data: Review of methods and applications. *Expert Syst. Appl.* **2017**, *73*, 220–239. [CrossRef]
42. Triguero, I.; Galar, M.; Vluymans, S.; Cornelis, C.; Bustince, H.; Herrera, F.; Saeys, Y. Evolutionary undersampling for imbalanced big data classification. In Proceedings of the 2015 IEEE Congress on Evolutionary Computation (CEC), Sendai, Japan, 25–28 May 2015; pp. 715–722.
43. Fernandes, E.; de Leon Ferreira, A.C.P.; Carvalho, D.; Yao, X. Ensemble of Classifiers based on MultiObjective Genetic Sampling for Imbalanced Data. *IEEE Trans. Knowl. Data Eng.* **2019**, *32*, 1104–1115. [CrossRef]
44. Ha, J.; Lee, J.S. A new under-sampling method using genetic algorithm for imbalanced data classification. In Proceedings of the 10th International Conference on Ubiquitous Information Management and Communication, DaNang, Vietnam, 4–6 January 2016; p. 95.
45. Zhang, L.; Zhang, D. Evolutionary cost-sensitive extreme learning machine. *IEEE Trans. Neural Netw. Learn. Syst.* **2016**, *28*, 3045–3060. [CrossRef]
46. Elkan, C. The foundations of cost-sensitive learning. In Proceedings of the International Joint Conference on Artificial Intelligence; Seattle, WA, USA, 4–10 August 2001; Volume 17, pp. 973–978.
47. Yang, X.S. *Nature-Inspired Optimization Algorithms*; Elsevier: Amsterdam, The Netherlands, 2014.
48. Fister, I., Jr.; Yang, X.S.; Fister, I.; Brest, J.; Fister, D. A brief review of nature-inspired algorithms for optimization. *arXiv* **2013**,arXiv:1307.4186.
49. Yang, X.S.; Cui, Z.; Xiao, R.; Gandomi, A.H.; Karamanoglu, M. *Swarm Intelligence and Bio-Inspired Computation: Theory and Applications*; Newnes: Newton, MA, USA, 2013.
50. Pardalos, P.M.; Prokopyev, O.A.; Busygin, S. Continuous approaches for solving discrete optimization problems. In *Handbook on Modelling for Discrete Optimization*; Springer: Berlin/Heidelberg, Germany, 2006; pp. 39–60.
51. Fister, D.; Fister, I.; Jagrič, T.; Brest, J. Wrapper-Based Feature Selection Using Self-adaptive Differential Evolution. In *Swarm, Evolutionary, and Memetic Computing and Fuzzy and Neural Computing*; Springer: Berlin/Heidelberg, Germany, 2019; pp. 135–154.
52. Ghosh, A.; Datta, A.; Ghosh, S. Self-adaptive differential evolution for feature selection in hyperspectral image data. *Appl. Soft Comput.* **2013**, *13*, 1969–1977. [CrossRef]
53. Tao, X.; Li, Q.; Guo, W.; Ren, C.; Li, C.; Liu, R.; Zou, J. Self-adaptive cost weights-based support vector machine cost-sensitive ensemble for imbalanced data classification. *Inf. Sci.* **2019**, *487*, 31–56. [CrossRef]

54. Brest, J.; Greiner, S.; Boskovic, B.; Mernik, M.; Zumer, V. Self-adapting control parameters in differential evolution: A comparative study on numerical benchmark problems. *IEEE Trans. Evol. Comput.* **2006**, *10*, 646–657. [CrossRef]
55. Zainudin, M.; Sulaiman, M.; Mustapha, N.; Perumal, T.; Nazri, A.; Mohamed, R.; Manaf, S. Feature selection optimization using hybrid relief-f with self-adaptive differential evolution. *Int. J. Intell. Eng. Syst.* **2017**, *10*, 21–29. [CrossRef]
56. Xue, Y.; Xue, B.; Zhang, M. Self-adaptive particle swarm optimization for large-scale feature selection in classification. *ACM Trans. Knowl. Discov. Data (TKDD)* **2019**, *13*, 1–27. [CrossRef]
57. Fister, D.; Fister, I.; Jagrič, T.; Brest, J. A novel self-adaptive differential evolution for feature selection using threshold mechanism. In Proceedings of the 2018 IEEE Symposium Series on Computational Intelligence (SSCI), Bangalore, India, 18–21 November 2018; pp. 17–24.
58. Mafarja, M.; Mirjalili, S. Whale optimization approaches for wrapper feature selection. *Appl. Soft Comput.* **2018**, *62*, 441–453. [CrossRef]
59. Soufan, O.; Kleftogiannis, D.; Kalnis, P.; Bajic, V.B. DWFS: A wrapper feature selection tool based on a parallel genetic algorithm. *PLoS ONE* **2015**, *10*, e0117988. [CrossRef]
60. Mafarja, M.; Eleyan, D.; Abdullah, S.; Mirjalili, S. S-shaped vs. V-shaped transfer functions for ant lion optimization algorithm in feature selection problem. In Proceedings of the international conference on future networks and distributed systems, Cambridge, UK, 19–20 July 2017; p. 21.
61. Ghareb, A.S.; Bakar, A.A.; Hamdan, A.R. Hybrid feature selection based on enhanced genetic algorithm for text categorization. *Expert Syst. Appl.* **2016**, *49*, 31–47. [CrossRef]
62. Lones, M.A. Mitigating metaphors: A comprehensible guide to recent nature-inspired algorithms. *SN Comput. Sci.* **2020**, *1*, 49. [CrossRef]
63. Połap, D. Polar bear optimization algorithm: Meta-heuristic with fast population movement and dynamic birth and death mechanism. *Symmetry* **2017**, *9*, 203. [CrossRef]
64. Kazikova A.; Pluhacek M.; Senkerik R.; Viktorin, A. Proposal of a New Swarm Optimization Method Inspired in Bison Behavior. *Recent Adv. Soft. Comput.* **2019**, 146–156. [CrossRef]
65. Arora, S.; Singh, S. Butterfly optimization algorithm: A novel approach for global optimization. *Soft. Comput.* **2019**, *23*, 715–734. [CrossRef]
66. Klein, C.E.; Mariani, V.C.; dos Santos Coelho, L. *Cheetah Based Optimization Algorithm: A Novel Swarm Intelligence Paradigm*; ESANN: Bruges, Belgium, 2018.
67. Pierezan, J.; Coelho, L.D.S. Coyote optimization algorithm: A new metaheuristic for global optimization problems. In Proceedings of the 2018 IEEE Congress on Evolutionary Computation (CEC), Rio de Janeiro, Brazil, 8–13 July 2018; pp. 1–8.
68. Jain, M.; Singh, V.; Rani, A. A novel nature-inspired algorithm for optimization: Squirrel search algorithm. *Swarm Evol. Comput.* **2019**, *44*, 148–175. [CrossRef]
69. Sampson, J.R. *Adaptation in Natural and Artificial Systems*; Holland, J.H., Ed.; The MIT Press: Cambridge, MA, USA, 1976.
70. Storn, R.; Price, K. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *J. Glob. Optim.* **1997**, *11*, 341–359. [CrossRef]
71. Beyer, H.G.; Schwefel, H.P. Evolution strategies—A comprehensive introduction. *Nat. Comput.* **2002**, *1*, 3–52. [CrossRef]
72. Yang, X.S. Harmony search as a metaheuristic algorithm. In *Music-Inspired Harmony Search Algorithm*; Springer: Berlin/Heidelberg, Germany, 2009; pp. 1–14.
73. Kennedy, J.; Eberhart, R. Particle swarm optimization. In Proceedings of ICNN'95-International Conference on Neural Networks, Perth, Australia, 27 November–1 December 11995; Volume 4, pp. 1942–1948.
74. Karaboga, D.; Basturk, B. A powerful and efficient algorithm for numerical function optimization: Artificial bee colony (ABC) algorithm. *J. Glob. Optim.* **2007**, *39*, 459–471. [CrossRef]
75. Yang, X.S.; Gandomi, A.H. Bat algorithm: A novel approach for global engineering optimization. *Eng. Comput.* **2012**. [CrossRef]
76. Yang, X.S.; Deb, S. Cuckoo search via Lévy flights. In Proceedings of the 2009 World congress on nature & biologically inspired computing (NaBIC), Coimbatore, India, 9–11 December 2009; pp. 210–214.
77. Friedman, J.; Hastie, T.; Tibshirani, R. *The Elements of Statistical Learning*; Springer: New York, NY, USA, 2001; Volume 1.

78. Oliphant, T. *NumPy: A guide to NumPy*; Trelgol Publishing: Spanish Fork, UT, USA, 2006.
79. McKinney, W. Data Structures for Statistical Computing in Python. In Proceedings of the 9th Python in Science Conference, Austin, TX, USA, 28 June–3 July 2010; pp. 51–56.
80. Yang, X.S. A new metaheuristic bat-inspired algorithm. In *Nature inspired cooperative strategies for optimization (NICSO 2010)*; Springer: Berlin/Heidelberg, Germany, 2010; pp. 65–74.
81. Dias, D.B.; Madeo, R.C.; Rocha, T.; Bíscaro, H.H.; Peres, S.M. Hand movement recognition for brazilian sign language: A study using distance-based neural networks. In Proceedings of the 2009 International Joint Conference on Neural Networks, Atlanta, GA, USA, 14–19 June 2009; pp. 697–704.
82. Calzolari, M. *Manuel-Calzolari/Sklearn-Genetic: Sklearn-Genetic 0.2*; Zenodo: Geneva, Switzerland, 2019. [CrossRef]
83. Reeves, C.R. Landscapes, operators and heuristic search. *Ann. Oper. Res.* **1999**, *86*, 473–490. [CrossRef]
84. Yang, Z.; Tang, K.; Yao, X. Large scale evolutionary optimization using cooperative coevolution. *Inf. Sci.* **2008**, *178*, 2985–2999. [CrossRef]
85. Zhang, H.; Ishikawa, M. An extended hybrid genetic algorithm for exploring a large search space. In Proceedings of the 2nd International Conference on Autonomous Robots and Agents, Kyoto, Japan, 10–11 December 2004; pp. 244–248.
86. Siedlecki, W.; Sklansky, J. A note on genetic algorithms for large-scale feature selection. In *Handbook of Pattern Recognition and Computer Vision*; World Scientific: London, UK, 1993; pp. 88–107.