Exact and Approximate SMOTE Algorithms for Handling Imbalanced Big Data in PySpark

*Ting-Chun Chen
Computer Science
University of Nottingham
Nottingham, England
psxtc3@nottingham.ac.uk

*Hewen Pang
Computer Science
University of Nottingham
Nottingham, England
psxhp2@nottingham.ac.uk

*Thomas Cotter
Computer Science
University of Nottingham
Nottingham, England
psytc8@nottingham.ac.uk

***Abstract*—Imbalanced data is a common problem in machine learning, affecting both small and big datasets. In small datasets, SMOTE algorithms are typically used to handle this problem by synthesising samples to oversample the minority class. However, since SMOTE and its variants use the nearest neighbour algorithm to produce these synthetic samples, it doesn't inherently scale well to big datasets. To address this issue, we built four scalable SMOTE algorithms in PySpark and evaluated their performance using a Decision Tree classifier. The algorithms were tested on two imbalanced datasets. Our research found that these four methods work well in pySpark, and allow the SMOTE algorithm to be parallelized. Therefore, producing a fast and scalable SMOTE algorithm is important and provides benefit to the big data community.**

***Index Terms*—Big Data, Imbalanced Data, SMOTE, PySpark, ENN**

I. Introduction And Background

Data Scientists often have to handle imbalanced datasets, where the cardinality of one class is significantly lower than the other classes. This problem occurs often in medical or fraud datasets where the positive class like a cancer diagnosis or credit card fraud is infrequent. There are multiple ways of dealing with this, including under-sampling and over-sampling. Chawla et al [1] introduced the SMOTE algorithm, which oversamples the minority class using synthetic data. For each point in the minority class, a random neighbor is selected from its K (usually 5) nearest neighbors. The synthetic sample is then computed as a random point in the feature space between the original point and the randomly chosen neighbor. This process can be repeated to generate more synthetic samples as necessary. Chawla et al found that this method of oversampling the minority class combined with the undersampling of the majority class outperformed simply varying class priors in Naive Bayes.

Since the SMOTE algorithm was published, there have been many variants of it. AdaSYN (Haibo et al, [2]), is an approach similar to SMOTE, except that it generates a different number of samples depending on an estimate of the local distribution of the class to be oversampled. There is also BorderlineSMOTE (Han et al [3]), again which is similar to SMOTE, however only minority samples near the borderline are over-sampled. Furthermore, a combination of SMOTE and undersampling on the synthetic samples has been tested by Batista et al [4]. In their paper, they tested a combination of SMOTE & Edited-Nearest-Neighbors (ENN). ENN is an undersampling algorithm which essentially removes points which are misclassified by their nearest neighbors. The combination of these two algorithms yielded excellent results for handling imbalanced datasets.

However, these algorithms are not inherently scalable to big data. The nearest neighbors algorithm is computationally expensive, comparing each point with every other point in the dataset. This can be shown by trying running the SMOTE algorithm on the large 5,000,000 record dataset sequentially. Running this on DataBricks results in an out of memory error. We can see that a innovative, scalable SMOTE solution is required. Juez-Gil et al [5], developed Approx-SMOTE, a highly scalable version of SMOTE which uses approximate nearest neighbor search. Additionally, an exact version, SMOTE-BD was also introduced by Basgall et al [6], which uses a exact nearest neighbors algorithm. Both of these solution are written in Scala.

SMOTE-BD uses knn-IS, an spark-based KNN produced by Mailo et al [7]. This is implented as a classifier, in which the test set is broadcast to each map function to compare every test sample against the whole training set. This exact solution requires the test set to fit into memory, which works because the test set is usually the smaller of the two sets. This is also the case for SMOTE, as the minority set is always the smallest subset of the whole dataset. Determing the distance between any two instances is performed by calculating Euclidean distance between the pair.

This paper details our approach in writing four SMOTE variant algorithms for PySpark, the Python API for Apache Spark. The algorithms we implemented are:

- **Local SMOTE**: A local solution, which applies the imblearn library SMOTE to partitions of the dataset.
- **Approximate SMOTE**: A scalable solution, which uses an approximate nearest neighbor algorithm.
- **Exact SMOTE**: A scalable solution, which uses an exact nearest neighbor algorithm.
- **Exact SMOTE-ENN**: A ensemble solution which combines Exact-SMOTE with ENN.

The rest of this paper is structured as follows. Section II details the methodology and reasoning behind our approach. Sec-

tion III details the experiments performed, including metrics and baselines. Finally, Section IV will contain a description of the main findings from the research, including an interpretation of these results.

II. Proposed Methodology

This section details the algorithms behind our approaches and why we chose them. For each algorithm, we can perform speed and metric (defined in III) based comparisons to determine the most optimal algorithm. Doing this allows us to solve our problem of building a highly scalable accurate SMOTE solution for big data.

Before analysis, pre-processing of the data was required. As we are using Euclidean distance to determine the nearest neighbors, we needed to normalize the data. We used MinMax scaling for this, scaling every point to be between 0 and 1. Moreover, Euclidean distance does not perform well with categorical data. We encoded categorical features such that distances between the values were represented numerically. For example, 1 feature 'Vehicle_Age' contained the values: '<1 year', '1-2 years' & '>2years'. These were encoded as 0, 1, & 2 respectively. Any features which could not be encoded in this way were dropped from the dataset.

Once the nearest neighbors have been obtained, both 'Approximate SMOTE' and 'Exact SMOTE' generate the synethetic samples in the same way. We have implemented a UDF to do this, which uses vector maths to calculate a random point in the feature space between the point and it's neighbor. This UDF is the mapped to our DataFrame using a select statement. To synthesise any categorical features, the results are rounded to the closest category.

A. Local SMOTE

Initially, we developed a 'Local' solution to the problem. This involves applying SMOTE to partitions of the dataset and combining the results at the end. To do this, we used Imblearn, a python library for handling imbalanced datasets [8]. We chose to split our pySpark DataFrame into 10 sections, and applied Imblearns SMOTE-NC algorithm to each of those. SMOTE-NC is a variant of SMOTE that handles categorical data. Since this is a 'Local' solution, the suitability of this solution to our problem is not optimal, however it serves as a secondary baseline to compare to the 'Global' approaches we produced.

B. Approximate SMOTE

The first 'Global' solution developed was a pySpark version of Approx-SMOTE. This algorithm uses an approximate version of the nearest neighbor algorithm. In our solution, this is implemented using a Locality Sensitive Hashing (LSH) technique [9]. The general idea of LSH is to use a family of functions (LSH families) to hash data points into buckets, so that the data points which are close to each other are in the same buckets, while data points that are far away from each other are very likely to be in different buckets. More specifically, we are using Bucketed Random Projection, which is the LSH family for Euclidean Distance. This allows us to use ApproxSimilarityJoin to perform a self-join on this projection, which returns pairs of rows in the dataset whose distance is smaller than a user defined threshold. In our case this threshold is infinity as we want to check the approximate distance between all rows. This method allows us to scale nearest neighbours to big data and theoretically, without a loss of speed. This method is directly from the PySpark library so is highly optimised.

C. Exact SMOTE

The second 'Global' solution developed was a pySpark version of the Exact-SMOTE algorithm. To do this, we used a KDTree [10] data structure to perform the nearest neighbor search.

This tree is constructed by recursively partitioning the feature space into smaller regions. At each partition a dimension is chosen that effectively splits the feature space in half. This is done by choosing the median in that dimension, and splitting the points on either side of this median. This tree can be built in worse case O(k log(n)) time, where k is the number of dimensions. The data structure can then be queried by starting at the root node and descending through the tree, choosing the subtree closest to the query point. As this tree is balanced, we can prune almost half the data points at each split as it is likely that those points are not the nearest neighbor. Due to it's speed and efficiency, we have used a KDTree for the exact nearest neighbor algorithm. If a datasets contains a sufficiently large number of points compared to the number of dimensions, the KDTree data structure allows for a more efficient querying of nearest neighbors than the exhaustive search used in knn-IS and SMOTE-BD. Since we are dealing with big data, the number of points in the dataset is consistently large.

Initializing this KDTree structure is very fast, as seen in Fig. 1. This shows the time taken to initialise a KDTree with varying dataset sizes. Each of these datasets had 10 dimensions. We can see that a dataset with 1,000,000 points in 10 dimensions takes just over 4 seconds to initialise the KDTree. This is very fast. However, the bottleneck of the nearest neighbor search lies in repeatedly querying this tree to find the k Nearest Neighbors of each point in the minority class. This bottleneck is a good target for parallelization. To do this, we broadcasted the KDTree object to MapPartitions function. This greatly reduces the time taken to produce all the k nearest neighbors for a point. The diagram showing the data transfer for the exact nearest neighbors algorithm in Exact SMOTE can be seen in Fig. 2.

To use a KDTree, the tree and the minority set would both have to fit in memory, however this is true with SMOTE-BD as well. Therefore, any dataset that can be used with SMOTE-BD, can be implemented more efficiently with a version of SMOTE that uses a KDTree. Fortunately, spark allows for broadcast variables of up to 8GB.

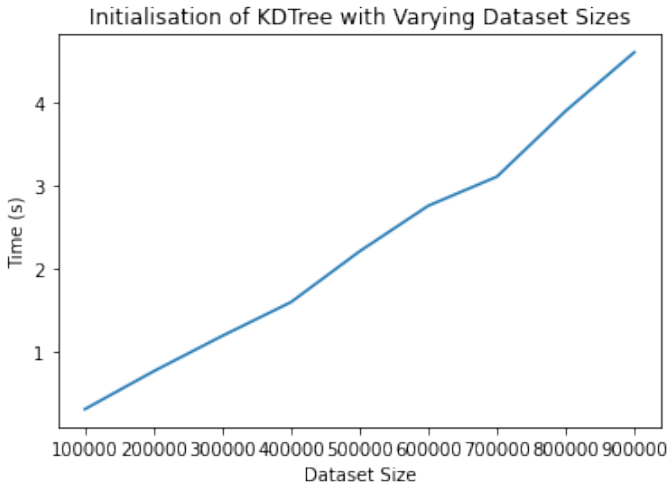

Fig. 1: KDTree Initialization

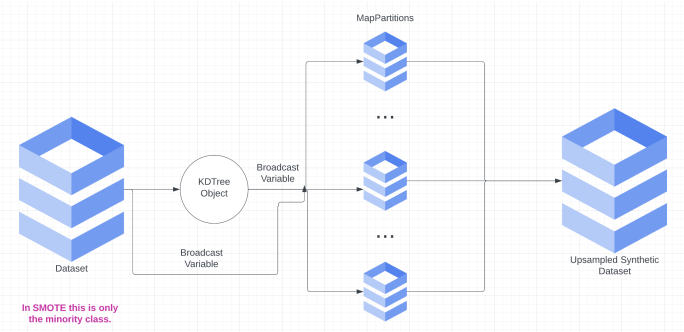

Fig. 2: NearestNeighbors Data Transfer

D. Exact ENN

The concept of ENNs was first proposed in 1972 [11]. We use this technique to both undersample the majority data before SMOTE and undersample the minority data after SMOTE. ENN is helpful when removing noise (useless and misleading information) from a input class (either minority or majority). It works by removing a point from the dataset if it misclassified by it's nearest neighbors (usually 3). For example, if a point is in the majority class, but all of it's 3 closest neighbors are in the minority class, it is removed from the dataset. We can roughly control the number of removed data points by adjusting the number of NNs (k), however, it can be tricky to control accurately. Our ENN implementation uses KDTree, for the same reasons as suggested in the Exact SMOTE section. However, in our implementation of ENN, only the target label has to be broadcasted to the mapPartitions function, rather than the feature vector. This is because we only need the target labels of each neighbor to determine if we should drop the point.

E. Classifiers

The chosen classification baseline for this project is a DecisionTree [12]. It is easy to use from the PySpark ml library, and it works well within the PySpark framework. Moreover, decision trees can automatically select the feature with the most distinguishing degree as the split, which can process discrete and continuous variables without any pre-processing operations such as feature scaling or feature conversion. Decision trees can also process data with missing values. Meanwhile, decision trees are a non-parametric learning method that does not depend on any assumptions and distributions and is suitable for various types of data. Ultimately, the new dataset generated by the SMOTE method will not affect the effect of the decision tree. It will still select distinguishing features as split points in the new synthetic data.

III. Experimental Set-Up

All experiments were performed on a databricks cluster with a driver (Standard_DS3_v2) and 10 workers (Standard_F4).

A. Datasets

We have used two datasets in our experiments. Both datasets are imbalanced, however one dataset is much smaller, allowing us to test our algorithms before applying them to the large dataset.

The first dataset used has 11 features and 382,154 rows, and is available on Kaggle [13]. Its purpose is to predict whether a customer would have an interest in Vehicle Insurance. The

features of the dataset include Demographics (gender, age, region code type), Vehicles (vehicle age, damage), Policy (premium, sourcing channel) and the target label 'Response'. Particularly, this target label is highly unbalanced, with 62,601 instances of label '1' - representing people who are interested in vehicle insurance and 319,544 instances of label '0' - representing people who are not interested at all. This results in an imbalance ratio of 4.9:1. The features 'Gender', 'Vehicle_Age', 'Vehicle_Damage' are strings, and therefore StringIndexer is used to pre-process the data and make sure that all the features are floats. We have dropped 3 features ('id', 'Region_Code, 'Policy_Sales_Channel') from the analysis as they are all IDs.

The second dataset is on Airline data and is available on Kaggle [14]. It has 28 features and 123,534,969 rows. However, for speed we test this dataset on random subsets of 5,000,000 and 1,000,000 rows. Assume the metrics are relative to 1,000,000 rows unless mentioned. The target label is 'Cancelled', which is 1 when the plane was cancelled and 0 was the plane was not cancelled. This label has an imbalance ration of 52:1. All ID features were dropped: 'CancellationCode', 'FlightNum', 'TailNum', & 'UniqueCarrier'. 'Dest' & 'Origin' were also dropped due to an inability to encode these values with distance. The dataset also contained missing values. These were either imputed with a constant value ('0') or mean imputation on the column. Please see the accompanying code for more details on this.

In the research, datasets used in all methods were divided into training sets and validation sets according to the same ratio of 0.7 and 0.3. All random seed parameters were set as 42 to ensure the same partitioning results. This was performed prior to the application of any SMOTE algorithm to prevent data leakage.

B. Metrics

In order to test the 'balance score' of the dataset, the research chooses Shannon Entropy [15]. Shannon Entropy is a measure of the amount of uncertainty or information content in a probability distribution. On a dataset of n instances, if you have k classes of sizes c_i, Shannon Entropy can be computed as $H = -\Sigma_{i=1}^{k} \frac{c_i}{n} log \frac{c_i}{n}$. This will tend towards 0 when the dataset is very imbalanced, and will tend towards $logk$ when the classes are balanced. Therefore, we can calculate a balanced score as $Balance = \frac{H}{logk}$, which will tend towards 1 for a balanced dataset.

In order to test if balancing the dataset improves classification metrics, the research chooses chooses the confusion matrix, which displays the number of true positives (TP), true negatives (TN), false positives (FP) and false negatives (FN) produced by the classifier. From this, we can calculate Accuracy ((TP + TN) / (TP + FN + FP + TN)), Precision (TP / (TP + FP)), Recall (TP / (TP + FN)), F1-Score ((2 * Precision * Recall) / (Precision + Recall)) and AUC to measure the effectiveness of the classifier. In particular, the Recall and Precision of minority labels will be studied. These metrics will be calculated for each of the algorithms we produced.

TABLE I: Baseline Metrics (Smaller Dataset)

Metric	Value
Balance Score	0.643
Accuracy	0.83
Precision (Minority Class)	0.0
Recall (Minority Class)	0.0
F1-Score	0.761
AUC	0.5

TABLE II: Baseline Metrics (Larger Dataset)

Metric	Value
Balance Score	0.134
Accuracy	0.998
Precision (Minority Class)	0.944
Recall (Minority Class)	0.999
F1-Score	0.998
AUC	0.999

The baselines for these metrics are subject to the original dataset before applying the SMOTE algorithm. They are determined from a DecisionTree and the balance score as previously described. The baselines for the smaller dataset can be seen in Table I. The baselines for the larger dataset be can seen in Table II:

C. Scalability

In addition to Exact-SMOTE being tested on how well it improves model performance on imbalanced datasets, we also want to test how well it scales to big data. In order to do this, we will time Exact-SMOTE with varying sizes of input dataset. These sizes are a fraction of the original dataset. They are: $\frac{1}{3}, \frac{1}{2}, 1$.

D. Models

We will be testing the classification results with a DecisionTree model. The DecisionModel is the base model imported from the pyspark.ml.classification library with default hyperparameters.

IV. Results and Discussion

In this section, we discuss our findings and results from this research. The results from both the smaller and larger dataset are presented here. We go into each algorithm in detail.

A. Local-SMOTE

TABLE III: Local-SMOTE Metrics

Dataset	Metric	Value
Smaller	Balance Score	1.0
	Accuracy	0.787
	Precision (Minority Class)	0.427
	Recall (Minority Class)	0.887
	F1-Score	0.812
	AUC	0.854
Larger	Balance Score	-
	Accuracy	-
	Precision (Minority Class)	-
	Recall (Minority Class)	-
	F1-Score	-
	AUC	-

We did not test the larmger dataset on Local-SMOTE due to memory & time restrictions.

B. Approx-SMOTE

TABLE IV: Approx-SMOTE Metrics

Dataset	Metric	Value
Smaller	Balance Score	1.0
	Accuracy	0.830
	Precision (Minority Class)	0.387
	Recall (Minority Class)	0.059
	F1-Score	0.774
	AUC	0.834
Larger	Balance Score	-
	Accuracy	-
	Precision (Minority Class)	-
	Recall (Minority Class)	-
	F1-Score	-
	AUC	-

We did not test the larger dataset on Approx-SMOTE due to memory & time restrictions. The Approx-SMOTE algorithm was interestingly not scalable. We believe that this results from using a threshold of 'infinity' in the approxSimilarityJoin() function.

C. Exact-SMOTE

TABLE V: Exact-SMOTE Metrics

Dataset	Metric	Value
Smaller	Balance Score	1.0
	Accuracy	0.827
	Precision (Minority Class)	0.356
	Recall (Minority Class)	0.065
	F1-Score	0.774
	AUC	0.832
Larger	Balance Score	0.476
	Accuracy	0.999
	Precision (Minority Class)	1.0
	Recall (Minority Class)	1.0
	F1-Score	0.999
	AUC	1.0

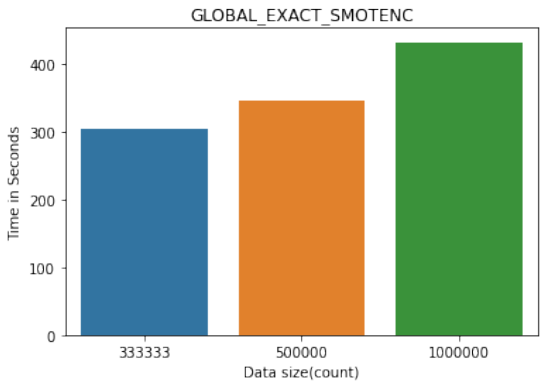

Fig. 3: Exact SMOTE Performance

Fig. 3 displays the time taken to run the Exact SMOTE algorithm with varying dataset sizes. The algorithm not only performs well on 1,000,000 samples (7 mins), it also scales well, taking an additional 120 seconds to process an extra 670,000 (420 seconds in total) samples compare to 300 seconds with 330,000 samples. The dataset with 5,000,000 records was also tested with Exact-SMOTE. The algorithm ran in **310.10** seconds.

D. Exact-SMOTE-ENN

TABLE VI: Exact-SMOTE-ENN Metrics

Dataset	Metric	Value
Smaller	Balance Score	0.859
	Accuracy	0.836
	Precision (Minority Class)	0.325
	Recall (Minority Class)	0.003
	F1-Score	0.762
	AUC	0.799
Larger	Balance Score	0.089
	Accuracy	0.995
	Precision (Minority Class)	0.999
	Recall (Minority Class)	0.734
	F1-Score	0.994
	AUC	0.870

The results in Table VI are from applying SMOTE then ENN (on the minority class) to both datasets. We can also apply ENN (majority) then SMOTE then ENN (minority) to the datasets. We applied this to the smalller dataset and the results are shown in Table VII. Doing this provided improvements on just the single ENN solution.

TABLE VII: Exact-ENN-SMOTE-ENN (Double ENN) Metrics

Dataset	Metric	Value
Smaller	Balance Score	0.996
	Accuracy	0.831
	Precision (Minority Class)	0.406
	Recall (Minority Class)	0.052
	F1-Score	0.772
	AUC	0.837

E. Discussion

In summary, Exact-SMOTE perform the best in terms of scalability. Neither Approx-SMOTE or Local-SMOTE could compete with the speed at which the nearest neighbors were calculated in the Exact-SMOTE algorithm. This makes sense for Local-SMOTE however is an interesting result for Approx-SMOTE. As mentioned, we believe this to be due to the threshold of 'infinity' in approxSimilarityJoin(). This is just not an efficient enough solution for a SMOTE algorithm.

In terms of classification performance, the Local-SMOTE performed the best followed by Exact-SMOTE. This is either due to how imblearn handles categorical data in their SMOTENC algorithm, or how they synthesise new results. The new synthetic data from the imblearn algorithm is likely to more closely match the original data than our algorithms, which makes the classification models perform better. All of our algorithms struggled with Recall performance on the minority class, which means they were still slightly overpredicting the majority class, even with the oversampled datasets.

Interestingly, ENN actually undersampled more data than SMOTE oversampled in the large dataset. The data itself might not be suitable to undersampling in this manner. It might help to apply ENN to the majority data several times to remove more majority points before applying SMOTE. We could also repeat ENN multiple times after SMOTE to remove any low quality synthetic samples. This could be done with more time on the project.

Using the KDTree proved to be a good idea, as it is much faster than the brute force method. The KDTree has some drawbacks however, such as not working well with categorical data. It also degrades in efficiency when the number of dimensions is higher compared to the number of points, although this isn't usually a problem in Big Data. Overall, broadcasting the KDTree proved to improve the SMOTE algorithm for Big data, and could potentially be used in a more widespread manner than this paper. Alternative tree structures such as Vantage point tree could alternatively be used for different distance metrics that suit more towards categorical data. However, in KNN-IS, when the test set is large, rather than it all being stored in memory, it is separated and fed into memory parts at a time. We did not implement the equivalent of this for SMOTE. However, with more time this is something we would like to look into.

V. Conclusion

From the results of the research, the SMOTE algorithm does have significant effect in the handling of imbalanced data. On the small dataset, Exact-SMOTE caused the precision rate of the decision tree to increase from 0 to 0.35. On the scalability test, the global Exact-SMOTE method was excellent. It processes a dataset with 5,000,000 data points, in 310 seconds. Although, this may be further improved as the Databricks cluster is a shared cluster, which means it may have been busy during the running of the tests. Therefore, this method designed in this research is a more rapid and scalable SMOTE method based on the pySpark framework.

This research makes up for the vacancy of the SMOTE method under the pySpark framework. Meanwhile, the four algorithms produced in this experiment can handle numerical and categorical data which can be mapped to a distance encoding. Finally, in this research Exact-SMOTE and Exact-ENN, both use the KDTree structure, resulting it greatly increased speed of returning proximity points and synthezing new data. In this era of booming big data technology, distributed data processing based on pySpark and larger data volume are inevitable. The method proposed in this research can be applied to the identification of credit card fraud, which is a extremely unbalanced dataset and the data volume is huge. The method propsed in this research is very suitable for distributed data processing and unbalanced data processing in this situation.

VI. Acknowledgements

The authors acknowledge the use of AI-assisted writing tools in the preparation of this paper. The AI provided suggestions that were reviewed and modified by the authors.

References

[1] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic minority over-sampling technique," *Journal of Artificial Intelligence Research*, vol. 16, pp. 321–357, jun 2002. [Online]. Available: https://doi.org/10.1613%2Fjair.953

[2] H. He, Y. Bai, E. A. Garcia, and S. Li, "Adasyn: Adaptive synthetic sampling approach for imbalanced learning," pp. 1322–1328, 2008. [Online]. Available: https://ieeexplore.ieee.org/document/4633969

[3] H. Han, W.-Y. Wang, and B.-H. Mao, "Borderline-smote: A new over-sampling method in imbalanced data sets learning," 2005. [Online]. Available: https://doi.org/10.1007/11538059_91

[4] G. Batista, R. Prati, and M.-C. Monard, "A study of the behavior of several methods for balancing machine learning training data," *SIGKDD Explorations*, vol. 6, pp. 20–29, 06 2004.

[5] M. Juez-Gil, Álvar Arnaiz-González, J. J. Rodríguez, C. López-Nozal, and C. García-Osorio, "Approx-smote: Fast smote for big data on apache spark," *Neurocomputing*, vol. 464, pp. 432–437, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231221012832

[6] M. J. Basgall, W. Hasperué, M. Naiouf, A. Fernández, and F. Herrera, "Smote-bd: An exact and scalable oversampling method for imbalanced classification in big data," *Journal of Computer Science and Technology*, vol. 18, no. 03, p. e23, Dec. 2018. [Online]. Available: https://journal.info.unlp.edu.ar/JCST/article/view/1122

[7] J. Maillo, S. Ramirez, I. Triguero, and F. Herrera, "knn-is: An iterative spark-based design of the k-nearest neighbors classifier for big data," *Knowledge-Based Systems*, 2016.

[8] Imblearn, "Imbalanced-learn," https://imbalanced-learn.org/stable/index.html, accessed: 2023-05-06.

[9] PySpark, "Locality sensitive hashing," =https://spark.apache.org/docs/2.2.3/ml-features.html, accessed: 2023-05-01.

[10] S. Learn, "Kdtree," https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KDTree.html, accessed: 2023-05-01.

[11] D. L. Wilson, "Asymptotic properties of nearest neighbor rules using edited data," *IEEE Transactions on Systems, Man and Cybernetics*, 1972.

[12] S. Patil and U. Kulkami, "Accuracy prediction for distributed decision tree using machine learning approach," *IEEE Xplore*, 2019.

[13] Kaggle, "Learning from imbalanced data," https://www.kaggle.com/datasets/arashnic/imbalanced-data-practice, accessed: 2023-05-01.

[14] Butler22, "Airline on-time performance data," https://www.kaggle.com/datasets/bulter22/airline-data, accessed: 2023-05-08.

[15] T. Eitrich, A. Kless, C. Druska, W. Meyer, and J. Grotendorst, "Classification of highly unbalanced cyp450 data of drugs using cost sensitive machine learning techniques," *Journal of Chemical Information and Modeling*, vol. 47, 2007.