NOVA IMS Information Management School

MAA

Mestrado em Métodos Analíticos Avançados
Master Program in Advanced Analytics

OVERSAMPLING FOR IMBALANCED LEARNING BASED ON K-MEANS AND SMOTE

Felix Last

Dissertation presented as partial requirement for obtaining the Master's degree in Advanced Analytics

NOVA Information Management School
Instituto Superior de Estatística e Gestão de Informação
Universidade Nova de Lisboa

NOVA Information Management School
Instituto Superior de Estatística e Gestão de Informação
Universidade Nova de Lisboa

OVERSAMPLING FOR IMBALANCED LEARNING BASED ON K-MEANS AND SMOTE

by

Felix Last

Dissertation presented as partial requirement for obtaining the Master's degree in Advanced Analytics

Advisor: Fernando Bação

ABSTRACT

Learning from class-imbalanced data continues to be a common and challenging problem in supervised learning as standard classification algorithms are designed to handle balanced class distributions. While different strategies exist to tackle this problem, methods which generate artificial data to achieve a balanced class distribution are more versatile than modifications to the classification algorithm. Such techniques, called oversamplers, modify the training data, allowing any classifier to be used with class-imbalanced datasets. Many algorithms have been proposed for this task, but most are complex and tend to generate unnecessary noise. This work presents a simple and effective oversampling method based on k-means clustering and SMOTE oversampling, which avoids the generation of noise and effectively overcomes imbalances between and within classes. Empirical results of extensive experiments with 71 datasets show that training data oversampled with the proposed method improves classification results. Moreover, k-means SMOTE consistently outperforms other popular oversampling methods. An implementation is made available in the python programming language.

KEYWORDS

Class-Imbalanced Learning; Oversampling; Classification; Clustering; Supervised Learning

INDEX

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

A-SUWO adaptive semi-unsupervised weighted oversampling 4
AUPRC area under the precision-recall curve 11
GBM gradient boosting machine 12
KNN k-nearest neighbors 12
LR logistic regression 12
PR precision-recall 11
SMOTE synthetic minority over-sampling technique 1
SOMO self-organizing map oversampling 5

1 INTRODUCTION

The class imbalance problem in machine learning describes classification tasks in which classes of data are not equally represented. In many real-world applications, the nature of the problem implies a sometimes heavy skew in the class distribution of a binary or multi-class classification problem. Such applications include fraud detection in banking, rare medical diagnoses, and oil spill recognition in satellite images, all of which naturally exhibit a minority class (Chawla et al., 2002; Kotsiantis et al., 2006, 2007; Galar et al., 2012).

The predictive capability of classification algorithms is impaired by class imbalance. Many such algorithms aim at maximizing classification accuracy, a measure which is biased towards the majority class. A classifier can achieve high classification accuracy even when it does not predict a single minority class instance correctly. For example, a trivial classifier which scores all credit card transactions as legit will score a classification accuracy of 99.9% assuming that 0.1% of transactions are fraudulent; however in this case, all fraud cases remain undetected. In conclusion, by optimizing classification accuracy, most algorithms assume a balanced class distribution (Provost, 2000; Kotsiantis et al., 2007).

Another inherent assumption of many classification algorithms is the uniformity of misclassification costs, which is rarely a characteristic of real-world problems. Typically in imbalanced datasets, misclassifying the minority class as the majority class has a higher cost associated with it than vice versa. An example of this is database marketing, where the cost of mailing to a non-respondent is much lower than the lost profit of not mailing to a respondent (Domingos, 1999).

Lastly, what is referred to as the "small disjuncts problem" is often encountered in imbalanced datasets (Galar et al., 2012). The problem refers to classification rules covering only a small number of training examples. The presence of only few samples make rule induction more susceptible to error (Holte et al., 1989). To illustrate the importance of discovering high quality rules for sparse areas of the input space, the example of credit card fraud detection is again considered. Assume that most fraudulent transactions are processed outside the card owner's home country. The remaining cases of fraud happen within the country, but show some different exceptional characteristic, such as a high amount, an unusual time or recurring charges. Each of these other characteristics applies to only a very small group of transactions, which by itself is often vanishingly small. However, adding up all these edge cases, they can make up a substantial portion of all fraudulent transactions. Therefore, it is important that classifiers pay adequate attention to small disjuncts (Holte et al., 1989).

Techniques aimed at improving classification in the presence of class imbalance can be divided into three broad categories[1]: algorithm level methods, data level methods, and cost-sensitive methods.

Solutions which modify the classification algorithm to cope with the imbalance are algorithm level techniques (Kotsiantis et al., 2006; Galar et al., 2012). Such techniques include changing the decision threshold and training separate classifiers for each class (Kotsiantis et al., 2006; Chawla et al., 2004).

In contrast, cost-sensitive methods aim at providing classification algorithms with different misclassification costs for each class. This requires knowledge of misclassification costs, which are dataset-dependent and commonly unknown or difficult to quantify. Additionally, the algorithms must be capable of incorporating the misclassification cost of each class or instance into their optimization. Therefore, these methods are regarded as operating both on data and algorithm level (Galar et al., 2012).

Finally, data-level methods manipulate the training data, aiming to change the class distribution towards a more balanced one. Techniques in this category resample the data by removing cases of the majority classes (undersampling) or adding instances to the minority classes by means of duplication or generation of new samples (oversampling) (Kotsiantis et al., 2006; Galar et al., 2012). Because undersampling removes data, such methods risk the loss of important concepts. Moreover, when the number of minority observations is small, undersampling to a balanced distribution yields an undersized dataset, which may in turn limit classifier performance. Oversampling, on the other hand, may encourage overfitting when observations are merely duplicated (Weiss et al., 2007). This problem can be avoided by adding genuinely new samples. One straightforward approach to this is synthetic minority over-sampling technique (SMOTE), which interpolates existing samples to generate new instances.

Data-level methods can be further discriminated into random and informed methods. Unlike random methods, which randomly choose samples to be removed or duplicated (e.g. Random Oversampling, SMOTE), informed methods take into account the distribution of the samples (Chawla et al., 2004). This allows informed methods

[1] The three categories are not exhaustive and new categories have been introduced, such as the combination of each of these techniques with ensemble learning (Galar et al., 2012).

to direct their efforts to critical areas of the input space, for instance to sparse areas (Nickerson et al., 2001), safe areas (Bunkhumpornpat et al., 2009), or to areas close to the decision boundary (Han et al., 2005). Consequently, informed methods may avoid the generation of noise and can tackle imbalances within classes.

Unlike algorithm-level methods, which are bound to a specific classifier, and cost-sensitive methods, which are problem-specific and need to be implemented by the classifier, data-level methods can be universally applied and are therefore more versatile (Galar et al., 2012).

Many oversampling techniques have proven to be effective in real-world domains. SMOTE is the most popular oversampling method that was proposed to improve random oversampling. There are multiple variations of SMOTE which aim to combat the original algorithm's weaknesses. Yet, many of these approaches are either very complex or alleviate only one of SMOTE's shortcomings. Additionally, few of them are readily available in a unified software framework used by practitioners.

This paper suggests the combination of the k-means clustering algorithm in combination with SMOTE to combat some of other oversampler's shortcomings with a simple-to-use technique. The use of clustering enables the proposed oversampler to identify and target areas of the input space where the generation of artificial data is most effective. The method aims at eliminating both between-class imbalances and within-class imbalances while at the same time avoiding the generation of noisy samples. Its appeal is the widespread availability of both underlying algorithms as well the effectiveness of the method itself.

While the proposed method could easily be extended to cope with multi-class problems, the focus of this work is placed on binary classification tasks. When working with more than two imbalanced classes, different aspects of classification, as well as evaluation, must be considered, which is discussed in detail by Fernández et al. (2013).

The remainder of this work is organized as follows. In section 2, related work is summarized and currently available oversampling methods are introduced. Special attention is paid to oversamplers which - like the proposed method - employ a clustering procedure. Section 3 explains in detail how the proposed oversampling technique works and which hyperparameters need to be tuned. It is shown that both SMOTE and random oversampling are limit cases of the algorithm and how they can be achieved. In section 4, a framework aimed at evaluating the performance of the proposed method in comparison with other oversampling techniques is established. The experimental results are shown in section 5, which is followed by section 6 presenting the conclusions.

2 RELATED WORK

Methods to cope with class imbalance either alter the classification algorithm itself, incorporate misclassification costs of the different classes into the classification process, or modify the data used to train the classifier. Resampling the training data can be done by removing majority class samples (undersampling) or by inflating the minority class (oversampling). Oversampling techniques either duplicate existing observations or generate artificial data. Such methods may work uninformed, randomly choosing samples which are replicated or used as a basis for data generation, or informed, directing their efforts to areas where oversampling is deemed most effective. Among informed oversamplers, clustering procedures are sometimes applied to determine suitable areas for the generation of synthetic samples.

Random oversampling randomly duplicates minority class instances until the desired class distribution is reached. Due to its simplicity and ease of implementation, it is likely to be the method that is most frequently used among practitioners. However, since samples are merely replicated, classifiers trained on randomly oversampled data are likely to suffer from overfitting[2] (Batista et al., 2004; Chawla et al., 2004).

In 2002, Chawla et al. (2002) suggested the SMOTE algorithm, which avoids the risk of overfitting faced by random oversampling. Instead of merely replicating existing observations, the technique generates artificial samples. As shown in figure 1, this is achieved by linearly interpolating a randomly selected minority observation and one of its neighboring minority observations. More precisely, SMOTE executes three steps to generate a synthetic sample. Firstly, it chooses a random minority observation $\vec{a}$. Among its k nearest minority class neighbors, instance $\vec{b}$ is selected. Finally, a new sample $\vec{x}$ is created by randomly interpolating the two samples: $\vec{x} = \vec{a} + w \times (\vec{b} - \vec{a})$, where w is a random weight in $[0, 1]$.

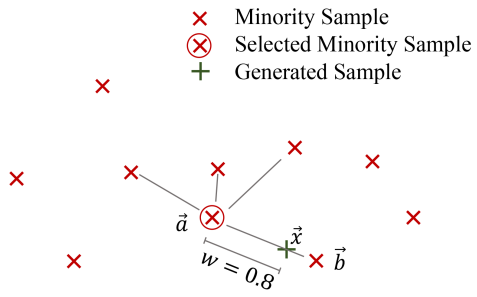

Figure 1: SMOTE linearly interpolates a randomly selected minority sample and one of its $k = 4$ nearest neighbors

However, the algorithm has some weaknesses dealing with imbalance and noise as illustrated in figure 2. One such drawback stems from the fact that SMOTE randomly chooses a minority instance to oversample with uniform probability. While this allows the method to effectively combat between-class imbalance, the issues of within-class imbalance and small disjuncts are ignored. Input areas counting many minority samples have a high probability of being inflated further, while sparsely populated minority areas are likely to remain sparse (Prati et al., 2004).

Another major concern is that SMOTE may further amplify noise present in the data. This is likely to happen when linearly interpolating a noisy minority sample, which is located among majority class instances, and its nearest minority neighbor. The method is susceptible to noise generation because it doesn't distinguish overlapping class regions from so-called safe areas (Bunkhumpornpat et al., 2009).

Finally, the algorithm does not specifically enforce the decision boundary. Instances far from the class border are oversampled with the same probability as those close to the boundary. It has been argued that classifiers could benefit from the generation of samples closer to the class border (Han et al., 2005).

Despite its weaknesses, SMOTE has been widely adopted by researchers and practitioners, likely due to its simplicity and added value with respect to random oversampling. Numerous modifications and extensions of

[2]Overfitting occurs when a model does not conform with the principle of parsimony. A flexible model with more parameters than required is predisposed to fit individual observations rather than the overall distribution, typically impairing the model's ability to predict unseen data (Hawkins, 2004). In random oversampling, overfitting may occur when classifiers construct rules which seemingly cover multiple observations, while in fact they only cover many replicas of the same observation (Batista et al., 2004).

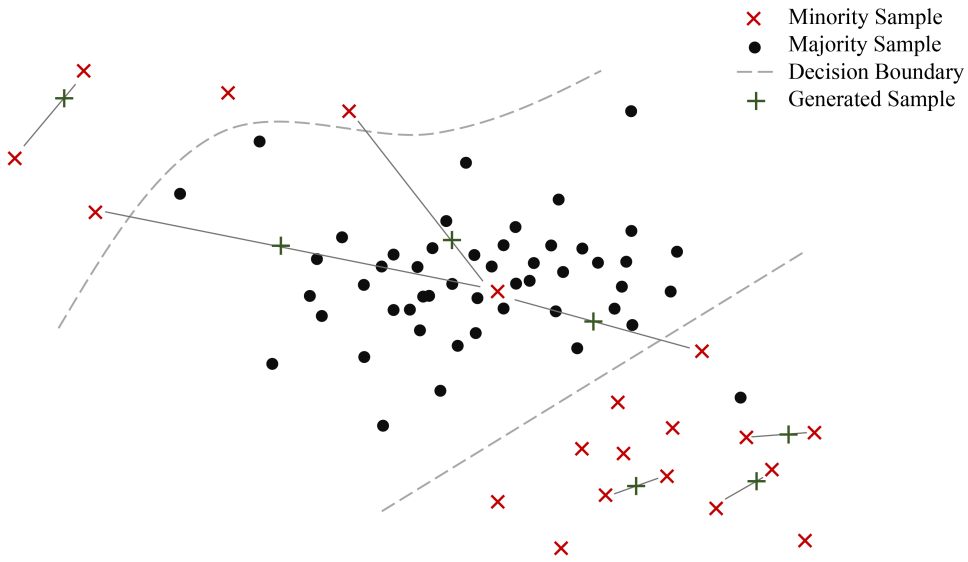

Figure 2: Behavior of SMOTE in the presence of noise and within-class imbalance

the technique have been proposed, which aim to eliminate its disadvantages. Such modifications typically address one of the original method's weaknesses. They may be divided according to their claimed goal into algorithms which aim to emphasize certain minority class regions, intend to combat within-class imbalance, or attempt to avoid the generation of noise.

Focussing its attention on the decision boundary, borderline-SMOTE1 belongs to the category of methods emphasizing class regions. It is the only algorithm discussed here which does not employ a clustering procedure and is included due to its popularity. The technique replaces SMOTE's random selection of observations with a targeted selection of instances close to the class border. The label of a sample's k nearest neighbors is used to determine whether it is discarded as noise, selected for its presumed proximity to the class border, or ruled out because it is far from the boundary. Borderline-SMOTE2 extends this method to allow interpolation of a minority instance and one of its *majority* class neighbors, setting the interpolation weight to less than 0.5 so as to place the generated sample closer to the minority sample (Han et al., 2005).

Cluster-SMOTE, another method in the category of techniques emphasizing certain class regions, uses k-means to cluster the minority class before applying SMOTE within the found clusters. The stated goal of this method is to boost class regions by creating samples within naturally occurring clusters of the minority class. It is not specified how many instances are generated in each cluster, nor how the optimal number of clusters can be determined (Cieslak et al., 2006). While the method may alleviate the problem of between-class imbalance, it does not help to eliminate small disjuncts.

Belonging to the same category, adaptive semi-unsupervised weighted oversampling (A-SUWO) introduced by Nekooeimehr and Lai-Yuen (2016), is a rather complex technique which applies clustering to improve the quality of oversampling. The approach is based on hierarchical clustering and aims at oversampling hard-to-learn instances close to the decision boundary.

Among techniques which aim to reduce within-class imbalance at the same time as between-class imbalance is cluster-based oversampling. The algorithm clusters the entire input space using k-means. Random oversampling is then applied within clusters so that: a) all majority clusters, b) all minority clusters, and c) the majority and minority classes are of the same size (Jo and Japkowicz, 2004). By replicating observations instead of generating new ones, this technique may encourage overfitting.

With a bi-directional sampling approach, Song et al. (2016) combine undersampling the majority class with oversampling the minority class. K-means clustering is applied separately within each class with the goal of achieving within- and between-class balance. For clustering the majority class, the number of clusters is set to the desired number of samples (equal to the geometric mean of instances per class). The class is undersampled by retaining only the nearest neighbor of each cluster centroid. The minority class is clustered into two partitions. Subsequently, SMOTE is applied in the smaller cluster. A number of iterations of clustering and SMOTE are performed until both classes are of equal size. It is unclear how many samples are added at

each iteration. Since the method clusters both classes separately, it is blind to overlapping class borders and may contribute to noise generation.

The self-organizing map oversampling (SOMO) algorithm transforms the input data into a two-dimensional space using a self-organizing map, where safe and effective areas are identified for data generation. SMOTE is then applied within clusters found in the lower dimensional space, as well as between neighboring clusters in order to correct within- and between-class imbalances (Douzas and Bacao, 2017).

Aiming to avoid noise generation, a clustering-based approach called CURE-SMOTE uses the hierarchical clustering algorithm CURE to clear the data of outliers before applying SMOTE. The rationale behind this method is that because SMOTE would amplify existing noise, the data should be cleared of noisy observations prior to oversampling (Ma and Fan, 2017). While noise generation is avoided, possible imbalances within the minority class are ignored.

Finally, Santos et al. (2015) cluster the entire input space with k-means. Clusters with few representatives are chosen to be oversampled using SMOTE. The algorithm is different from most oversampling methods in that SMOTE is applied regardless of the class label. The class label of the generated sample is copied from the nearest of the two parents. The algorithm thereby targets dataset imbalance, rather than imbalances between or within classes and cannot be used to solve class imbalance.

In summary, there has been a lot of recent research aimed at the improvement of imbalanced dataset resampling. Some proposed methods employ clustering techniques before applying random oversampling or SMOTE. While most of them manage to combat some weaknesses of existing oversampling algorithms, none have been shown to avoid noise generation and alleviate imbalances both between and within classes at the same time. Additionally, many techniques achieve their respective improvements at the cost of high complexity, making the techniques difficult to implement and use.

3 PROPOSED METHOD

The method proposed in this work employs the simple and popular k-means clustering algorithm in conjunction with SMOTE oversampling in order to rebalance skewed datasets. It manages to avoid the generation of noise by oversampling only in safe areas. Moreover, its focus is placed on both between-class imbalance and within-class imbalance, combating the small disjuncts problem by inflating sparse minority areas. The method is easily implemented due to its simplicity and the widespread availability of both k-means and SMOTE. It is uniquely different from related methods not only due to its low complexity but also because of its effective approach to distributing synthetic samples based on cluster density.

3.1 ALGORITHM

K-means SMOTE consists of three steps: clustering, filtering, and oversampling. In the clustering step, the input space is clustered into k groups using k-means clustering. The filtering step selects clusters for oversampling, retaining those with a high proportion of minority class samples. It then distributes the number of synthetic samples to generate, assigning more samples to clusters where minority samples are sparsely distributed. Finally, in the oversampling step, SMOTE is applied in each selected cluster to achieve the target ratio of minority and majority instances. The algorithm is illustrated in figure 3.

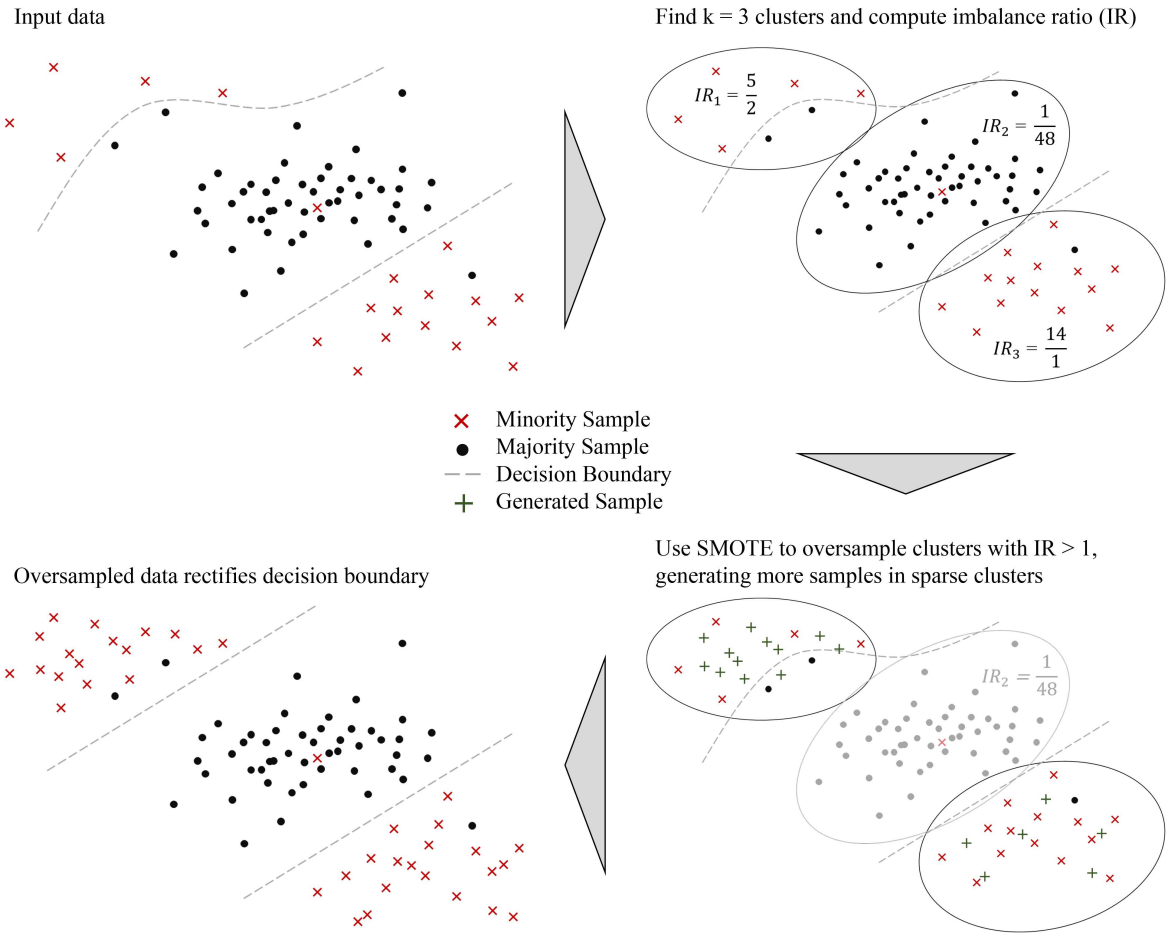

Figure 3: K-means SMOTE oversamples safe areas and combats within-class imbalance[3]

The k-means algorithm is a popular iterative method of finding naturally occurring groups in data which can be represented in a Euclidean space. It works by iteratively repeating two instructions: First, assign each observation to the nearest of k cluster centroids. Second, update the position of the centroids so

[3]The rectification of the decision boundary in the lower left of the image is desired because the two majority samples are considered outliers. The classifier is thus able to induce a simpler rule, which is less error prone.

that they are centered between the observations assigned to them. The algorithm converges when no more observations are reassigned. It is guaranteed to converge to a typically local optimum in a finite number of iterations (MacQueen, 1967). For large datasets where k-means may be slow to converge, more efficient implementations could be used for the clustering step of the k-means SMOTE, such as mini-batch k-means as proposed by (Sculley, 2010). All hyperparameters of k-means are also hyperparameters of the proposed algorithm, most notably k, the number of clusters. Finding an appropriate value for k is essential for the effectiveness of k-means SMOTE as it influences how many minority clusters, if any, can be found in the filter step.

Following the clustering step, the filter step chooses clusters to be oversampled and determines how many samples are to be generated in each cluster. The motivation of this step is to oversample only clusters dominated by the minority class, as applying SMOTE inside minority areas is less susceptible to noise generation. Moreover, the goal is to achieve a balanced distribution of samples *within* the minority class. Therefore, the filter step allocates more generated samples to sparse minority clusters than to dense ones.

The selection of clusters for oversampling is based on each cluster's proportion of minority and majority instances. By default, any cluster made up of at least 50 % minority samples is selected for oversampling. This behavior can be tuned by adjusting the imbalance ratio threshold (or irt), a hyperparameter of k-means SMOTE which defaults to 1. The imbalance ratio of a cluster c is defined as $\frac{majorityCount(c)+1}{minorityCount(c)+1}$. When the imbalance ratio threshold is increased, cluster choice is more selective and a higher proportion of minority instances is required for a cluster to be selected. On the other hand, lowering the threshold loosens the selection criterion, allowing clusters with a higher majority proportion to be chosen.

To determine the distribution of samples to be generated, filtered clusters are assigned sampling weights between zero and one. A high sampling weight corresponds to a low density of minority samples and yields more generated samples. To achieve this, the sampling weight depends on how dense a single cluster is compared to how dense all selected clusters are on average. Note that when measuring a cluster's density, only the distances among minority instances are considered. The computation of the sampling weight may be expressed by means of five sub-computations:

1. For each filtered cluster f, calculate the Euclidean distance matrix, ignoring majority samples.
2. Compute the mean distance within each cluster by summing all non-diagonal elements of the distance matrix, then dividing by the number non-diagonal elements.
3. To obtain a measure of density, divide each cluster's number of minority instances by its average minority distance raised to the power of the number of features m: $density(f) = \frac{minorityCount(f)}{averageMinorityDistance(f)^m}$.
4. Invert the density measure as to get a measure of sparsity, i.e. $sparsity(f) = \frac{1}{density(f)}$.
5. The sampling weight of each cluster is defined as the cluster's sparsity factor divided by the sum of all clusters' sparsity factors.

Consequently, the sum of all sampling weights is one. Due to this property, the sampling weight of a cluster can be multiplied by the overall number of samples to be generated to determine the number of samples to be generated in that cluster.

In the oversampling step of the algorithm, each filtered cluster is oversampled using SMOTE. For each cluster, the oversampling procedure is given all points of the cluster along with the instruction to generate $\|samplingWeight(f) \times n\|$ samples, where n is the overall number of samples to be generated. Per synthetic sample to generate, SMOTE chooses a random minority observation $\vec{a}$ within the cluster, finds a random neighboring minority instance $\vec{b}$ of that point and determines a new sample $\vec{x}$ by randomly interpolating $\vec{a}$ and $\vec{b}$. In geometric terms, the new point $\vec{x}$ is thus placed somewhere along a straight line from $\vec{a}$ to $\vec{b}$. The process is repeated until the number of samples to be generated is reached.

SMOTE's hyperparameter k nearest neighbors, or knn, constitutes among how many neighboring minority samples of $\vec{a}$ the point $\vec{b}$ is randomly selected. This hyperparameter is also used by k-means SMOTE. Depending on the specific implementation of SMOTE, the value of knn may have to be adjusted downward when a cluster has fewer than $knn + 1$ minority samples. Once each filtered cluster has been oversampled, all generated samples are returned and the oversampling process is completed.

The proposed method is distinct from related techniques in that it clusters the entire dataset regardless of the class label. An unsupervised approach enables the discovery of overlapping class regions and may aid the avoidance of oversampling in unsafe areas. This is in contrast to cluster-SMOTE, where only minority class

instances are clustered (Cieslak et al., 2006) and to the aforementioned combination of oversampling and undersampling where both classes are clustered separately (Song et al., 2016). Another distinguishing feature is the unique approach to the distribution of generated samples across clusters: sparse minority clusters yield more samples than dense ones. The previously presented method cluster-based oversampling, on the other hand, distributes samples based on cluster size (Jo and Japkowicz, 2004). Since k-means may find clusters of varying density, but typically of the same size (MacQueen, 1967), distributing samples according to cluster density can be assumed to be an effective way to combat within-class imbalance. Lastly, the use of SMOTE circumvents the problem of overfitting, which random oversampling has been shown to encourage.

Input: X (matrix of observations)
y (target vector)
n (number of samples to be generated)
k (number of clusters to be found by k-means)
irt (imbalance ratio threshold)
knn (number of nearest neighbors considered by SMOTE)
de (exponent used for computation of density; defaults to the number of features in X)

begin

`// Step 1: Cluster the input space and filter clusters with more minority instances than majority instances.`

$clusters \leftarrow kmeans(X)$
$filteredClusters \leftarrow \varnothing$
for $c \in clusters$ **do**
  $imbalanceRatio \leftarrow \frac{majorityCount(c)+1}{minorityCount(c)+1}$
  if $imbalanceRatio < irt$ **then**
    $filteredClusters \leftarrow filteredClusters \cup \{c\}$
  end
end

`// Step 2: For each filtered cluster, compute the sampling weight based on its minority density.`

for $f \in filteredClusters$ **do**
  $averageMinorityDistance(f) \leftarrow mean(euclideanDistances(f))$
  $densityFactor(f) \leftarrow \frac{minorityCount(f)}{averageMinorityDistance(f)^{de}}$
  $sparsityFactor(f) \leftarrow \frac{1}{densityFactor(f)}$
end
$sparsitySum \leftarrow \sum_{f \in filteredClusters} sparsityFactor(f)$
$samplingWeight(f) \leftarrow \frac{sparsityFactor(f)}{sparsitySum}$

`// Step 3: Oversample each filtered cluster using SMOTE. The number of samples to be generated is computed using the sampling weight.`

$generatedSamples \leftarrow \varnothing$
for $f \in filteredClusters$ **do**
  $numberOfSamples \leftarrow \|n \times samplingWeight(f)\|$
  $generatedSamples \leftarrow generatedSamples \cup \{SMOTE(f, numberOfSamples, knn\}$
end
return $generatedSamples$

end

Algorithm 1: Proposed method based on k-means and SMOTE

3.2 LIMIT CASES

In the following, it is shown that SMOTE and random oversampling can be regarded as limit cases of the more general method proposed in this work. In k-means SMOTE, the input space is clustered using k-means. Subsequently, some clusters are selected and then oversampled using SMOTE. Considering the case where the number of clusters k is equal to 1, all observations are grouped in one cluster. For this only cluster to

be selected as a minority cluster, the imbalance ratio threshold needs to be set so that the imbalance ratio of the training data is met. For example, in a dataset with 100 minority observations and 10,000 majority observations, the imbalance ratio threshold must be greater than or equal to $\frac{10{,}000+1}{100+1} \approx 99.02$. The single cluster is then selected and oversampled using SMOTE; since the cluster contains all observations, this is equivalent to simply oversampling the original dataset with SMOTE. Instead of setting the imbalance ratio threshold to the exact imbalance ratio of the dataset, it can simply be set to positive infinity.

If SMOTE did not interpolate two different points to generate a new sample but performed the random interpolation of one and the same point, the result would be a copy of the original point. This behavior could be achieved by setting the parameter "k nearest neighbors" of SMOTE to zero if the concrete implementation supports this behavior. As such, random oversampling may be regarded as a specific case of SMOTE.

This property of k-means SMOTE is of very practical value to its users: since it contains both SMOTE and random oversampling, a search of optimal hyperparameters could include the configurations for those methods. As a result, while a better parametrization may be found, the proposed method will perform at least as well as the better of both oversamplers. In other words, SMOTE and random oversampling are fallbacks contained in k-means SMOTE, which can be resorted to when the proposed method does not produce any gain with other parametrizations. Table 1 summarizes the parameters which may be used to reproduce the behavior of both algorithms.

	k	irt	knn
SMOTE	1	∞	
Random Oversampling	1	∞	0

Table 1: Limit case configurations

4 RESEARCH METHODOLOGY

The ultimate purpose of any resampling method is the improvement of classification results. In other words, a resampling technique is successful if the resampled data it produces improves the prediction quality of a given classifier. Therefore, the effectiveness of an oversampling method can only be assessed indirectly by evaluating a classifier trained on oversampled data. This proxy measure, i.e. the classifier performance, is only meaningful when compared with the performance of the same classification algorithm trained on data which has not been resampled. Multiple oversampling techniques can then be ranked by evaluating a classifier's performance with respect to each modified training set produced by the resamplers.

A general concern in classifier evaluation is the bias of evaluating predictions for previously seen data. Classifiers may perform well when making predictions for rows of data used during training, but poorly when classifying new data. This problem is also referred to as overfitting. Oversampling techniques have been observed to encourage overfitting, which is why this bias should be carefully avoided during their evaluation. A general approach is to split the available data into two or more subsets of which only one is used during training, and another is used to evaluate the classification. The latter is referred to as the holdout set, unknown data, or test dataset.

Arbitrarily splitting the data into two sets, however, may introduce additional biases. One potential issue that arises is that the resulting training set may not contain certain observations, preventing the algorithm from learning important concepts. Cross-validation combats this issue by randomly splitting the data many times, each time training the classifier from scratch using one portion of the data before measuring its performance on the remaining share of data. After a number of repetitions, the classifier can be evaluated by aggregating the results obtained in each iteration. In k-fold cross-validation, a popular variant of cross-validation, k iterations, called folds, are performed. During each fold, the test set is one of k equally sized groups. Each group of observations is used exactly once as a holdout set. K-fold cross-validation can be repeated many times to avoid potential bias due to random grouping (Japkowicz, 2013).

While k-fold cross validation typically avoids the most important biases in classification tasks, it might distort the class distributions when randomly sampling from a class-imbalanced dataset. In the presence of extreme skews, there may even be iterations where the test set contains no instances of the minority class, in which case classifier evaluation would be ill-defined or potentially strongly biased. A simple and common approach to this problem is to use stratified cross-validation, where instead of sampling completely at random, the original class distribution is preserved in each fold (Japkowicz, 2013).

4.1 METRICS

Of the various assessment metrics traditionally used to evaluate classifier performance, not all are suitable when the class distribution is not uniform. However, there are metrics which have been employed or developed specifically to cope with imbalanced data.

Classification assessment metrics compare the true class membership of each observation with the prediction of the classifier. To illustrate the alignment of predictions with the true distribution, a confusion matrix (figure 4) can be constructed. Possibly deriving from medical diagnoses, a positive observation is a rare case and belongs to the minority class. The majority class is considered negative (Japkowicz, 2013).

	P Positives	N Negatives	
PP Predicted Positives	TP True Positives	FP False Positives	Precision $\frac{TP}{PP}$
PN Predicted Negatives	FN False Negatives	TN True Negatives	
	Sensitivity / Recall $\frac{\mathrm{TP}}{P}$	Specificity $\frac{TN}{N}$	

Figure 4: Confusion matrix

When evaluating a single classifier in the context of a finite dataset with fixed imbalance ratio, the confusion matrix provides all necessary information to assess the classification quality. However, when comparing different

classifiers or evaluating a single classifier in variable environments, the absolute values of the confusion matrix are non-trivial.

The most common metrics for classification problems are accuracy and its inverse, error rate.

$$Accuracy = \frac{TP + TN}{P + N};\ ErrorRate = 1 - Accuracy \tag{1}$$

These metrics show a bias toward the majority class in imbalanced datasets. For example, a naive classifier which predicts all observations as negative would achieve 99% accuracy in a dataset where only 1% of instances are positive. While such high accuracy suggests an effective classifier, the metric obscures the fact that not a single minority instance was predicted correctly (He and Garcia, 2009).

Sensitivity, also referred to as recall or true positive rate, explains the prediction accuracy among minority class instances. It answers the question "How many minority instances were correctly classified as such?" Specificity answers the same question for the majority class. Precision is the rate of correct predictions among all instances predicted to belong to the minority class. It indicates how many of the positive predictions are correct (He and Garcia, 2009).

The F1-score, or F-measure, is the (often weighted) harmonic mean of precision and recall. In other terms, the indicator rates both the completeness and exactness of positive predictions (He and Garcia, 2009; Japkowicz, 2013).

$$F1 = \frac{(1+\alpha) \times (sensitivity \times precision)}{sensitivity + \alpha \times precision} = \frac{(1+\alpha) \times (\frac{TP}{P} \times \frac{TP}{PP})}{\frac{TP}{P} + \alpha \times \frac{TP}{PP}} \tag{2}$$

The geometric mean score, also referred to as g-mean or g-measure, is defined as the geometric mean of sensitivity and specificity. The two components can be regarded as per-class accuracy. The g-measure aggregates both metrics into a single value in $[0, 1]$, assigning equal importance to both (He and Garcia, 2009; Japkowicz, 2013).

$$g\text{-}mean = \sqrt{sensitivity \times specificity} = \sqrt{\frac{TP}{P} \times \frac{TN}{N}} \tag{3}$$

Precision-recall (PR) diagrams plot the precision of a classifier as a function of its minority accuracy. Classifiers outputting class membership confidences (i.e. continuous values in $[0, 1]$) can be plotted as multiple points in discrete intervals, resulting in a PR curve. Commonly, the area under the precision-recall curve (AUPRC) is computed as a single numeric performance metric (He and Garcia, 2009; Japkowicz, 2013).

The choice of metric depends to a great extent on the goal their user seeks to achieve. In certain practical tasks, one specific aspect of classification may be more important than another (e.g. in medical diagnoses, false negatives are much more critical than false positives). However, to determine a general ranking among oversamplers, no such focus should be placed. Therefore, the following unweighted metrics are chosen for the evaluation.

- g-mean
- F1-score
- AUPRC

4.2 OVERSAMPLERS

The following list enumerates the oversamplers used as a benchmark for the evaluation of the proposed method, along with the set of hyperparameters used for each. The optimal imbalance ratio is not obvious and has been discussed by other researchers (Provost, 2000; Estabrooks et al., 2004). This work aims at creating comparability among oversamplers; consequently, it is most important that oversamplers achieve the same imbalance ratio. Therefore, all oversampling methods were parametrized to generate as many instances as necessary so that minority and majority classes count the same number of samples.

- random oversampling

- SMOTE
  - $knn \in \{3, 5, 20\}$
- borderline-SMOTE1
  - $knn \in \{3, 5, 20\}$
- borderline-SMOTE2
  - $knn \in \{3, 5, 20\}$
- k-means SMOTE
  - $k \in \{2, 20, 50, 100, 250, 500\}$
  - $knn \in \{3, 5, 20, \infty\}$
  - $irt \in \{1, \infty\}$
  - $de \in \{0, 2, numberOfFeatures\}$
- no oversampling

4.3 CLASSIFIERS

For the evaluation of the various oversampling methods, several different classifiers are chosen to ensure that the results obtained can be generalized and are not constrained to the usage of a specific classifier. The choice of classifiers is further motivated by the number of hyperparameters: classification algorithms with few or no hyperparameters are less likely to bias results due to their specific configuration.

Logistic regression (LR) is a generalization of linear regression which can be used for binary classification. Fitting the model is an optimization problem which can be solved using simple optimizers which require no hyperparameters to be set (McCullagh, 1984). Consequently, results achieved by LR are easily reproducible, while also constituting a baseline for more sophisticated approaches.

Another classification algorithm referred to as k-nearest neighbors (KNN) assigns an observation to the class most of its nearest neighbors belong to. How many neighbors are considered is determined by the method's hyperparameter k (Fix and Hodges Jr., 1951).

Finally, gradient boosting over decision trees, or simply gradient boosting machine (GBM), is an ensemble technique used for classification. In the case of binary classification, one shallow decision tree is induced at each stage of the algorithm. Each tree is fitted to observations which could not be correctly classified by decision trees of previous stages. Predictions of GBM are made by majority vote of all trees. In this way, the algorithm combines several simple models (referred to as weak learners) to create one effective classifier. The number of decision trees to generate, which in binary classification is equal to the number of stages, is a hyperparameter of the algorithm (Friedman, 2001).

As further explained in section 4.5, various combinations of hyperparameters are tested for each classifier. All classifiers are used as implemented in the python library scikit-learn (Pedregosa et al., 2011) with default parameters unless stated otherwise. The following list enumerates the classifiers used in this study along with a set of values for their respective hyperparameters.

- LR
- KNN
  - $k \in \{3, 5, 8\}$
- GBM
  - $numberOfTrees \in \{50, 100, 200\}$

4.4 DATASETS

To evaluate k-means SMOTE, 12 imbalanced datasets from the UCI Machine Learning Repository (Lichman, 2013) are used. Those datasets containing more than two classes were binarized using a one-versus-rest approach, labeling the smallest class as the minority and merging all other samples into one class. In order to generate additional datasets with even higher imbalance ratios, each of the aforementioned datasets was randomly undersampled to generate up to six additional datasets. The imbalance ratio of each dataset was increased approximately by multiplication factors of 2, 4, 6, 10, 15 and 20, but only if a given factor did not reduce a dataset's total number of minority samples to less than eight. Furthermore, the python library scikit-learn (Pedregosa et al., 2011) was used to generate ten variations of the artificial "MADELON" dataset, which poses a difficult binary classification problem (Guyon, 2003).

Table 2 lists the datasets used to evaluate the proposed method, along with important characteristics. The artificial datasets are referred to as simulated. Undersampled versions of the original datasets are omitted from the table; in the figures presented in section 5, the factor used for undersampling is appended to the dataset names. All datasets used in the study are made available at `https://github.com/felix-last/evaluate-kmeans-smote/releases/download/v0.0.1/uci_extended.tar.gz` for the purpose of reproducibility.

Dataset	# features	# instances	# minority instances	# majority instances	imbalance ratio
breast_tissue	9	106	36	70	1.94
ecoli	7	336	52	284	5.46
glass	9	214	70	144	2.06
haberman	3	306	81	225	2.78
heart	13	270	120	150	1.25
iris	4	150	50	100	2.00
libra	90	360	72	288	4.00
liver_disorders	6	345	145	200	1.38
pima	8	768	268	500	1.87
segment	16	2310	330	1980	6.00
simulated1	20	4000	25	3975	159.00
simulated2	20	4000	23	3977	172.91
simulated3	20	4000	23	3977	172.91
simulated4	20	4000	26	3974	152.85
simulated5	20	4000	23	3977	172.91
simulated6	200	3000	20	2980	149.00
simulated7	200	3000	19	2981	156.89
simulated8	200	3000	15	2985	199.00
simulated9	200	3000	13	2987	229.77
simulated10	200	3000	22	2978	135.36
vehicle	18	846	199	647	3.25
wine	13	178	71	107	1.51

Table 2: Summary of datasets used to evaluate and compare the proposed method

4.5 EXPERIMENTAL FRAMEWORK

To evaluate the proposed method, the oversamplers, metrics, datasets, and classifiers discussed in this section are used. Results are obtained by repeating 5-fold cross-validation five times. For each dataset, every metric is computed by averaging their values across runs. In addition to the arithmetic mean, the standard deviation is calculated.

To achieve optimal results for all classifiers and oversamplers, a grid search procedure is used. For this purpose, each classifier and each oversampler specifies a set of possible values for every hyperparameter. Subsequently, all possible combinations of an algorithm's hyperparameters are generated and the algorithm is executed once for each combination. All metrics are used to score all resulting classifications, and the best value obtained for each metric is saved.

To illustrate this process, consider an example with one oversampler and one classifier. The following list shows each algorithm with several combinations for each parameter.

- SMOTE
  - knn: 3,6
- GBM
  - numberOfTrees: 10, 50

The oversampling method SMOTE is run two times, generating two different sets of training data. For each set of training data, the classifier LR is executed twice. Therefore, the classifier will be executed four times. The possible combinations are:

- $knn = 3, numberOfTrees = 10$
- $knn = 3, numberOfTrees = 50$
- $knn = 6, numberOfTrees = 10$
- $knn = 6, numberOfTrees = 50$

If two metrics are used for scoring, both metrics score all four runs; the best result is saved. Note that one metric may find that the combination $knn = 3, numberOfTrees = 10$ is best, while a different combination might be best considering another metric. In this way, each oversampler and classifier combination is given the chance to optimize for each metric.

5 EXPERIMENTAL RESULTS

In order to conclude whether any of the evaluated oversamplers perform consistently better than others, the cross-validated scores are used to derive a rank order, assigning rank one to the best performing and rank six to the worst performing technique (Dem, 2006). This results in different rankings for each of five experiment repetitions, again partitioned by dataset, metric, and classifier. To aggregate the various rankings, each method's assigned rank is averaged across datasets and experiment repetitions. Consequently, a method's mean rank is a real number in the interval $[1.0, 6.0]$. The mean ranking results for each combination of metric and classifier are shown in figure 5.

By testing the null hypothesis that differences in terms of rank among oversamplers are merely a matter of chance, the Friedman test (Friedman, 1937) determines the statistical significance of the derived mean ranking. The test is chosen because it does not assume normality of the obtained scores (Dem, 2006). At a significance level of $a = 0.05$, the null hypothesis is rejected for all evaluated classifiers and evaluation metrics. Therefore, the rankings are assumed to be significant.

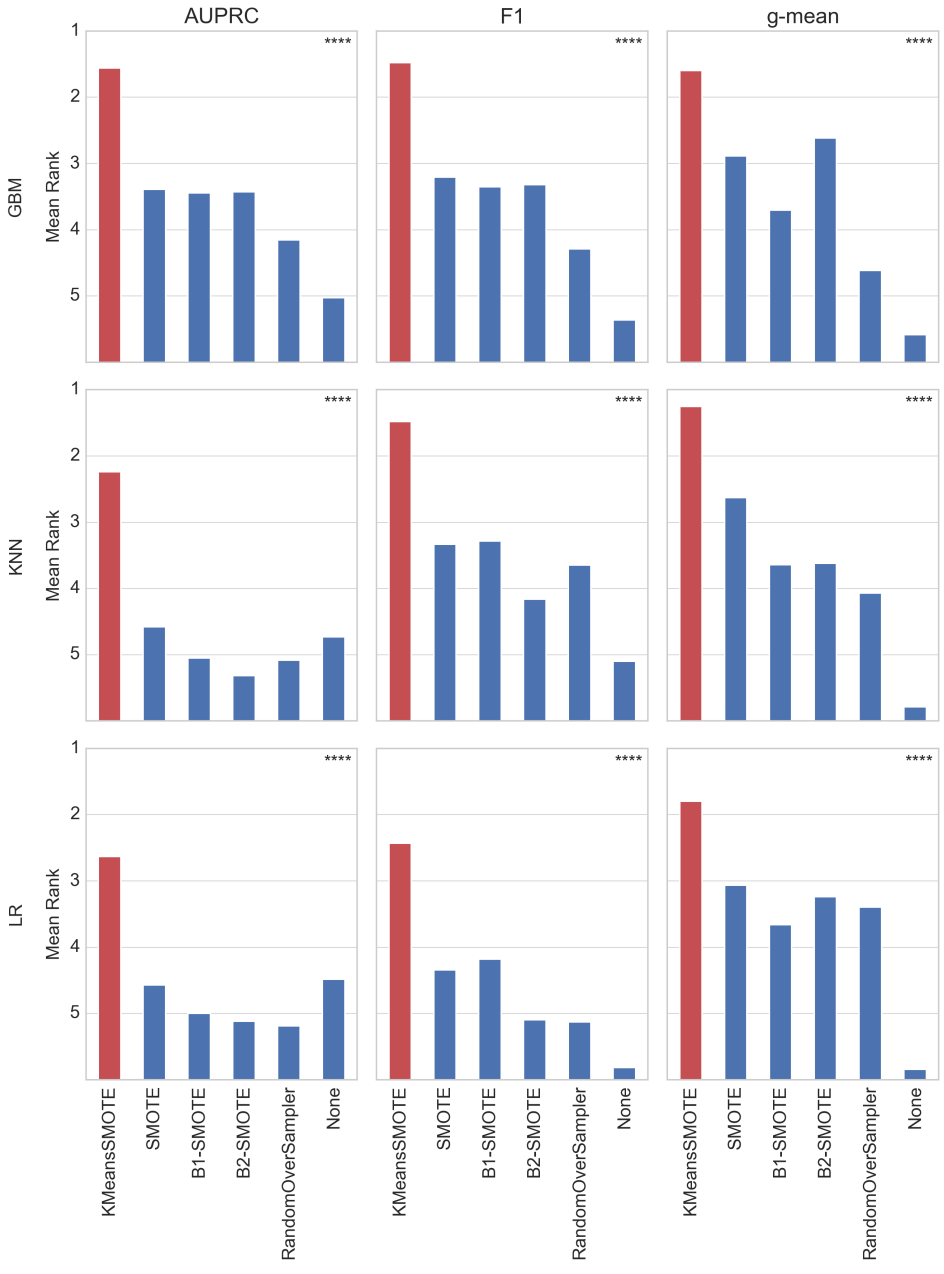

Figure 5: Mean ranking of evaluated oversamplers for different classifiers and metrics

The mean ranking shows that the proposed method outperforms other methods with regard to all evaluation metrics. Notably, the technique's superiority can be observed independently of the classifier. In six out of nine cases, k-means SMOTE achieves a mean rank better than two, whereas in the other three cases the mean rank is at least three. Furthermore, k-means SMOTE is the only technique with a mean ranking better than three with respect to F1 score and AUPRC, boosting classification results when other oversamplers tie or accomplish a similar rank as no oversampling.

Generally, it can be observed that - aside from the proposed method - SMOTE, borderline-SMOTE1, and borderline-SMOTE2 typically achieve the best results, while no oversampling frequently earns the worst rank. Remarkably, LR and KNN achieve a similar rank without oversampling as with SMOTE with regard to AUPRC, while both are only dominated by k-means SMOTE. This indicates that the proposed method may improve classifier performance even in situations where SMOTE is not able to achieve any improvement versus the original training data.

For a direct comparison to the baseline method, SMOTE, the average optimal scores attained by k-means SMOTE for each dataset are subtracted by the respective scores reached by SMOTE. The resulting score improvements achieved by the proposed method are summarized in figures 6 and 7.

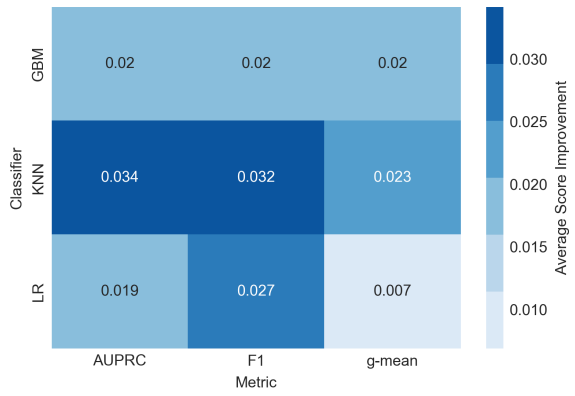

Figure 6: Mean score improvement of the proposed method versus SMOTE across datasets

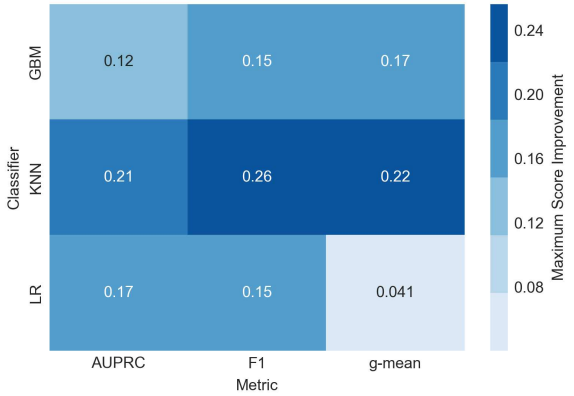

Figure 7: Maximum score improvement of the proposed method versus SMOTE

The KNN classifier appears to profit most from the application of k-means SMOTE, where maximum score improvements of more than 0.2 are observed across all metrics. The biggest mean score improvements are also achieved using KNN, with average gains ranging from 0.023 to 0.034. It can further be observed that all classifiers benefit from the suggested oversampling procedure. With only one exception, maximum score improvements of more than 0.1 are found for all classifiers and metrics.

Taking a closer look at the combination of classifier and metric which, on average, benefit most from the application of the proposed method, figure 8 shows the AUPRC achieved per dataset by each of the two oversamplers in conjunction with the KNN classifier. Although absolute scores and score differences between the two methods are dependent on the choice of metric and classifier, the general trend shown in the figure is observed for all other metrics and classifiers, which are omitted for clarity.

In the large majority of cases, k-means SMOTE outperforms SMOTE, proving the relevance of the clustering procedure. Only in 2 out of 71 datasets tested there were no improvements through the use of k-means SMOTE. On average, k-means SMOTE achieves an AUPRC improvement of 0.034. The biggest gains of the proposed method appear to be occurring in the score range of 0.2 to 0.8. On the lower end of that range, k-means SMOTE achieves an average gain of more than 0.2 compared to SMOTE in the "glass6" dataset. Prediction of dataset "vehicle15" is improved from an AUPRC of approximately 0.35 to 0.5. The third biggest gain occurs in the "ecoli6" dataset, where the proposed method obtains a score of 0.8 compared to less than 0.7 accomplished by SMOTE. The score difference among oversamplers is smaller at the extreme ends of the scale. For nine of the simulated datasets, KNN attains a score very close to zero independently of the choice of the oversampler. Similarly, for the datasets where an AUPRC around 0.95 is attained ("libra", "libra2", "iris", "iris6"), gains of k-means SMOTE are less than 0.02.

As described in sections 2 and 3, a core goal of the applied clustering approach is to avoid the generation of noise, which SMOTE is predisposed to amplify. Since minority samples generated in majority class regions

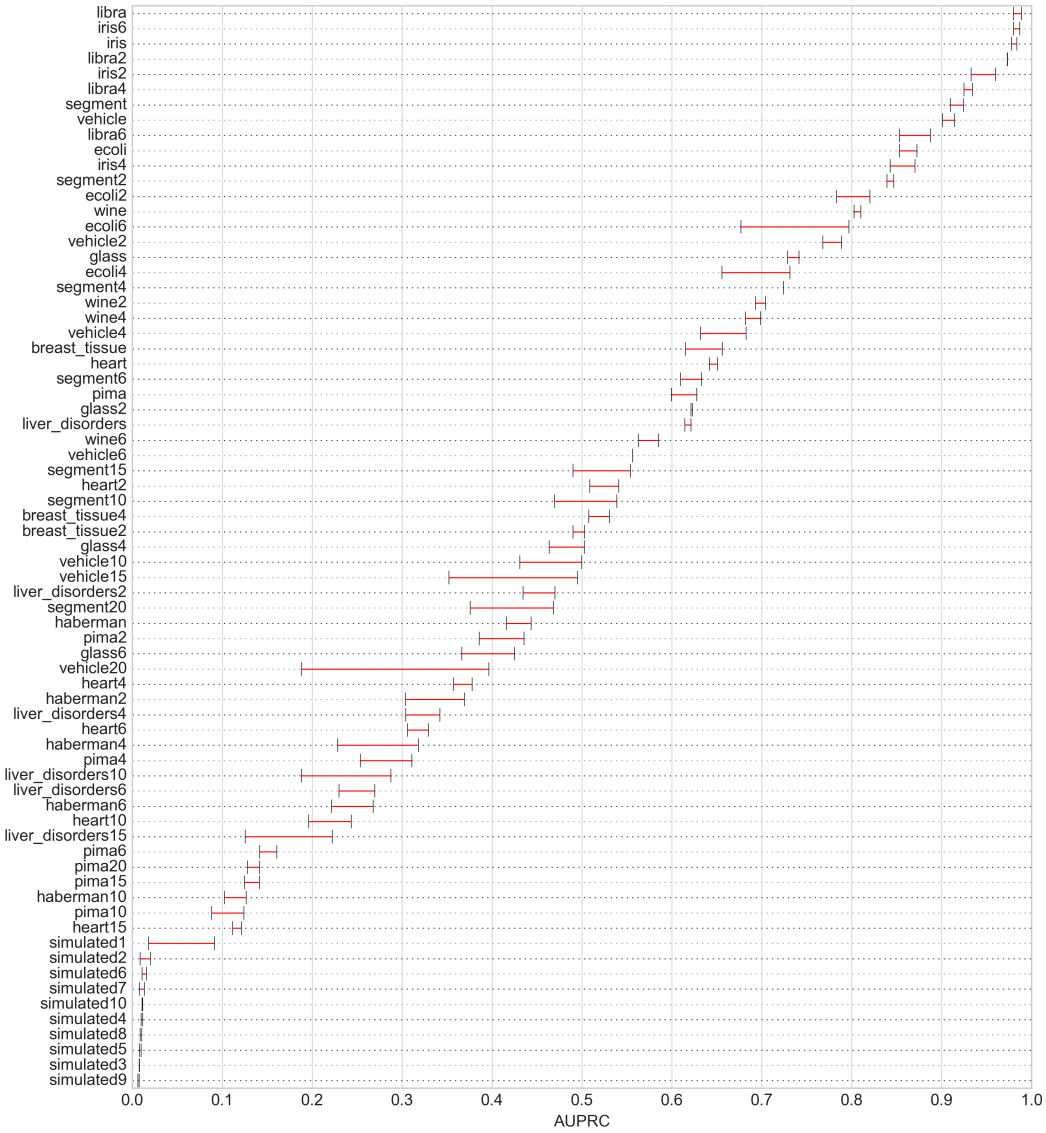

Figure 8: Performance of KNN classifier trained on data oversampled with SMOTE (left) and k-means SMOTE (right)

may contribute to an increase in false positives, the false positive count can be regarded as an indicator of the level of noise generation. As a basis to analyze whether the proposed oversampler successfully reduces the generation of noise, figure 9 visualizes the number of false positives after the application of k-means SMOTE as a percentage of the number of false positives observed using SMOTE. The false positive counts used in the chart are averages across experiment repetitions, choosing the hyperparameter configuration which attains the lowest number for each technique.

The figure illustrates that k-means SMOTE is able to reduce the number of false positives compared to SMOTE in every dataset and independently of the classifier. In many cases, k-means SMOTE eliminates more than 90% of false positives in comparison to the baseline method. On average, a 55% reduction of false positives compared to SMOTE is found, the biggest improvements being attained by the KNN classifier. Among the datasets where over 80% of false positives could be avoided are the artificial datasets as well as undersampled versions of the "segment" and "libra" datasets.

The discussed results are based on 5-fold cross-validation with five repetitions, using tests to assure statistical significance. Mean ranking results show that oversampling using k-means SMOTE improves the performance of different evaluated classifiers on imbalanced data. In addition, the proposed oversampler dominates all evaluated oversampling methods in mean rankings regardless of the classifier. Studying absolute gains of the proposed algorithm compared to the baseline method, it is found that all classifiers used in the study benefit

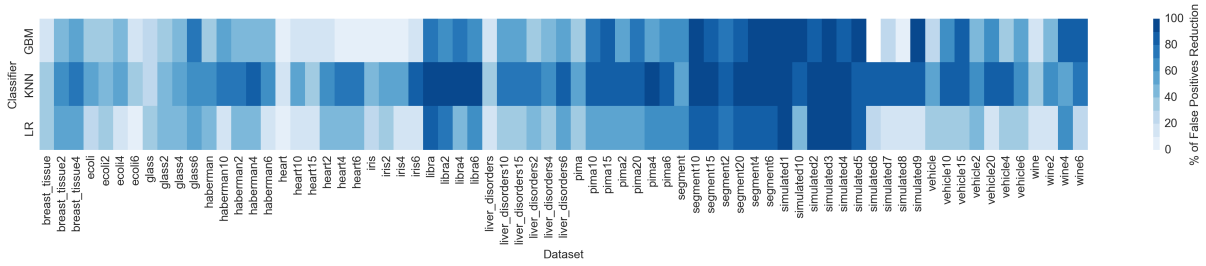

Figure 9: Relative reduction of false positives compared to SMOTE

from the clustering and sample distribution procedure of k-means SMOTE. Additionally, results of classifying almost any evaluated dataset could be improved by applying the proposed method. Generally, classification problems which are neither very easy nor very difficult profit most, allowing significant score increases of up to 0.26. Lastly, it is found that the number of false positives, which are regarded as an indicator of noise generation, can be effectively reduced by the suggested oversampling technique. Overall it is concluded that k-means SMOTE is effective in generating samples which aid classifiers in the presence of imbalance.

6 CONCLUSION

Imbalanced data poses a difficult task for many classification algorithms. Resampling training data toward a more balanced distribution is an effective way to combat this issue independently of the choice of the classifier. However, balancing classes by merely duplicating minority class instances encourages overfitting, which in turn degrades the model's performance on unseen data. Techniques which generate artificial samples, on the other hand, often suffer from a tendency to generate noisy samples, impeding the inference of class boundaries. Moreover, most existing oversamplers do not counteract imbalances within the minority class, which is often a major issue when classifying class-imbalanced datasets. For oversampling to effectively aid classifiers, the amplification of noise should be avoided by detecting safe areas of the input space where class regions do not overlap. Additionally, any imbalance within the minority class should be identified and samples are to be generated as to level the distribution.

The proposed method achieves these properties by clustering the data using k-means, allowing to focus data generation on crucial areas of the input space. A high ratio of minority observations is used as an indicator that a cluster is a safe area. Oversampling only safe clusters enables k-means SMOTE to avoid noise generation. Furthermore, the average distance among a cluster's minority samples is used to discover sparse areas. Sparse minority clusters are assigned more synthetic samples, which alleviates within-class imbalance. Finally, overfitting is discouraged by generating genuinely new observations using SMOTE rather than replicating existing ones.

Empirical results show that training various types of classifiers using data oversampled with k-means SMOTE leads to better classification results than training with unmodified, imbalanced data. More importantly, the proposed method consistently outperforms the most widely available oversampling techniques such as SMOTE, borderline-SMOTE, and random oversampling. The biggest gains appear to be achieved in classification problems which are neither extremely difficult nor extremely simple. Finally, the evaluation of the experiments conducted shows that the proposed technique effectively reduces noise generation, which is crucial in many applications. The results are statistically robust and apply to various metrics suited for the evaluation of imbalanced data classification.

The effectiveness of the algorithm is accomplished without high complexity. The method's components, k-means clustering and SMOTE oversampling, are simple and readily available in many programming languages, so that practitioners and researchers may easily implement and use the proposed method in their preferred environment. Further facilitating practical use, an implementation of k-means SMOTE in the python programming language is made available (see `https://github.com/felix-last/kmeans_smote`) based on the imbalanced-learn framework (Lemaître et al., 2017).

A prevalent issue in classification tasks, data imbalance is exhibited naturally in many important real-world applications. As the proposed oversampler can be applied to rebalance any dataset and independently of the chosen classifier, its potential impact is substantial. Among others, k-means SMOTE may, therefore, contribute to the prevention of credit card fraud, the diagnosis of diseases, as well as the detection of abnormalities in environmental observations.

Future work may consequently focus on applying k-means SMOTE to various other real-world problems. Additionally, finding optimal values of k and other hyperparameters is yet to be guided by rules of thumb, which could be deducted from further analyses of the relationship between optimal hyperparameters for a given dataset and the dataset's properties.

REFERENCES

Batista, G. E. A. P. A., Prati, R. C., and Monard, M. C. (2004). A study of the behavior of several methods for balancing machine learning training data. *ACM SIGKDD Explorations Newsletter*, 6(1):20–29.

Bunkhumpornpat, C., Sinapiromsaran, K., and Lursinsap, C. (2009). Safe-level-smote: Safe-level-synthetic minority over-sampling technique for handling the class imbalanced problem. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 5476 LNAI, pages 475–482.

Chawla, N. V., Bowyer, K. W., Hall, L. O., and Kegelmeyer, W. P. (2002). Smote: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16:321–357.

Chawla, N. V., Japkowicz, N., and Drive, P. (2004). Editorial: Special issue on learning from imbalanced data sets. *ACM SIGKDD Explorations Newsletter*, 6(1):1–6.

Cieslak, D. A., Chawla, N. V., and Striegel, A. (2006). Combating imbalance in network intrusion datasets. In *Granular Computing, 2006 IEEE International Conference on*, pages 732–737. IEEE.

Dem (2006). Statistical comparisons of classifiers over multiple data sets. *Journal of Machine learning research*, 7(Jan):1–30.

Domingos, P. (1999). Metacost: A general method for making classifiers. In *Proceedings of the 5th International Conference on Knowledge Discovery and Data Mining*, pages 155–164.

Douzas, G. and Bacao, F. (2017). Self-organizing map oversampling (somo) for imbalanced data set learning. *Expert Systems with Applications*, 82:40–52.

Estabrooks, A., Jo, T., and Japkowicz, N. (2004). A multiple resampling method for learning from imbalanced data sets. *Computational intelligence*, 20(1):18–36.

Fernández, A., López, V., Galar, M., Del Jesus, M. J., and Herrera, F. (2013). Analysing the classification of imbalanced data-sets with multiple classes: Binarization techniques and ad-hoc approaches. *Knowledge-Based Systems*, 42:97–110.

Fix, E. and Hodges Jr., J. (1951). Discriminatory analysis - nonparametric discrimination: Consistency properties.

Friedman, J. H. (2001). Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29(5):1189–1232.

Friedman, M. (1937). The use of ranks to avoid the assumption of normality implicit in the analysis of variance. *Journal of the American Statistical Association*, 32(200):675.

Galar, M., Fernandez, A., Barrenechea, E., Bustince, H., and Herrera, F. (2012). A review on ensembles for the class imbalance problem: Bagging-, boosting-, and hybrid-based approaches.

Guyon, I. (2003). Design of experiments of the nips 2003 variable selection benchmark.

Han, H., Wang, W.-Y., and Mao, B.-H. (2005). Borderline-smote: A new over-sampling method in imbalanced data sets learning. *Advances in intelligent computing*, 17(12):878–887.

Hawkins, D. M. (2004). The problem of overfitting. *Journal of chemical information and computer sciences*, 44(1):1–12.

He, H. and Garcia, E. A. (2009). Learning from imbalanced data. *IEEE Transactions on Knowledge and Data Engineering*, 21(9):1263–1284.

Holte, R. C., Acker, L., Porter, B. W., et al. (1989). Concept learning and the problem of small disjuncts. In *IJCAI*, volume 89, pages 813–818.

Japkowicz, N. (2013). Assessment metrics for imbalanced learning. In He, H. and Ma, Y., editors, *Imbalanced learning*, pages 187–206. John Wiley & Sons.

Jo, T. and Japkowicz, N. (2004). Class imbalances versus small disjuncts. *ACM SIGKDD Explorations Newsletter*, 6(1):40–49.

Kotsiantis, S., Kanellopoulos, D., and Pintelas, P. (2006). Handling imbalanced datasets: A review. *Science*, 30(1):25–36.

Kotsiantis, S., Pintelas, P., Anyfantis, D., and Karagiannopoulos, M. (2007). Robustness of learning techniques in handling class noise in imbalanced datasets.

Lemaître, G., Nogueira, F., and Aridas, C. K. (2017). Imbalanced-learn: A Python Toolbox to Tackle the Curse of Imbalanced Datasets in Machine Learning. *Journal of Machine Learning Research*, 18(17):1–5.

Lichman, M. (2013). Uci machine learning repository.

Ma, L. and Fan, S. (2017). Cure-smote algorithm and hybrid algorithm for feature selection and parameter optimization based on random forests. *BMC bioinformatics*, 18(1):169.

MacQueen, J. (1967). Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297.

McCullagh, P. (1984). Generalized linear models. *European Journal of Operational Research*, 16(3):285–292.

Nekooeimehr, I. and Lai-Yuen, S. K. (2016). Adaptive semi-unsupervised weighted oversampling (a-suwo) for imbalanced datasets. *Expert Systems with Applications*, 46:405–416.

Nickerson, A., Japkowicz, N., and Milios, E. E. (2001). Using unsupervised learning to guide resampling in imbalanced data sets. In *AISTATS*.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in python. *Journal of Machine learning research*, 12:2825–2830.

Prati, R. C., Batista, G., and Monard, M. C. (2004). Learning with class skews and small disjuncts. In *SBIA*, pages 296–306.

Provost, F. (2000). Machine learning from imbalanced data sets 101. In *Proceedings of the AAAI'2000 workshop on imbalanced data sets*, volume 68, pages 1–3. AAAI Press.

Santos, M. S., Abreu, P. H., García-Laencina, P. J., Simão, A., and Carvalho, A. (2015). A new cluster-based oversampling method for improving survival prediction of hepatocellular carcinoma patients. *Journal of biomedical informatics*, 58:49–59.

Sculley, D. (2010). Web-scale k-means clustering. In *Proceedings of the 19th international conference on World wide web*, pages 1177–1178, 1772690. ACM.

Song, J., Huang, X., Qin, S., and Song, Q. (2016). A bi-directional sampling based on k-means method for imbalance text classification. In *Computer and Information Science (ICIS), 2016 IEEE/ACIS 15th International Conference on*, pages 1–5.

Weiss, G. M., McCarthy, K., and Zabar, B. (2007). Cost-sensitive learning vs. sampling: Which is best for handling unbalanced classes with unequal error costs? *DMIN*, 7:35–41.

A APPENDICES

A.1 IMPLEMENTATION OF K-MEANS SMOTE

The following listing shows the python implementation of k-means SMOTE. Documentation strings were omitted for brevity. The full implementation can also be found at `https://github.com/felix-last/kmeans_smote`. Program documentation is available at `https://kmeans-smote.readthedocs.io/en/latest/`.

```
# Authors: Felix Last
# License: MIT

import warnings
import math
import numpy as np

from sklearn.utils import check_random_state
from sklearn.metrics.pairwise import euclidean_distances

from imblearn.over_sampling.base import BaseOverSampler
from imblearn.over_sampling import SMOTE
from imblearn.over_sampling import RandomOverSampler
from imblearn.exceptions import raise_isinstance_error
from imblearn.utils import check_neighbors_object
from imblearn.utils.deprecation import deprecate_parameter

class KMeansSMOTE(BaseOverSampler):
    def __init__(self,
                 ratio='auto',
                 random_state=None,
                 kmeans_args={},
                 smote_args={},
                 imbalance_ratio_threshold=1.0,
                 minority_weight=0.66,
                 density_power=None,
                 use_minibatch_kmeans=True,
                 n_jobs=1):
        super(KMeansSMOTE, self).__init__(ratio=ratio, random_state=
            random_state)
        self.imbalance_ratio_threshold = imbalance_ratio_threshold
        self.kmeans_args = kmeans_args
        self.smote_args = smote_args
        self.random_state = random_state
        self.minority_weight = minority_weight
        self.n_jobs = n_jobs
        self.use_minibatch_kmeans = use_minibatch_kmeans
        self.density_power = density_power

    def _cluster(self, X):
        if self.use_minibatch_kmeans:
            from sklearn.cluster import MiniBatchKMeans as KMeans
        else:
            from sklearn.cluster import KMeans as KMeans

        kmeans = KMeans(**self.kmeans_args)
        if(X.shape[0] < kmeans.n_clusters):
            self.kmeans_args['n_clusters'] = X.shape[0]
            kmeans = KMeans(**self.kmeans_args)
```

```
        if self.use_minibatch_kmeans and 'init_size' not in self.
            kmeans_args:
            self.kmeans_args['init_size'] = min(2 * self.kmeans_args['
                n_clusters'], X.shape[0])
            kmeans = KMeans(**self.kmeans_args)

        kmeans.fit_transform(X)
        cluster_assignment = kmeans.labels_
        # kmeans.labels_ does not use continuous labels, i.e. some labels
            in 0..n_clusters may not exist. Tidy up this mess.
        return cluster_assignment

    def _filter_clusters(self, X, y, cluster_assignment,
        minority_class_label):
        # compute the shape of the density factors
        # since the cluster labels are not continuous, make it large enough
             to fit all values up to the largest cluster label
        largest_cluster_label = np.max(np.unique(cluster_assignment))
        sparsity_factors = np.zeros((largest_cluster_label + 1,), dtype=np.
            float64)
        minority_mask = (y == minority_class_label)
        sparsity_sum = 0

        for i in np.unique(cluster_assignment):
            cluster = X[cluster_assignment == i]
            mask = minority_mask[cluster_assignment == i]
            minority_count = cluster[mask].shape[0]
            majority_count = cluster[~mask].shape[0]
            imbalance_ratio = (majority_count + 1) / (minority_count + 1)
            if (imbalance_ratio < self.imbalance_ratio_threshold) and (
                minority_count > 1):
                distances = euclidean_distances(cluster[mask])
                non_diagonal_distances = distances[
                    ~np.eye(distances.shape[0], dtype=np.bool)
                ]
                average_minority_distance = np.mean( non_diagonal_distances
                    )
                if average_minority_distance is 0:
                    average_minority_distance = 1e-1 # to avoid division by
                    0
                density_factor = minority_count / (
                    average_minority_distance ** self.density_power)
                sparsity_factors[i] = 1 / density_factor

        # prevent division by zero; set zero weights in majority clusters
        sparsity_sum = sparsity_factors.sum()
        if sparsity_sum == 0:
            sparsity_sum = 1 # to avoid division by zero
        sparsity_sum = np.full(sparsity_factors.shape, sparsity_sum, np.
            asarray(sparsity_sum).dtype)
        sampling_weights = (sparsity_factors / sparsity_sum)

        return sampling_weights


    def _sample(self, X, y):
        self._set_subalgorithm_params()
```

```
# determine optimal number of clusters if none is given
if 'n_clusters' not in self.kmeans_args:
    labels, generate_counts = zip(*self.ratio_.items())
    minority_index, majority_index = np.argmax(generate_counts), np
        .argmin(generate_counts)
    minority_label, majority_label = labels[minority_index], labels
        [majority_index]
    minority_count = X[y == minority_label].shape[0]
    majority_count = X[y == majority_label].shape[0]
    if (self.minority_weight < 0) or (self.minority_weight > 1):
        raise ValueError('minority_weight should be between 0 and
            1. Got {}'.format(self.minority_weight))
    # n_clusters is set to weighted mean of minority and majority
        count
    self.kmeans_args['n_clusters'] = math.floor(
        (minority_count * self.minority_weight)
        + (majority_count * (1-self.minority_weight)))

if self.density_power is None:
    self.density_power = X.shape[1]

resampled = list()
for minority_class_label, n_samples in self.ratio_.items():
    if n_samples == 0:
        continue
    cluster_assignment = self._cluster(X)
    sampling_weights = self._filter_clusters(X, y,
        cluster_assignment, minority_class_label)
    smote_args = self.smote_args.copy()
    if np.count_nonzero(sampling_weights) > 0:
        # perform k-means smote
        for i in np.unique(cluster_assignment):
            cluster_X = X[cluster_assignment == i]
            cluster_y = y[cluster_assignment == i]
            if sampling_weights[i] > 0:
                # determine ratio for oversampling the current
                    cluster
                target_ratio = {label: np.count_nonzero(cluster_y
                    == label) for label in self.ratio_}
                cluster_minority_count = np.count_nonzero(cluster_y
                    == minority_class_label)
                generate_count = int(round(n_samples *
                    sampling_weights[i]))
                target_ratio[minority_class_label] = generate_count
                    + cluster_minority_count

                # make sure that cluster_y has more than 1 class,
                    adding a random point otherwise
                remove_index = -1
                if np.unique(cluster_y).size < 2:
                    remove_index = cluster_y.size
                    cluster_X = np.append(cluster_X, np.zeros((1,
                        cluster_X.shape[1])), axis=0)
                    majority_class_label = next( key for key in
                        self.ratio_.keys() if key !=
                        minority_class_label )
```

```
                    target_ratio[majority_class_label] = 1 +
                        target_ratio[majority_class_label]
                    cluster_y = np.append(cluster_y, np.asarray(
                        majority_class_label).reshape((1,)), axis=0)

                # modify copy of the user defined smote_args to
                    reflect computed parameters
                smote_args['ratio'] = target_ratio

                smote_args = self._validate_smote_args(smote_args,
                    cluster_minority_count)
                oversampler = SMOTE(**smote_args)

                # if k_neighbors is 0, perform random oversampling
                    instead of smote
                if 'k_neighbors' in smote_args and smote_args['
                    k_neighbors'] == 0:
                        oversampler_args = {}
                        if 'random_state' in smote_args:
                            oversampler_args['random_state'] =
                                smote_args['random_state']
                        oversampler = RandomOverSampler(**
                            oversampler_args)

                # finally, apply smote to cluster
                with warnings.catch_warnings():
                    # ignore warnings about minority class getting
                        bigger than majority class as this would
                        only be true within this cluster
                    warnings.filterwarnings(action='ignore',
                        category=UserWarning, message='After over-
                        sampling\, the number of samples \(.*\) in
                        class .* will be larger than the number of
                        samples in the majority class \(class #.*
                        \-\> .*\)')
                    cluster_resampled_X, cluster_resampled_y =
                        oversampler.fit_sample(cluster_X, cluster_y)

                if remove_index > -1:
                    # since SMOTE's results are ordered the same
                        way as the data passed into it, the
                        temporarily added point is at the same index
                         position as it was added.
                    cluster_resampled_X = np.delete(
                        cluster_resampled_X, remove_index, 0)
                    cluster_resampled_y = np.delete(
                        cluster_resampled_y, remove_index, 0)

                resampled.append( (cluster_resampled_X,
                    cluster_resampled_y) )

            else:
                # don't oversample this cluster, just add it to
                    resampled results
                resampled.append((cluster_X, cluster_y))
    else:
        # all weights are zero -> perform regular smote
```

```
                minority_count = np.count_nonzero( y ==
                    minority_class_label )
                warnings.warn('No minority clusters found for class {}.
                    Performing regular SMOTE. Try changing the number of
                    clusters. Recommended number of clusters: between {} and
                     the number of majority class instances.'.format(
                    minority_class_label, minority_count))

                smote_args = self._validate_smote_args(smote_args,
                    minority_count)
                oversampler = SMOTE(**smote_args)
                return oversampler.fit_sample(X, y)

        resampled = list(zip(*resampled))
        if(len(resampled) > 0):
            X_resampled = np.concatenate(resampled[0], axis=0)
            y_resampled = np.concatenate(resampled[1], axis=0)
        else:
            # no samples were generated because none were requested
            X_resampled = X.copy()
            y_resampled = y.copy()
        return X_resampled, y_resampled


    def _validate_smote_args(self, smote_args, minority_count):
        # determine max number of nearest neighbors considering sample size
        max_k_neighbors =  minority_count - 1
        # check if max_k_neighbors is violated also considering smote's
            default
        smote = SMOTE(**smote_args)
        if smote.k_neighbors > max_k_neighbors:
            smote_args['k_neighbors'] = max_k_neighbors
            smote = SMOTE(**smote_args)
        return smote_args

    def _set_subalgorithm_params(self):
        # copy random_state to sub-algorithms
        if self.random_state is not None:
            if 'random_state' not in self.smote_args:
                    self.smote_args['random_state'] = self.random_state
            if 'random_state' not in self.kmeans_args:
                self.kmeans_args['random_state'] = self.random_state

        # copy n_jobs to sub-algorithms
        if self.n_jobs is not None:
            if 'n_jobs' not in self.smote_args:
                    self.smote_args['n_jobs'] = self.n_jobs
            if 'n_jobs' not in self.kmeans_args:
                if not self.use_minibatch_kmeans:
                    self.kmeans_args['n_jobs'] = self.n_jobs
```

A.2 USAGE EXAMPLE OF K-MEANS SMOTE IMPLEMENTATION

The following listing shows a simple example of how the python implementation of k-means SMOTE can be used. Once the class KMeansSMOTE is instantiated with the desired parameters, a call to the object's fit_sample method is made to oversample the minority class.

```
import numpy as np
```

```
from imblearn.datasets import fetch_datasets
from kmeans_smote import KMeansSMOTE

datasets = fetch_datasets(filter_data=['oil'])
X, y = datasets['oil']['data'], datasets['oil']['target']

[print('Class {} has {} instances'.format(label, count))
for label, count in zip(*np.unique(y, return_counts=True))]

kmeans_smote = KMeansSMOTE(
  kmeans_args={
    'n_clusters': 100
  },
  smote_args={
    'k_neighbors': 10
  }
)
X_resampled, y_resampled = kmeans_smote.fit_sample(X, y)

[print('Class {} has {} instances after oversampling'.format(label, count))
for label, count in zip(*np.unique(y_resampled, return_counts=True))]
```

Expected output:

```
Class -1 has 896 instances
Class 1 has 41 instances
Class -1 has 896 instances after oversampling
Class 1 has 896 instances after oversampling
```