

**Credit Card Fraud Detection using Parallel and Distributed Computing techniques**

**High Performance Parallel Machine Learning and AI**



**Credit Card Fraud Detection using Parallel and Distributed Computing techniques**

## **Introduction**

### Background:

As digital payments become increasingly popular, the convenience they bring is often accompanied by rising concerns about fraud payments. Lots of people have suffered from the fraud payments. Having entered their card details during an online transaction, but later discovered unauthorized transactions on their accounts. Fraudulent activities would not only result in financial losses to bank account holders, but also weaken trust and reliability in digital payment systems. It’s a critical thing to address such issues, to ensure the safety and reliability of online financial transactions.

### Motivation:

With the increasing volume and frequency of bank transactions, it’s impractical and inefficient to detect fraudulent payments manually. Both individuals and financial institutions are facing heightened risks of fraud, particularly as high-value transactions are becoming more prevalent. Real-time detection of fraudulent activities is essential for banks to protect their customers and maintain trust among account holders. However, traditional local computing systems struggle to handle the vast and bulky volume of real-time transaction data efficiently.

To overcome these limitations, we plan to use parallel computing techniques and scalable parallel computing frameworks, including Dask, Joblib and multiprocessing pool, combined with advanced machine learning models, to offer a powerful and cost-effective solution. This approach enables efficient processing of large-scale consumer data, and enhances the accuracy and speed of fraud detection. By integrating these technologies, the banking industry can better identify fraudulent activities, minimize financial losses for both clients and businesses, and improve the overall security of financial systems.

### Goal:

The goal of this project is to develop a bank fraud detection system using parallel computing and machine learning models, to identify fraudulent transactions effectively. We aim to implement parallel computing techniques for data loading and preprocessing, train our machine learning model with optimized parameters, and evaluate the training efficiency by comparing different numbers of CPUs and several parallelization strategies.

## **Methodology**

The goal of this study is to use parallel computing techniques to detect fraudulent transactions. We would work on data loading, data visualization, data preprocessing, machine learning model training, and finally integrate parallelism into the machine learning model training.

## Data Loading, Visualization and Preprocessing with Parallelization:

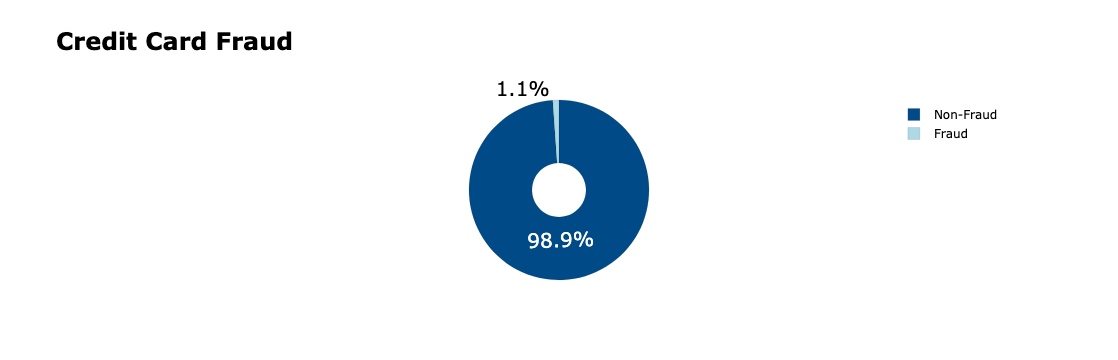
### Data Loading

We loaded our data into a Dask DataFrame. Dask Dataframe is a flexible, parallel computing library designed for handling large-scale data by operating on chunks of data in parallel, reducing memory usage and speeding up computations. Dask is very useful when the dataset is too big, or when the computation is too slow and other techniques don’t work.[1]

### Data Visualization and Exploratory Data Analysis(EDA)

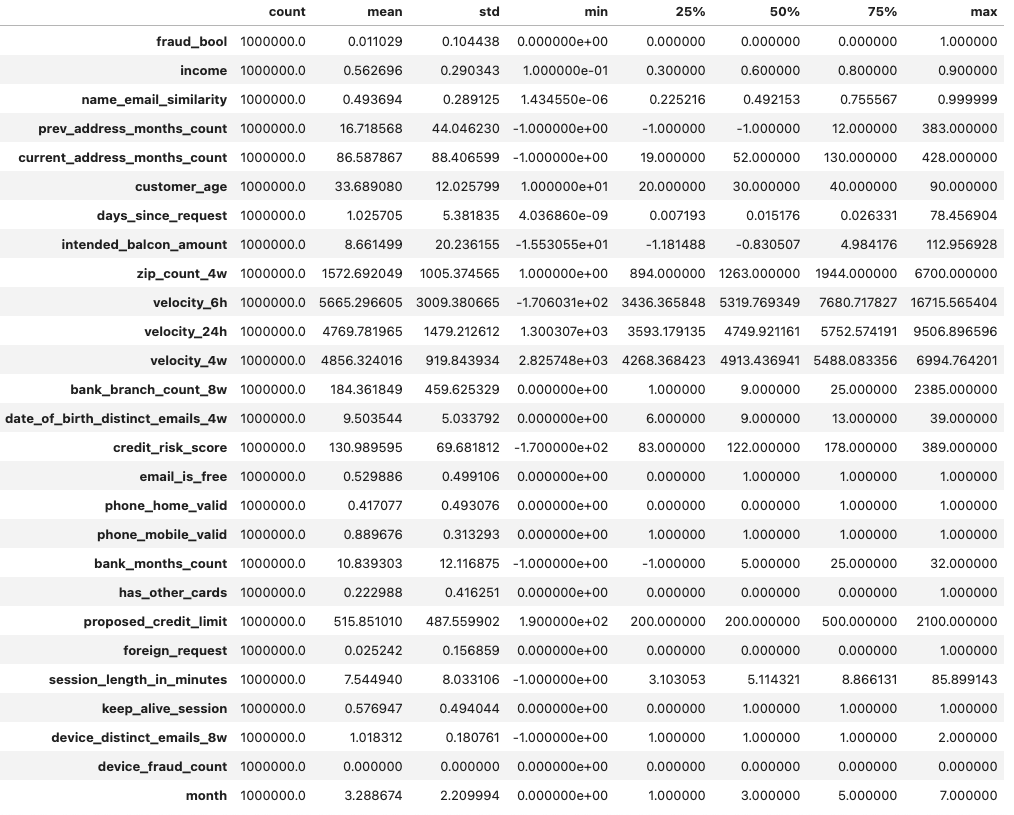
#### Statistics on Class Distribution

To understand the distribution of positive and negative samples in the dataset, we introduced Plotly to visualize the class distribution, which provided a clear and interactive representation of the number of positive (fraudulent) and negative (non-fraudulent) samples. It is shown in the graph that we have 98.9% non-fraud samples, and only 1.1% fraud samples, so it’s a strongly imbalanced dataset.



#### Summary Statistics of Features

We utilized the describe function in pandas to get the summary statistics for each column in the dataset, including the counts, mean, standard deviation, minimum value, maximum value, and quartiles for numerical features in 25%, 50%, and 75%, to help us better understand the data’s structure and identify potential outliers.



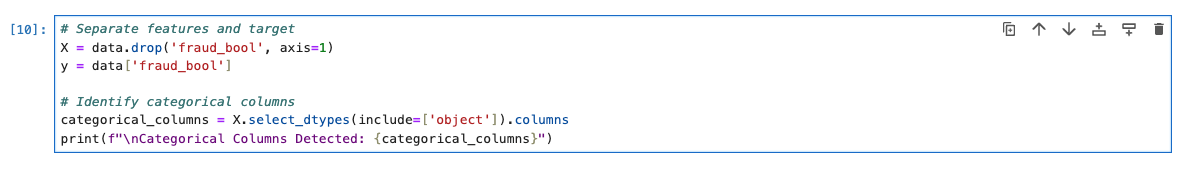
### Data Preprocessing

In the preprocessing step, we observed that the dataset contained no null values, so we could skip handling missing data. However, there were non-numeric columns that required encoding to convert categorical data into numerical data. Additionally, the dataset was highly imbalanced, and it’s required to introduce a resampling strategy to balance the data, ensuring the model could effectively distinguish between the target classes. In our project, we applied Random under-sampling to balance the minority and majority classes at a ratio of 1:2.

We worked on splitting dataframe dimensions, encoding non-numeric columns, handling class imbalance, scaling the features, and finally split our dataset into training tests and testing sets.

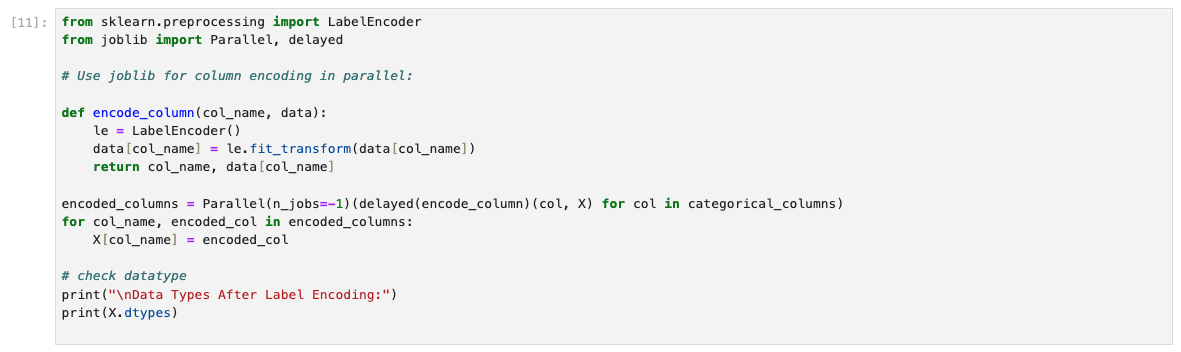
#### **Dataframe Dimensions**:

We started by splitting the dataset into features (X) and target (y) variables to prepare for further processing.



#### Encoding Non-Numeric Columns

Non-numeric columns in the dataset were encoded with LabelEncoder, transforming categorical values into numeric representations. Also, we integrated with Joblib to parallelize the encoding process. The encoding was achieved using the following approach:



#### Handling Class Imbalance

A highly imbalanced dataset can hinder the training process and result in a model that struggles to effectively differentiate between classes, particularly the minority class. Our dataset originally contained 1,000,000 samples, but only a few of them were positive samples. To address this problem, we applied Random Under Sampling (RUS) in the imbalanced-learn library, to rebalance the classes in our dataset.

RandomUnderSampler class is used to perform random under-sampling, and it can under-sample the majority class(es) by randomly picking samples with or without replacement.

*sampling\_strategy* parameter represents the ratio when resampling the data set. When the passed-in value is float, it corresponds to the desired ratio of the number of samples in the minority class over the number of samples in the majority class after resampling.[2]

When handling the imbalanced targets, we set the sampling\_strategy to 0.5, which means the number of fraud samples and non-fraud samples had a ratio of 1:2. Random Under Sampler retains all samples of the minority class and randomly selects a corresponding number of samples from the majority class.. This approach ensured the machine learning model had enough samples to effectively learn what fraud samples look like.

We also experimented with another method, SMOTE (Synthetic Minority Over-sampling Technique), from the imbalanced-learn library.[3] However, SMOTE generates synthetic samples for the minority class, and when the minority class has very few samples, these synthetic samples tend to be highly similar to each other. Another issue with SMOTE is that it increases the number of minority class samples to balance the dataset, which can result in a significant surge in the size of the minority class. This posed a problem for our dataset, as the original class distribution consisted of only 1.1% minority samples and 98.9% majority samples. Utilizing SMOTE would have doubled the size of our dataset, which ultimately slowed down the training process. Due to the large size of the dataset and the lack of distinctiveness among the samples in the minority class, SMOTE is not the most suitable method for addressing class imbalance.

Before applying the Random Under Sampler, our dataset consisted of 988,971 non-fraud samples and 11,029 fraud samples. After applying the Random Under Sampler, the non-fraud samples were reduced to 22,058 non-fraud samples, and there’s still 11,029 fraud samples.

#### Scale the Features

To scale the features, we used StandardScaler in the scikit-learn library to normalize them to a standard distribution with a mean of 0 and a standard deviation of 1. We did the feature scaling with parallelism by splitting it into chunks using numpy and applied the scaling process to each chunk in parallel with Joblib. After scaling, the chunks were recombined into a single dataset, ensuring both efficiency and consistency in the normalization process.

#### Split the Training Set and Testing Set

In our project, we splitted our dataset by setting 80% of the samples as the training set and 20% as the test set. After splitting the dataset, we checked the data shapes to ensure the process was completed correctly.



## Machine Learning Model Training

In our machine learning model training process, we focused on finding the optimal parameters, training the model with these best parameters, and evaluating its performance.

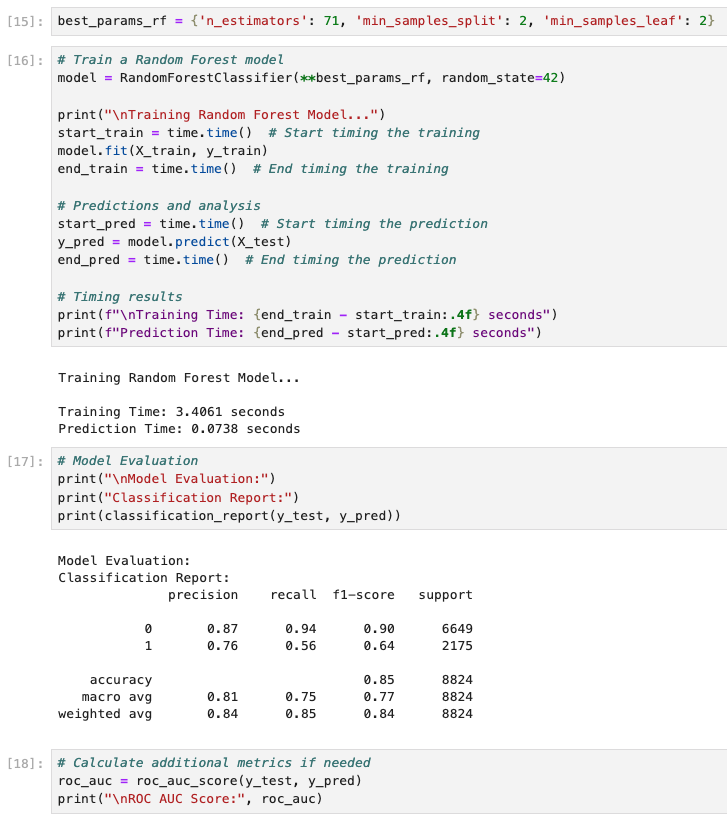
We introduced Optuna combined with GPU acceleration to efficiently search for the optimal parameters, used Random Forest as our base machine learning model, and evaluated the model's performance using the ROC-AUC score and F1 score.

### 2.1. Finding Optimal Parameter

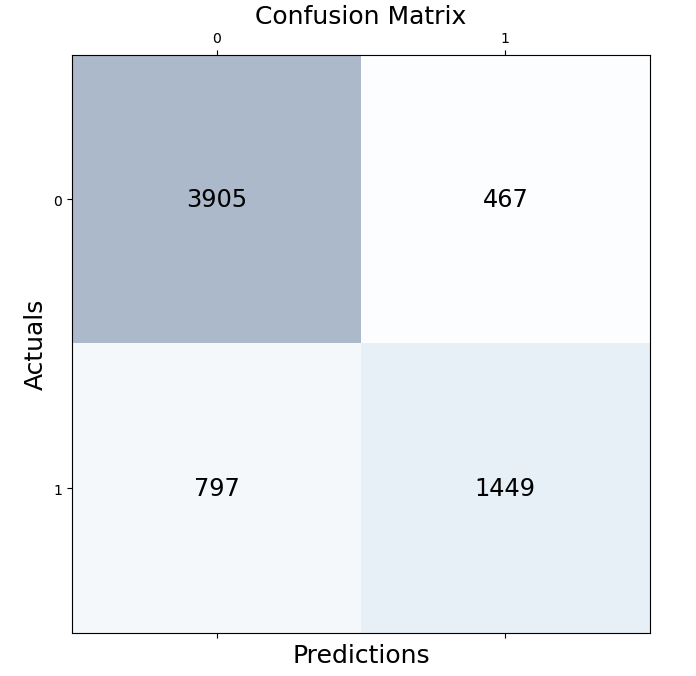
We utilized the Optuna library for hyperparameter optimization, performing this task on Google Colab with CUDA acceleration. Optuna is an open-source framework designed to automate hyperparameter tuning. While its primary goal is to enhance accuracy, it also allows users to specify a list of metrics to optimize. In our case, the hyperparameter search focused on maximizing the ROC-AUC score and F1 score, as these metrics are particularly important for evaluating performance on an imbalanced dataset.

### 2.2 Model Training & Evaluation

We selected the Random Forest as our model, and passed in our best parameters from the result of Optuna. After Training the model, we checked the model’s precision, recall rate, f-1 score and ROC-AUC score.



· We plotted the confusion matrix to visualize the classification results, showing 3,905 true negatives (TN), 1,449 true positives (TP), 467 false positives (FP), and 797 false negatives (FN). Our model performs well in identifying non-fraudulent transactions (TN) and detecting many fraudulent ones (TP). However, the 797 false negatives indicate a significant number of missed fraud cases, highlighting the need to improve recall, as undetected fraud could have serious consequences. In contrast, false positives (normal transactions incorrectly flagged as fraud) would primarily result in customer service intervention. Additionally, the 467 false positives indicate room for reducing unnecessary alerts. Balancing precision and recall will be critical for further optimizing the model.



## Parallelize the Machine Learning Model Training and Performance Evaluations

In this section, we introduced three methods of parallelization: Joblib, Python's Multiprocessing Pool, and Dask. We trained the Random Forest Classifier using 1, 2, 4, and 8 CPUs with each method to assess their performance and efficiency. After we finished the training, we evaluated these methods by analyzing the training time, calculating the speedup, and measuring the efficiency across different CPU configurations.

### Parallelize the Machine Learning Model Training

We utilized Joblib, Python's Multiprocessing Pool, and Dask to train the model in parallelism, and trained the model with 1, 2, 4 and 8 CPUs.

Joblib is a Python library designed for providing lightweight pipelining in Python, particularly for tasks involving large data processing or computationally intensive workloads. It provides an easy-to-use interface for parallelizing loops, functions, and processes. Joblib supports two main backends: loky and threading. loky is recommended to run functions that manipulate Python objects. threading is a low-overhead alternative that is most efficient for functions that release the Global Interpreter Lock. [5]

In our work, we used the loky backend for parallelization, as loky is ideal for our task, which involves computationally intensive model training with the Random Forest Classifier, as it efficiently utilizes multiple CPUs without being constrained by the GIL.

Multiprocessing is a package that supports spawning processes using an API similar to the threading module. The multiprocessing package offers both local and remote concurrency, effectively side-stepping the Global Interpreter Lock by using subprocesses instead of threads. Due to this, the multiprocessing module allows the programmer to fully leverage multiple processors on a given machine. It runs on both POSIX and Windows. [6]

In our work, we utilized Python's multiprocessing module to parallelize the training of the Random Forest Classifier. The multiprocessing module supports spawning processes, allowing us to bypass Python's Global Interpreter Lock (GIL) and fully leverage multiple CPUs for our computations.

Dask is a parallel computing library in Python that enables scalable computation for large datasets. It is designed to handle workloads that exceed the memory of a single machine, supporting both multithreading and multiprocessing, as well as distributed computing across clusters. Dask can build task graphs for computations, dynamically scheduling and optimizing tasks based on dependencies.

In our work, we set the number of Dask workers equal to the number of available CPUs during training. We divided the training and testing datasets into several chunks and distributed them across the workers, and each worker was assigned to train the model on a single core, ensuring an efficient parallelization process.

### Performance Evaluations

#### Timing

|  | Joblib | Multiprocessing | Dask |
| --- | --- | --- | --- |
| 1 CPU | 3.63 sec | 3.64 sec | 4.46 sec |
| 2 CPUs | 3.03 sec | 2.40 sec | 4.49 sec |
| 4 CPUs | 2.25 sec | 1.79 sec | 4.59 sec |
| 8 CPUs | 1.98 sec | 1.53 sec | 4.76 sec |

#### Speedup

As Dask has an increasing time cost when it’s training with more CPU, we only compare the speedup and efficiency when our model is trained under Joblib and Multiprocessing.

|  | Joblib | Multiprocessing |
| --- | --- | --- |
| 1 CPU | 1.0 | 1.0 |
| 2 CPUs | 1.19 | 1.52 |
| 4 CPUs | 1.61 | 2.04 |
| 8 CPUs | 1.83 | 2.38 |

#### Efficiency

|  | Joblib | Multiprocessing |
| --- | --- | --- |
| 1 CPU | 1.0 | 1.0 |
| 2 CPUs | 0.60 | 0.76 |
| 4 CPUs | 0.40 | 0.61 |
| 8 CPUs | 0.23 | 0.30 |

From the results above, we observed that Joblib and Multiprocessing perform better when training the model with more CPUs, while Dask does not show the same improvement. Overall, Multiprocessing performs better than Joblib, as it shows a higher speedup and better efficiency.

## Description of dataset

To implement this into practice, we are using the following Kaggle Dataset: Bank Account Fraud Dataset Suite (NeurIPS 2022). It is a classic example of classification problems. The in-depth information about the data is mentioned below:

• **income (numeric):** Annual income of the applicant (in decile form). Ranges between

[0.1, 0.9].

• **name\_email\_similarity (numeric):** Metric of similarity between email and applicant’s

name. Higher values represent higher similarity. Ranges between [0, 1].

• **prev\_address\_months\_count (numeric):** Number of months in previous registered

address of the applicant, i.e. the applicant’s previous residence, if applicable. Ranges

between [−1, 380] months (-1 is a missing value).

• **current\_address\_months\_count (numeric):** Months in currently registered address of

the applicant. Ranges between [−1, 429] months (-1 is a missing value).

• **customer\_age (numeric):** Applicant’s age in years, rounded to the decade. Ranges

between [10, 90] years.

• **days\_since\_request (numeric):** Number of days passed since application was done.

Ranges between [0, 79] days.

• **intended\_balcon\_amount (numeric):** Initial transferred amount for application.

Ranges between [−16, 114] (negatives are missing values).

• **payment\_type (categorical):** Credit payment plan type. 5 possible (annonymized)

values.

• **zip\_count\_4w (numeric):** Number of applications within same zip code in last 4 weeks.

Ranges between [1, 6830].

**• velocity\_6h (numeric):** Velocity of total applications made in last 6 hours i.e., average

number of applications per hour in the last 6 hours. Ranges between [−175, 16818].

**• velocity\_24h (numeric):** Velocity of total applications made in last 24 hours i.e., average

number of applications per hour in the last 24 hours. Ranges between [1297, 9586]

**• velocity\_4w (numeric):** Velocity of total applications made in last 4 weeks, i.e., average

number of applications per hour in the last 4 weeks. Ranges between [2825, 7020].

**• bank\_branch\_count\_8w (numeric):** Number of total applications in the selected bank

branch in last 8 weeks. Ranges between [0, 2404].

**• date\_of\_birth\_distinct\_emails\_4w (numeric):** Number of emails for applicants with

same date of birth in last 4 weeks. Ranges between [0, 39].

**• employment\_status (categorical):** Employment status of the applicant. 7 possible

(anonymized) values.

**• credit\_risk\_score (numeric):** Internal score of application risk. Ranges between

[−191, 389].

**• email\_is\_free (binary):** Domain of application email (either free or paid).

**• housing\_status (categorical):** Current residential status for applicant. 7 possible

(anonymized) values.

**• phone\_home\_valid (binary):** Validity of provided home phone.

**• phone\_mobile\_valid (binary):** Validity of provided mobile phone.

**• bank\_months\_count (numeric):** How old is the previous account (if held) in months.

Ranges between [−1, 32] months (-1 is a missing value).

**• has\_other\_cards (binary):** If the applicant has other cards from the same banking company.

**• proposed\_credit\_limit (numeric):** Applicant’s proposed credit limit. Ranges between

[200, 2000].

**• foreign\_request (binary):** If the origin country of request is different from the bank's country.

**• source (categorical):** Online source of application. Either browser (INTERNET) or

app (TELEAPP).

**• session\_length\_in\_minutes (numeric):** Length of user session in banking website in

minutes. Ranges between [−1, 107] minutes (-1 is a missing value).

**• device\_os (categorical):** Operative system of device that made request. Possible values

are: Windows, macOS, Linux, X11, or other.

**• keep\_alive\_session (binary):** User option on session logout.

**• device\_distinct\_emails (numeric):** Number of distinct emails in banking website from

the used device in last 8 weeks. Ranges between [−1, 2] emails (-1 is a missing value).

**• device\_fraud\_count (numeric):** Number of fraudulent applications with used device.

Ranges between [0, 1].

**• month (numeric):** Month where the application was made. Ranges between [0, 7].

**• fraud\_bool (binary):** If the application is fraudulent or not.

## Results and Analysis

For model training and predictions through multiple processors, we have used Discovery cluster with the following specifications –

1. Memory – 64GB
2. CPUs – 8
3. Python version - 3.8
4. Miscellaneous Python Libraries - see requirements.txt
5. Google Colab - connected to T4 GPU

### Part 1: Traditional Machine Learning Models Analysis

#### Random Forest Classifier

In our Random Forest Classifier without parallelism, we trained the model using a single core. The training time was 2.63 seconds, and the prediction time was 0.06 seconds. The model achieved a ROC-AUC score of 0.77, with an F1 score of 0.86 for non-fraud and 0.70 for fraud. These results indicate that while the model performs reasonably well, it still falls short in accurately detecting fraudulent transactions.

**Interpretation:**

As we resampled the dataset, its size significantly decreased from the original 1,000,000 samples (over 200 MiB) to only 33,087 samples. This reduction in data size contributed to the fast training time of 2.63 seconds, even without parallelism. However, this smaller dataset may also limit the model’s ability to generalize to unseen data, as it has fewer samples to learn from.

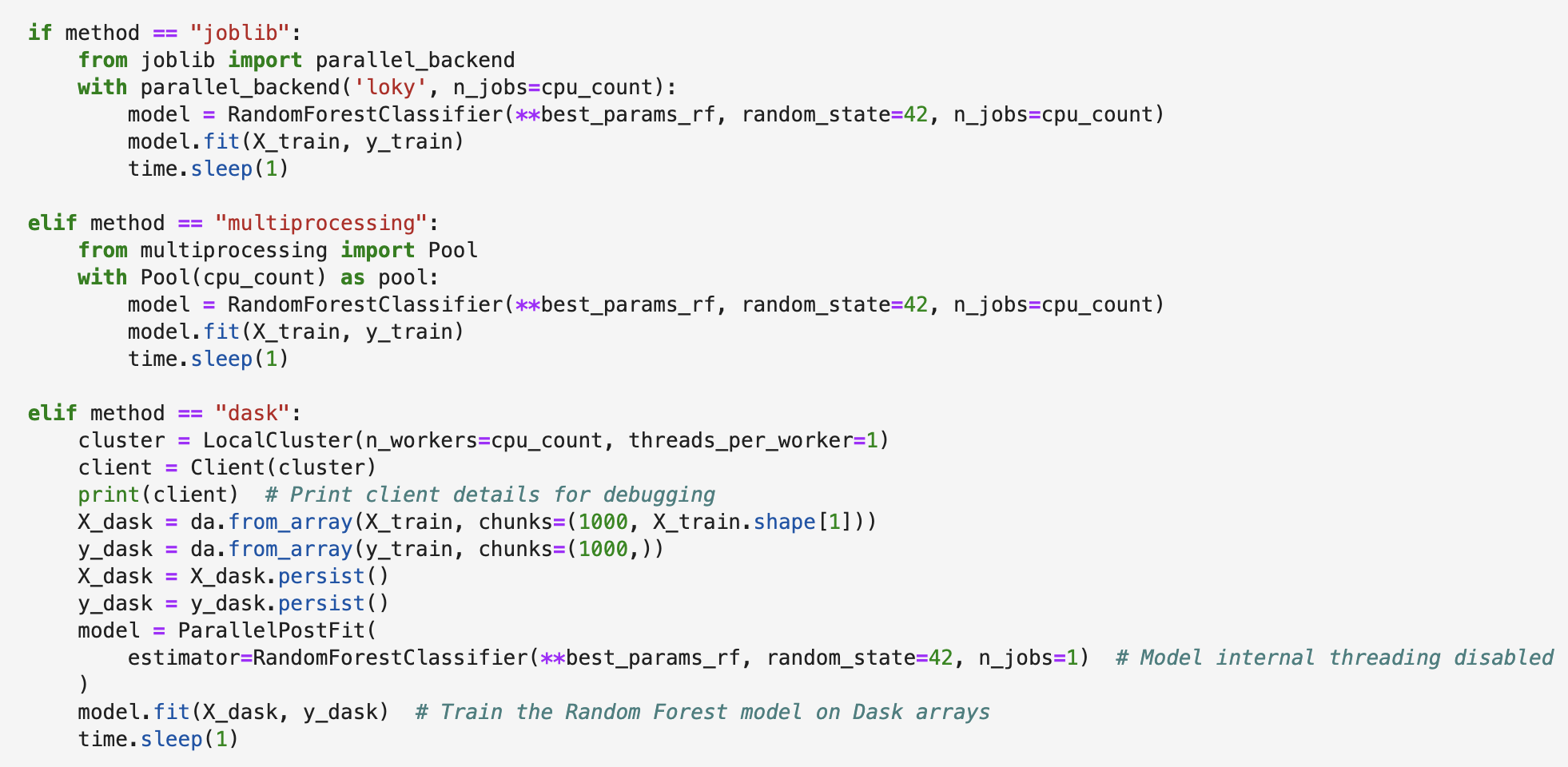
While the model achieved a reasonable ROC-AUC score of 0.77, the F1 scores for non-fraud (0.86) and fraud (0.70) indicate that it performs better on the majority class (non-fraud) than on the minority class (fraud). This imbalance in performance suggests that the model still struggles with accurately detecting fraudulent transactions, and it’s likely due to the class imbalance in the original dataset.

Further optimization, such as hyperparameter tuning, the use of advanced resampling techniques, or ensemble methods, should focus on improving the recall for fraud detection while maintaining the overall performance on non-fraudulent transactions. Additionally, to use SMOTE Boost, which involves integrating random forest classifiers with SMOTE, is also a potential choice..

### Part 2: Parallelization in Machine Learning Model Training

Training machine learning models, particularly ensemble-based ones such as Random Forests, requires significant computational resources due to the creation and aggregation of multiple decision trees. Parallelization offers a means to expedite this process by distributing computations across numerous CPUs or nodes. This study evaluated three parallelization methods—Joblib, Multiprocessing, and Dask—for their performance in training a Random Forest Classifier on a large, balanced dataset.

## Framework Overview



Parallel frameworks enable the simultaneous execution of tasks, such as building individual decision trees in a Random Forest across multiple CPU cores.

**Joblib:** Widely integrated into the Scikit-learn library, provides a straightforward API for leveraging multi-core processing through backend configurations.

**Multiprocessing:** Utilizes Python’s multiprocessing library to spawn processes that independently execute tasks on separate CPUs.

**Dask:** A more sophisticated solution for parallelism and distributed computing, suitable for larger datasets that exceed a machine’s memory.

## Experimental Setup

This analysis compares these methods by varying the number of CPUs (1, 2, 4, and 8) to observe their scalability and training efficiency. Each framework is assessed based on training time, speedup, and efficiency.

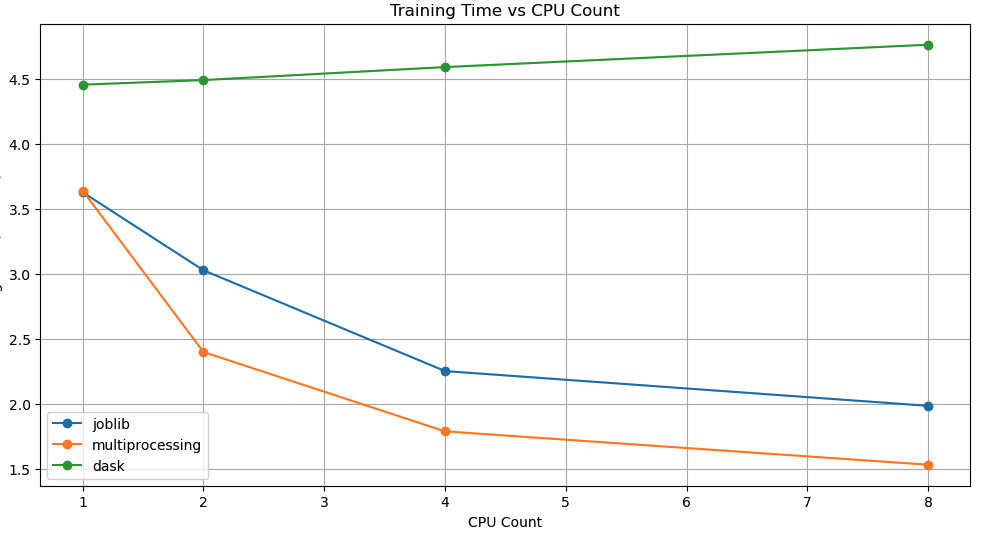
### Model Hyperparameters

To ensure fairness, the same hyperparameters were used across all frameworks. These parameters, tuned for optimal performance, included:

* **n\_estimators=75**: Number of decision trees in the forest.
* **max\_depth=None**: Unlimited tree depth to allow for maximum model flexibility.
* **min\_samples\_split=4** and **min\_samples\_leaf=1**: Control over tree splits to avoid overfitting.

### Part3: Results and Analysis

#### Training Time Comparison

****

Joblib:

* Training time significantly decreased as CPU cores increased, showcasing near-linear scalability.
* At 8 cores, training time reached approximately 1.98 seconds, demonstrating efficient resource utilization.

Multiprocessing:

* Similar to Joblib, training time decreased consistently with added CPU cores.
* Multiprocessing outperformed Joblib, achieving the shortest training time of 1.53 seconds at 8 cores, highlighting optimized parallelism.

Dask:

* Unlike Joblib and Multiprocessing, Dask showed scalability challenges.
* Training time unexpectedly rose to 4.76 seconds at 8 cores, indicating inefficiencies in resource management. This distinct behavior warrants a separate discussion.

#### Speedup Comparison

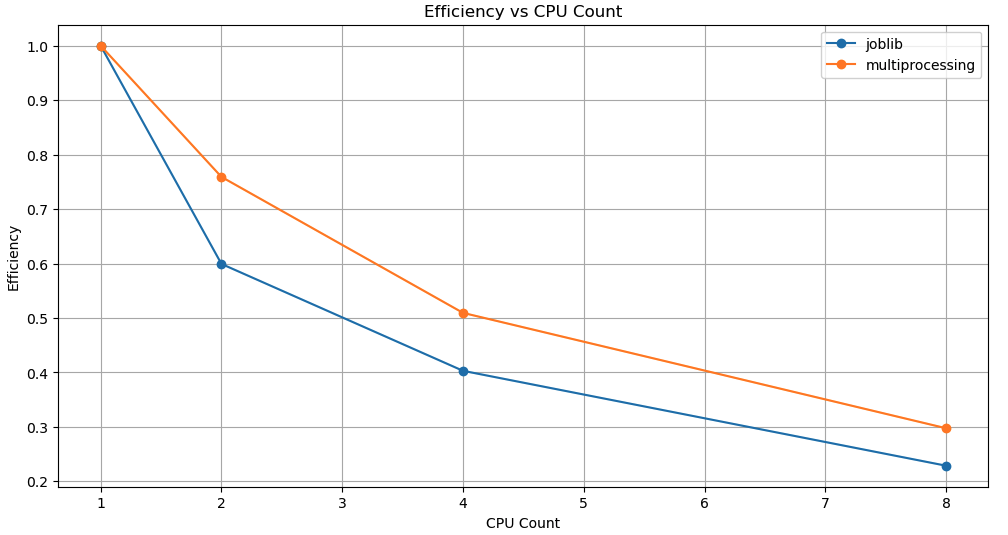
#### 

Speedup measures how well the training time improves as more CPUs are added:

Both **Joblib** and **Multiprocessing** demonstrated near-linear speedup, with **Multiprocessing** achieving a speedup of 2.39x at 8 cores.

**Multiprocessing** outperformed **Joblib** slightly in terms of speedup due to optimized process-level parallelism.

#### Efficiency Comparison



**Multiprocessing** maintained higher efficiency across all configurations, suggesting optimal CPU utilization.

**Joblib** efficiency dropped as the CPU count increased but still performed reasonably well.

### Observing Dask Performance

Dask’s performance is analyzed separately due to its unique design and operational challenges when compared to Joblib and Multiprocessing. Unlike the latter two frameworks, which focus on local parallelism, Dask is designed for distributed computing and larger-than-memory datasets. However, in this specific local setup with smaller data partitions, its overhead becomes significant.

#### Training Time

Dask displayed slower performance and scalability issues, particularly as the number of cores increased, where training time rose unexpectedly to 4.76 seconds at 8 cores. This anomaly is likely caused by additional overhead in coordinating distributed tasks.

#### Scalability and Overhead

**Task Scheduling Overhead:** Dask dynamically manages tasks using a Directed Acyclic Graph (DAG). With more CPUs, coordination overhead grows, impacting runtime for smaller workloads.

**Shared File System Bottlenecks:** Warnings highlight Dask workers using network storage for scratch data. This introduces latency, especially as worker count increases.

**Inefficient Partitioning:** Poorly optimized or imbalanced data partitions result in higher communication costs and underutilized CPUs.

### Scalability and Overhead

The results demonstrate that Joblib and Multiprocessing scale efficiently with the number of CPU cores, as evidenced by the sharp decline in training time. Dask struggled with scalability in this specific setup:

**Joblib and Multiprocessing:** Both frameworks benefit from their straightforward resource management on a single machine. They efficiently utilize all available CPU cores without significant communication overhead.

**Dask:** While Dask excels in distributed or larger-than-memory computations, its performance was constrained in this scenario due to the additional coordination overhead required for managing tasks, especially for smaller datasets and multi-core setups.

## Conclusion

* For handling imbalanced datasets, SMOTE (Synthetic Minority Oversampling Technique) addresses the issue by generating synthetic samples for the minority class, effectively balancing the dataset. On the other hand, the Random Under Sampler achieves balance by retaining all minority class samples while randomly selecting a subset of majority class samples based on the specified sampling ratio. These two methods yield different outcomes: SMOTE often boosts metrics like accuracy and AUC-ROC, sometimes reaching values as high as 0.99. However, this doesn't necessarily make the model more robust, as the synthetic samples tend to be highly similar to each other, potentially leading to overfitting. In contrast, the Random Under Sampler is generally more reliable since it relies solely on real data, although it discards a portion of the majority class samples randomly. A key difference is that, SMOTE expands the dataset size by adding synthetic data, while the Random Under Sampler reduces the dataset size by removing excess majority samples.
* The more imbalanced a dataset is, the lower the precision and AUC-ROC score tend to be. When the fraudulent samples contain only a small fraction of the data, the model struggles to learn the distinguishing features of fraud (or non-fraud) cases, resulting in very poor performance. However, as the proportion of minority samples increases, the model's ability to categorize them correctly improves. When we trained our model on the original imbalanced dataset, its precision remained very poor, never exceeding 60%, and only got a AUC-ROC score of 0.5, which means it was just guessing. While it was easy for the model to classify the majority of non-fraud samples correctly, identifying the minority fraudulent samples was highly challenging. By applying the Random Under Sampler, the model's precision and AUC-ROC scores improved significantly. The confusion matrix also showed better results: the model started to identify more fraudulent samples.
* While Joblib starts slightly faster than Multiprocessing, it doesn’t scale as well when more CPUs are added. This is because Joblib relies on threads, which are limited by Python’s GIL, while Multiprocessing uses processes, avoiding this limitation. As more CPUs are used, the performance gap between the two methods would likely grow, making Multiprocessing the better choice for larger workloads.
* Dask’s performance with Random Forest suffers because it is not optimized for tree-based algorithms. Without specific optimizations for tree-based models, Dask introduces unnecessary overhead, leading to slower performance. For Random Forest, frameworks like Joblib or Multiprocessing are more efficient due to their simpler task execution and lower communication demands. But Dask is not as poor as expected, because Dask is useful for parallelizing and distributing large-scale data processing tasks across multiple cores or clusters. However, communicating between processes can still cause additional overhead costs, which makes it less efficient for certain computationally intensive, single-node tasks compared to more lightweight libraries like Joblib or Multiprocessing.

## References

[1] Dask — Dask documentation. (n.d.). <https://docs.dask.org/en/stable/dataframe.html>

[2] Random Under Sampler

<https://imbalanced-learn.org/stable/references/generated/imblearn.under_sampling.RandomUnderSampler.html>

[3]SMOTE: <https://imbalanced-learn.org/stable/references/generated/imblearn.over_sampling.SMOTE.html#r001eabbe5dd7-1>

[4] ROC-AUC score:

<https://developers.google.com/machine-learning/crash-course/classification/roc-and-auc>

[5] Joblib: <https://joblib.readthedocs.io/en/latest/generated/joblib.parallel_backend.html>

[6] Multiprocessing: <https://docs.python.org/3/library/multiprocessing.html>

[7] Bank Account Fraud Dataset Suite (NeurIPS 2022) <https://www.kaggle.com/datasets/sgpjesus/bank-account-fraud-dataset-neurips-2022/data?select=Base.csv>