Handling Class Imbalance in Big Data: A Comparative Study of Distributed Resampling Techniques in Apache Spark

Yash Shetye
Department of Computer Science
University of Nottingham
Nottingham, England
psxys21@nottingham.ac.uk

Shubhankar Shahade
Department of Computer Science
University of Nottingham
Nottingham, England
psxss45@nottingham.ac.uk

Abdullah Liaqat
Department of Computer Science
University of Nottingham
Nottingham, England
psxal13@nottingham.ac.uk

Ahmed Mustafa Bhangwar
Department of Computer Science
University of Nottingham
Nottingham, England
psxab22@nottingham.ac.uk

***Abstract*— Class imbalance is a significant challenge which can lead to degraded performance and biased model training. The study focuses on evaluating the efficiency and effectiveness of three resampling techniques namely under-sampling, over-sampling and SMOTE regarding both Spark's Global and Local distributed processing. To ensure comparability a fixed set of imbalanced ratios were tested using consistent partitioning strategy. Under-sampling offered the lowest resource usage and execution time as demonstrated in the results whereas over-sampling provided balanced trade-off. SMOTE incurs the highest computational cost. The findings highlight the importance of selecting a resampling method based on application constraints. Limitations include the exclusion of hyperparameter tuning and fixed partition size, chosen to control complexity.**

Keywords—SMOTE, under-sampling, over-sampling, class imbalance

I. Introduction

While an ideal dataset has a healthy, balanced distribution, that is not the case in real world. Datasets often exhibit often exhibit a highly imbalanced class distribution. Especially in many applications relating to fraud detection, medical diagnosis, and anomaly detection. The presence of imbalance often creates significant challenges. For example, in fraud detection, there might be 99% legitimate transactions and only 1% fraudulent ones. Machine Learning algorithms aim to maximize overall accuracy. Therefore, when the model learns to successfully predict the majority class, the accuracy can be high but the model still might not be good as it would have a poor predictive performance for the minority class.

While traditional resampling techniques such as oversampling, undersampling, and Synthetic Minority Over-sampling Technique (SMOTE) have been developed to address class imbalance, their scalability becomes a critical concern when applied to Big Data environments.

Big Data environment such as Apache Spark are designed to handle huge volumes of data at unparalleled speeds. As the reliance and use of Big Data in real world applications continues to grow, it is essential to adapt imbalance handling techniques to distributed computing frameworks.

To address this gap, this study investigates the following research questions:

1. For distinct levels of class imbalance which resampling technique namely under-sampling, SMOTE, oversampling scales best in Spark?

2. How is execution time, memory usage, and accuracy impacted by class imbalance ratio in Spark?

3. How are imbalanced datasets compared to balanced datasets handled in Spark?

For this study, the Credit Card Fraud Detection dataset [1] from Kaggle was used for testing different imbalance classification methods. The dataset is akin to real world and with previous works from other authors present, evaluation of the behaviour of sampling methods can prove to be easier to verify. The results of these studies can offer deep insights into designing and implementing scalable computationally and predictively efficient solutions for handling class imbalances in such environments.

II. LITERATURE REVIEW

While there have been several studies on the problem of class imbalance in Big Data, there is a lack of research in the scalability and efficiency of the resampling techniques proposed by previous studies in distributed environment, especially Apache Spark. This area warrants further exploration.

A similar study on the context of credit card fraud detection by evaluating the performance of class balancing techniques was done by Sisodia et al. (2017)[2]. Credit card fraud dataset are wrought with extreme class imbalance and hence resampling is almost always needed to enhance model performance. Studies by [2] demonstrated that but it also emphasized the need to balance predictive improvements against computational overheads. This study attempts to build on to the work done by Sisodia by extending research into distributed settings using Spark.

Domain experts part of the study done by He and Garcia (2009) [3] argue that there is not a strict definition as to what majority: minority ratio constitutes as a high-class imbalance despite suggesting that a high imbalance refers to scenarios where majority: minority class ratio ranges from 100:1 to 10000:1. Triguero et al. (2015) [4] further support this point as their study had imbalance ratios around 50:1 and yet it presented them with complex modeling challenges [4]. Reflecting upon the works and arguments presented by above studies, this paper investigates various imbalance ratios - 577:1 (original), 100:1, 50:1, 10:1, and 1:1—to systematically evaluate how varying degrees of imbalance affect both classification performance and computational scalability in a distributed environment.

According to Hasanin et al. (2019)[5], sampling methods such as oversampling, undersampling, and SMOTE can improve minority class representation, their effectiveness diminishes when applied on a large-scale dataset. The study highlighted that this occurred due to increased computing costs and complexities involving data distribution. This observation underscores the importance of evaluating how resampling methods perform when adapted for distributed frameworks such as Apache Spark, which is a key focus of the present study.

Current research attempts to address the gap noted by Leevy et al. (2018)[6] in empirical studies that assess how resampling strategies interact with distributed machine learning frameworks in terms of both predictive performance and system-level efficiency. [6] Surveyed methods for addressing high-class imbalance in Big Data and stressed the importance of scalable, distributed solutions. We address this gap by implementing and evaluating distributed oversampling, undersampling and SMOTE- based techniques in Apache Spark.

S. Salloum et al. [7] have done a similar approach for applying resampling techniques globally and locally within Spark partitions. They developed the Random Sample Partition (RSP) framework, which partitions Big Data into non- overlapping subsets for efficient sampling and ensemble learning.

Spark's performance can be enhanced by Optimizing task scheduling strategies. A recent survey by Li and Wen [8] analyzes various Spark scheduling strategy optimization techniques. Li and Wen emphasize the need for balancing scheduling overhead and data movement costs. The study [8] suggests that combining by parallelism, node resource evaluation and fault tolerance, higher priority tasks can be allocated more resources. This prevents shortages of resources and ensures tasks are executed efficiently. Our study's design lies on similar ideas of parallelism by varying Spark partitions and analysis of execution time and memory monitors resource usage.

Our current research adopts Random Forest as the base classifier in distributed Spark environment. Chen et al. [9] suggested a Parallel Random Forest (PRF) algorithm optimized for Big Data classification in Apache Spark. Their approach minimized data movement through vertical partitioning with the algorithm combining data- parallel and task- parallel strategies. This improved scalability and execution time while maintaining high classification accuracy. Our study reinforces the importance of evaluating different partitioning strategies to large imbalanced datasets.

III. METHODOLOGY

A. *Dataset description and preprocessing:*

The publicly available dataset related to credit card fraud detection is used in this study. The dataset consists of features which are already anonymized and obtained through the Principal Component Analysis (PCA) as mentioned in the data source. There are a total of 30 features labeled as V1 through V28, transaction amount and transaction time. The class variable represents whether the transaction is fraudulent (1) or non-fraudulent (0). The dataset is huge consisting of 284807 rows. The original dataset is highly imbalanced with only 492 rows belonging to fraudulent transactions and 284315 rows belonging to non-fraudulent making a non-fraud to fraud ratio of 577:1. It is also verified that dataset does not contain any missing values.

B. *Imbalanced dataset generation:*

To study the impact of imbalance ratio on the performance of a machine learning model multiple datasets are generated with varying imbalance ratio of majority to minority. The imbalanced ratios considered are 1:1, 10:1, 50:1, 100:1 and 577:1. The pie chart below visually represents the imbalance ratios:

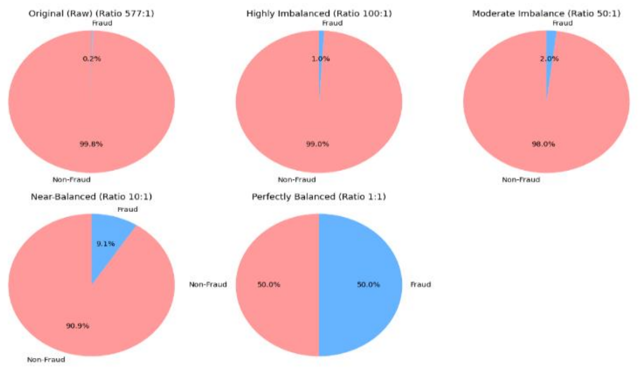

Figure 1: Class Distribution Across Different Imbalance Scenarios in Fraud Detection

C. *Resampling techniques:*

To rebalance the training data three different data level imbalance handle techniques were implemented from scratch using pyspark and Python. The techniques involve random under-sampling, random over-sampling and custom Synthetic Minority Over Sampling Technique.

1) Random Under-Sampling:

This technique involves deleting samples from the majority class to reduce the size and achieve the desired imbalance ratio. Python's sampling method is employed to keep the desired subset of majority class size and discard the rest. This technique has no impact on the minority class since the original size and kept. There is a risk of potentially important samples being discarded. To achieve a 1:1 ratio the majority class samples need to be reduced from 284315 to 492 (~99% reduction in majority class).

2) Random Over-Sampling:

This technique is based on duplicating samples from the minority class to increase its representation in the overall dataset. The minority class samples are duplicated using python's sampling method until the desired count is not reached. In this method all the original data is preserved specifically the majority class and its properties. However, the minority samples are duplicated which can increase the risk of overfitting since the information is being repeated. To achieve a 1:1 ratio the minority class needs to be increased from 492 to 284315 (~99% increase in minority samples).

3)Custom SMOTE:

In SMOTE the new minority samples are generated by the interpolation between the existing minority sample and one of the nearest neighbors. The SMOTE involves entire custom implementation using the k-NN. k-NN search was employed to search for the 5 nearest minority samples at the spark driver using the minority class feature vectors. To create a synthetic sample one of the 5 nearest neighbors was selected at random and feature values are interpolated along a line joining the selected minority instance and its neighbor. According to the standard SMOTE algorithm the interpolation factor used is a random number between 0 and 1. The number of synthetic samples needs to be generated depends on the imbalance ratio. However, generating the sample synthetically can increase the time drastically specifically for the cases where 1:1 ratio needs to be obtained among majority to minority ratio.
This approach can lead to bottleneck being driver-centric, however, guarantees consistency with original SMOTE.

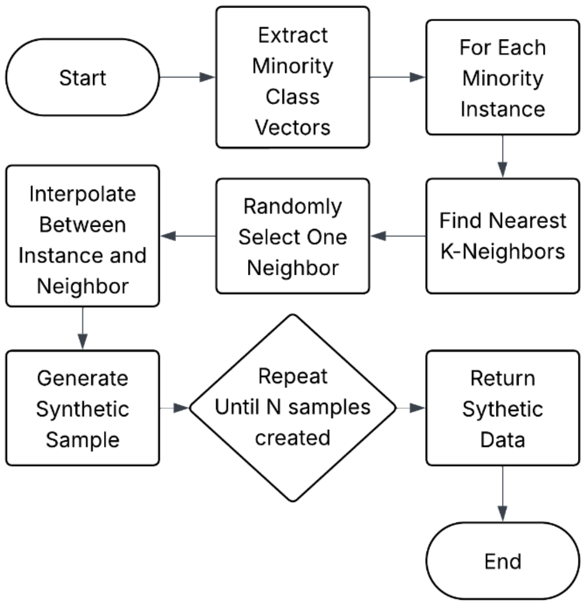
Start
Extract Minority Class Vectors
For Each Minority Instance
Find Nearest K-Neighbors
Randomly Select One Neighbor
Interpolate Between Instance and Neighbor
Generate Synthetic Sample
Repeat Until N samples created
Return Sythetic Data
End

Figure 2: Custom SMOTE implementation visualized using Flowchart

D. *Distributed Data Preparation and Partitioning Strategy:*

In Apache Spark parallel computation is done by distributing the data across various partitions. To evaluate the performance of various models under the varying techniques the number of partitions needs to be controlled to study the effect of partitions on model performance.

1) Repartitiong of data:

Spark's repartition was used to partition the dataset before model training and evaluation. The number of partitions varied among {2, 4, 8, 16} to assess how partition influences the performance:

- Execution time
- Memory usage
- Model evaluation performance

To maintain consistency in performance comparisons across various resampling techniques the partitioning logic is implemented in both the local and global distributed designs.

E. *Global vs. Local Distributed Learning Design:*

The performance and scalability of classification under class imbalance is assessed by implementing two distributed modeling techniques:

- Global model using Spark MLlib
- Local ensemble model via partition-wise training.

1) Global Model (Spark MLlib – Centralized Training over Distributed Data)

In global approach, a single model involving Spark MLlib's built in RandomForestClassifier was trained using the entire dataset. The dataset was repartitioned after applying the desired sampling technique and processed using Spark's distributed engine. Spark automatically distributes the work across worker nodes using internal pipelining and parallelism.
Since the data distribution is handled internally in spark the manual intervention for parallelism is minimal and offers efficient scaling for large datasets. However, such models may not be adopted well in the case of data skewness across partitions.

2) *Local Ensemble Models*

A custom partition-aware ensemble was implemented in a local distributed approach. Instead of training the single model on the entire dataset, each partition of data is considered as an independent subset and used to train separate Random Forest using Spark's mapPartitions().

Data was passed to local RandomForestClassifier from scikit-learn after converting into NumPy arrays. Model training was done by data belonging to each partition simulating localized learning. Majority voting was used to combine the predictions after collecting all the models to the driver. It allows scaling in truly decentralized environments. However, it can increase the cost related to memory and execution time due to training at local level.

F. *Integration of Spark UI for Performance Monitoring*

The impact of each distributed job on the system performance is obtained by incorporating low-level insights using Spark UI and event logging. The integration of spark UI was essential to deep dive into the performance metrics beyond the output of machine learning models. UI and event logging can provide valuable insights related to various stages of model training.

IV. EXPERIMENTAL STUDY AND RESULTS

A. *Execution Setup*

A single machine was used to conduct all the tasks. Spark driver and executor were set up in a laptop along with the following software and hardware configuration:

- Processor: AMD Ryzen 7 7435HS CPU (8 cores / 16 threads, up to 4.5 GHz).
- Software: 64-bit Windows 24H2, Python 3 (Anaconda distribution), and Apache Spark v3.5.1 (PySpark). Spark was installed via Anaconda and ran in local mode.
- Memory: 24 GB RAM.

To utilize the CPU's cores(master("local[8]")) we initialized a SparkSession in local mode with up to 8 parallel threads. To simulate a single node cluster environment with proper resource allocations the Spark configuration was tuned.

- spark.driver.memory = 4g (driver heap memory set to 4 GB)
- spark.executor.memory = 12g (executor heap memory set to 12 GB)
- spark.executor.memoryOverhead = 2g (off-heap overhead memory of 2 GB for executor)
- spark.sql.shuffle.partitions = 400 (shuffle partitions set to 400 for Spark SQL operations)

The above settings made sure that the Spark application had sufficient memory to operate the data and any shuffle operations additionally we could also capture in depth runtime metrics. A huge number of shuffle partitions (400) were selected to handle potential shuffling in MLib algorithms and data sampling, avoiding any single partition from becoming congested in case of vast data shuffles.

To validate the number of tasks launched for each stage we used the Spark UI and event logs. Also made sure the amount of data was read and written during shuffles and storage usage on the executor. We confirmed that our memory settings were honored and that there was no unexpected spilling to disk or garbage collection problems by examining the UI's environment and executor tabs.

B. *Procedure:*

To assess the interplay between class imbalance handling techniques, data partitioning configurations and distributed learning strategies, we ran a comprehensive series of experiments. We judged three resampling methods namely Random Oversampling, Random Under sampling and SMOTE across 2 learning paradigms which consists of a local approach where each partition trained its own model using scikit-learn and mapPartitions() and a global approach leveraging Spark's built-in RandomForestClassifier. For every technique, datasets were resampled to simulate 5 class imbalance ratios (1:1, 10:1, 50:1, 100:1, and 577:1 (original). After resampling, to observe the effects of parallelism, the data was partitioned again using Spark's repartition () method with partition counts set to 2,4,8 and 16. To train a Random Forest model using the respective global or local strategy each configuration was implemented. The model's performance was then tested using AUC, Accuracy and F1 Score. Execution time was recorded programmatically. Additional performance metrics such as memory usage, task duration and shuffle read/write volumes were extracted from the event logs and Spark UI. For comparative analysis and visualization all outputs were systematically exported. For making sure fairness and reproducibility throughout trails the experiments were conducted under constant memory settings.

C. *Results:*

Due to the involvement of nearest neighbor to generate synthetic data SMOTE has the highest execution time among all the considered imbalance ratios. Since under-sampling involves the removal of data leading to decrease in input data size and hence fastest execution time. Local approach has consistently taken more time specifically for SMOTE. Global under-sampling has outperformed other techniques in terms of the execution time whereas local SMOTE is the least efficient. The graph shows the execution time trends calculated using the time difference in python and not Spark UI. The number of partitions is fixed at 4 and execution time is plotted against varying imbalance ratios.

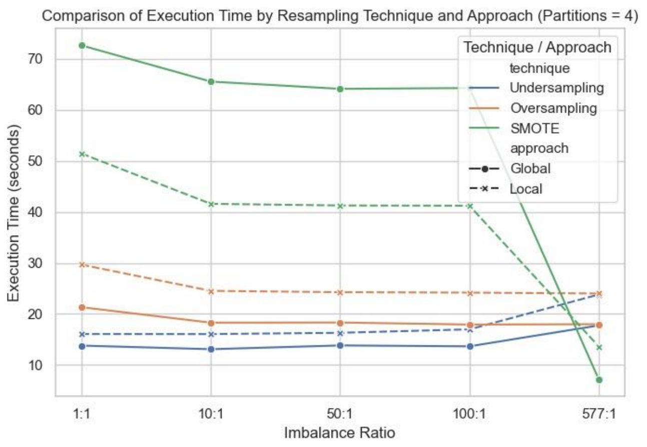

Figure 3: Comparison of Execution Time by Resampling technique and approach using Python.

The time execution is also obtained from the event log file of Spark and then analyzed to compare the results obtained through the code. SMOTE has shown an increased execution time for imbalance ratio of 1:1 during both approaches which shows sensitivity to increased synthetic data. Under-sampling remained the best performing method in both the charts and similarly other insights are also comparable which ensure that execution time obtained in both approaches is accurate.

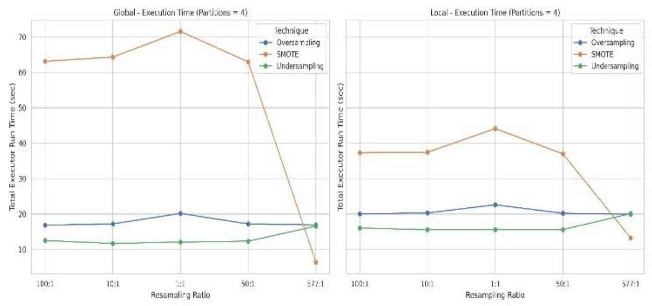

Figure 4: Comparison of Execution Time by Resampling technique obtained using Spark UI event log.

The comparison chart for f1-score has shown key differences between resampling techniques by varying class imbalance ratios. The resampling techniques involving under-sampling, over-sampling and SMOTE have shown high values for f1 even when the data is highly imbalanced. Due to the limited partition level data with increasing imbalance ratios local approach involving under-sampling and over-sampling has shown a sharp decrease in f1-score. This shows that the global approach is more robust and resilient towards the high imbalance.

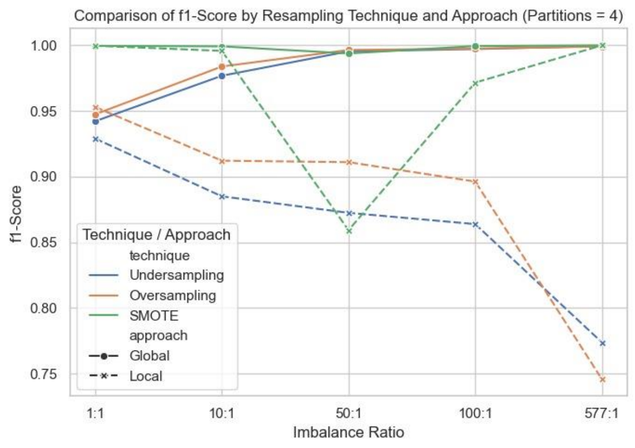

Figure 5: f1-score comparison by resampling techniques and approach with 4 partitions.

Across varying imbalance ratios, the global and local approaches have demonstrated clear distinction in performance as shown by accuracy chart. In the global setting SMOTE has shown almost perfect accuracy followed by the under-sampling. The accuracy in global approach remained almost consistent even for high class imbalance (577:1) however for local approaches accuracy dropped significantly with high class imbalance. This can happen due to the presence of the limited data for training among partitions.

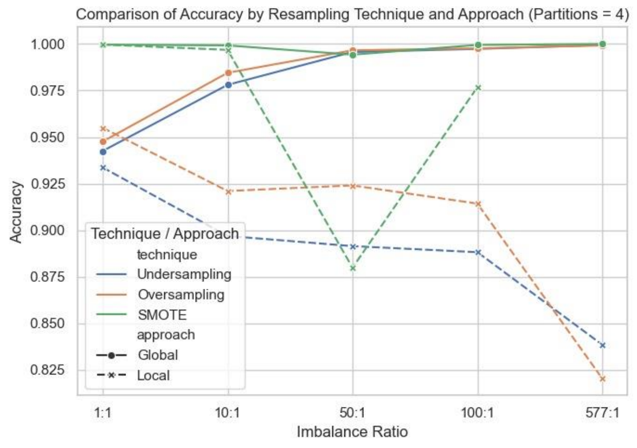

Figure 6: Accuracy comparison by resampling techniques and approach with 4 partitions.

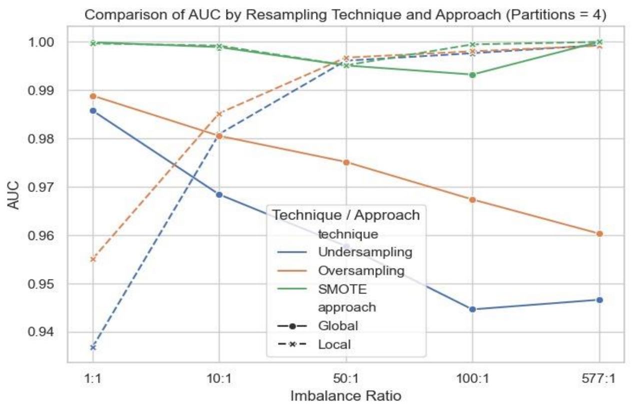

Figure 7: AUC comparison by resampling techniques and approach with 4 partitions.

SMOTE is the most effective technique as highlighted by AUC comparison, constantly achieving the highest scores across local and global models. Global SMOTE got outperformed by local SMOTE at high imbalance levels, indicating better minority class focus. Under sampling works well generally yet declines with higher imbalance, although oversampling struggles because of overfitting. AUC ranges differ from accuracy and F1, providing in depth insight into robustness under imbalance.

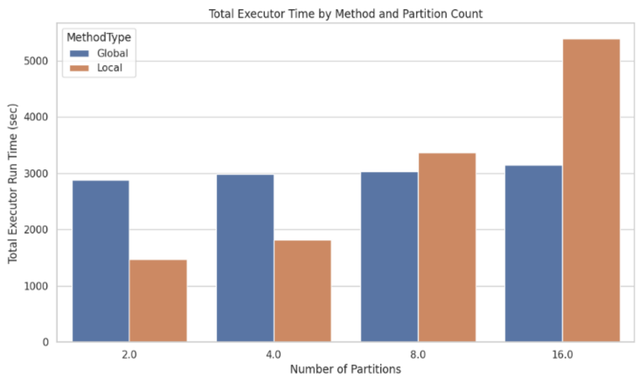

Figure 8: Total Executor Time by Approach and Partition count

According to figure 8, Global methods have consistency in scaling up across increasing partition counts with only a small increase in execution time. However, Local Methods seem to perform much better with reduced partition counts but at 16 partitions there is a steep increase which is most likely due to task scheduling or resource contention overhead. Overall global methods are stable under parallelism whereas local methods might degrade with increased partitions.

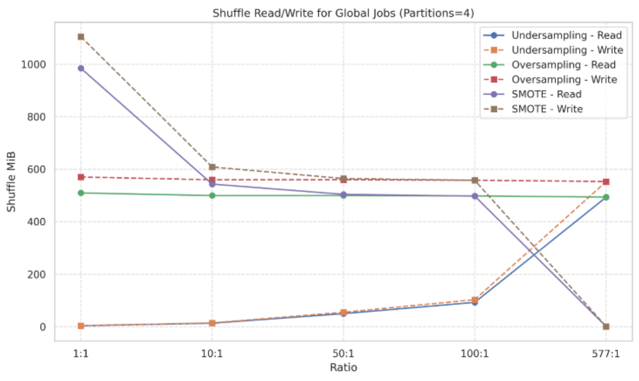

Figure 9: Shuffle read/write comparison by resampling techniques and global approach with 4 partitions.

Figure 9 shows that SMOTE has the greatest shuffle read/write volume at the lower imbalance ratios 1:1 and 10:1, therefore indicates significant data movement and computational overhead. Oversampling appears to remain moderately constant across all the different ratios, showing stable shuffle usage. Under-sampling technique has a smaller shuffle at low ratios but has a sharp increase at the ratio 577:1 caused by higher relative minority sampling. SMOTE has a sharp decrease at the ratio of 577:1 which implies reduced synthetic data generation. Compared with the other techniques, SMOTE overall is the most shuffle intensive technique globally, especially at normal imbalance levels.

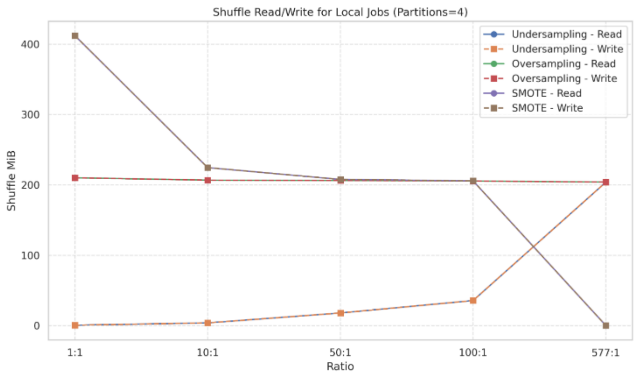

Figure 10: Shuffle read/write comparison by resampling techniques and local approach with 4 partitions

In the local approach, SMOTE has the greatest Shuffle read/write at the lowest imbalance ratio 1:1, but this decreases as the ratios are increased. Oversampling appears to have the most stable shuffle across all the rations which indicates predictable overhead. Under-sampling has the lowest shuffle at lowest ratio (1:1) but the shuffle increases as rations are increased or higher imbalance, like the global approach. The convergence of all techniques near 577:1 implies reduced shuffle needs at extreme ratios. Considering all techniques, SMOTE has the most shuffle early on and the Under-sampling technique appears to be the most efficient for local processing

V. DISCUSSION

The visualizations offer insights into the experimental results especially into the trade-offs between computational cost and classification performance for the different resampling strategies we implemented under class imbalance, using a fixed partition count of four.

In terms of Execution Time and Scalability, it is observed from the execution time plot as shown in Fig. 4 that SMOTE has the highest computational cost, especially in the global approach where the execution time exceeds that of oversampling and under sampling consistently. Upon increasing the number of partitions as seen in Fig. 68the scalability of Local methods worsens. In the SMOTE technique, execution time rises very sharply at 16 partitions. On the other hand, global methods have more consistency in their execution times across all partitions.

The Shuffle Read/Write Overhead analysis further validates our observations. Fig. 9 and Fig. 10 show that Global SMOTE tops the chart for shuffle read/write volumes suggesting heavy data movement. In contrast, under sampling shows negligible shuffle activity, confirming its lightweight nature. In places where network I/O is a concern, Local methods are more suitable as they demonstrate lower shuffle overhead than Global methods in distributed settings.

While under sampling outperforms others in execution time, scalability and shuffle overhead, in terms of Model Performance SMOTE (Global) regularly outperforms others with F1-scores and accuracy exceeding 0.99 at most imbalance ratios. This can be shown in Fig. 5, Fig. 6 and Fig. 7. SMOTE maintains a near-perfect AUC of 1.0 across all ratios in Fig. 7 indicating excellent class separability. Oversampling performs competitively in moderate imbalance settings but shows a drop in accuracy and F1-score at high imbalance (577:1).

Lastly, while the Global approach generally yields better predictive performance, it has a higher shuffle and execution time. This is exactly the opposite of the Local approach. However, Local approaches also tend to underperform at extreme imbalance ratios, especially for SMOTE and Oversampling.

No hyperparameter tuning was performed for the classification models due to computational limitations. Default parameters were used consistently across all configurations. Due to the large number of combinations to compare the results the number of partitions was fixed at 4, which may not fully generalize the results.

SMOTE creates synthetic samples by interpolation that might not always showcase realistic minority class instances, specifically at extremely high imbalance ratios. This could alter models' generalization, despite high F1 scores or AUC

VI. CONCLUSIONS

This study evaluates the impact of three resampling techniques—under-sampling, Oversampling, and SMOTE with both Global and Local approach across different Imbalance ratios in a distributed Spark Environment. The results revealed that under sampling has the lowest shuffle and execution time overall therefore making it the most computationally efficient for large scale imbalance applications. SMOTE, while having the highest F1-Scores, AUC and accuracy has highest processing cost which is evident in the local approach with high partition counts. Oversampling has a much more balanced performance and processing cost.

From a practical point of view, choosing the correct resampling technique involves a tradeoff between accuracy and efficiency. For real-time or resource-constrained applications, under sampling is a better choice, whereas SMOTE might be better suited for offline training scenarios where model performance is important. The number of partitions are set as 4 to provide fair and balanced view for all the techniques while having manageable processing cost.

Future work should improve the analysis by incorporating Hyperparameter tuning, testing and use of multiple datasets as this would provide much more in-depth analysis.

REFERENCES

[1] Kaggle, "Credit Card Fraud Detection Dataset," *Kaggle*. [Online]. Available: https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud. [Accessed: May 5, 2025].

[2] D. S. Sisodia, N. K. Reddy, and S. Bhandari, "Performance evaluation of class balancing techniques for credit card fraud detection," in *Proc. IEEE Int. Conf. Power, Control, Signals and Instrumentation Engineering (ICPCSI)*, 2017, pp. 839–844, doi: 10.1109/ICPCSI.2017.8392219.

[3] H. He and E. Garcia, "Learning from imbalanced data," *IEEE Trans. Knowl. Data Eng.*, vol. 21, no. 9, pp. 1263–1284, Sept. 2009, doi: 10.1109/TKDE.2008.239.

[4] I. Triguero, S. Rio, V. Lopez, J. Bacardit, J. Benítez, and F. Herrera, "ROSEFW-RF: The winner algorithm for the ECBDL'14 big data competition: An extremely imbalanced big data bioinformatics problem," *Knowl.-Based Syst.*, vol. 87, pp. 69–79, Sept. 2015, doi: 10.1016/j.knosys.2015.06.014.

[5] T. Hasanin, T. M. Khoshgoftaar, J. L. Leevy, and R. A. Bauder, "Severely imbalanced Big Data challenges: Investigating data sampling approaches," *J. Big Data*, vol. 6, no. 1, pp. 1–20, Nov. 2019, doi: 10.1186/s40537-019-0274-4.

[6] J. L. Leevy, T. M. Khoshgoftaar, R. A. Bauder, and N. Seliya, "A survey on addressing high-class imbalance in big data," *J. Big Data*, vol. 5, no. 1, pp. 1–29, Nov. 2018, doi: 10.1186/s40537-018-0151-6.

[7] S. Salloum, R. Dautov, X. Chen, P. X. Peng, and J. Zomaya, "Big Data Analytics on Apache Spark: A Survey," *Knowledge and Information Systems*, vol. 56, no. 2, pp. 493–524, 2018.

[8] C. Li and X. Wen, "A Survey of Spark Scheduling Strategy Optimization Techniques and Development Trends," *Computers, Materials & Continua*, vol. 78, no. 3, pp. 3319–3338, 2025. doi: 10.32604/cmc.2025.063047.

[9] J. Chen, K. Li, Z. Tang, K. Bilal, S. Yu, C. Weng, and K. Li, "A Parallel Random Forest Algorithm for Big Data in a Spark Cloud Computing Environment," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 4, pp. 919–933, Apr. 2017.