Credit Card Fraud Detection Project Report

1. Introduction

Credit cards are widely used for online transactions, providing convenience but also posing risks of fraud. Credit card fraud is the unauthorized use of someone else's credit card information to make purchases or withdraw cash. Detecting such fraudulent transactions is crucial to ensure that customers are not wrongly charged for purchases they did not make.

This project focuses on developing a model to detect fraudulent credit card transactions using a dataset from September 2013, which contains 492 frauds out of 284,807 transactions. The dataset is highly imbalanced, with the positive class (frauds) accounting for only 0.172% of all transactions.

2. Business Understanding

As digital transactions have become prevalent, banks face the challenge of monitoring a high volume of transactions to prevent fraud. Manual monitoring is impractical due to the sheer number of transactions, making automated fraud detection models essential. The goal is to build a model that accurately predicts fraudulent transactions in real-time, minimizing financial losses and customer inconvenience.

3. Project Folder Structure

- **data/raw:** Contains the original dataset (creditcard.csv).
- **data/processed-data:** Contains final processed dataset and split data for training, testing and undersampled and oversampled data used in the notebooks.
- **models:** Contains trained models and the final hyperparameter-tuned model for deployment. It also has Scaler used for Amount Column. Below is the summary of the folder.
- **visuals:** Contains all figures and images generated during the project.
- **requirements.txt:** Lists all dependencies required to run the project.
- **README.md:** This file, explaining the project structure and usage.

Key Files and Directories

- **data:** Contains processed files such as X_train, y_train, X_test, y_test, X_train_undersampled, y_train_undersampled, X_train_smote, y_train_smote used in the notebooks.
- **models:** Contains trained models in joblib file and JSON files with metrics of each model, and the scaler (amount_scaler.joblib).
- **notebooks:**
    - **Notebook 1:** data-exploration-and-preprocessing.ipynb
    - **Notebook 2:** model-training-evaluation.ipynb
    - **Notebook 3:** hyperparameter-tuning-final-model-evaluation.ipynb
- **visuals:** Contains PNG files of plots and figures from EDA and model evaluation.

4. Installation Instructions:

Prerequisites

- Python 3.6 or higher
- Jupyter Notebook or Jupyter Lab
- Libraries listed in the requirements.txt file

How to Run the Project

1. Ensure the dataset (raw & processed-data) is located in the right folder as mentioned above(#2)
2. Open and run the Jupyter notebooks in the following sequence:
    - **Notebook 1:** data-exploration-and-preprocessing.ipynb
    - **Notebook 2:** model-training-evaluation.ipynb
    - **Notebook 3:** hyperparameter-tuning-final-model-evaluation.ipynb
3. Execute the cells in each notebook to replicate the results.

5. Project Steps and Reasoning

5.1. Exploratory Data Analysis (EDA) & Data Pre-processing :

Initially dataset is loaded and sample is visualized using .head()

STEP 3 - VISUALIZE DATA - EDA

View and Understand the Data

```
In [5]: # Lets see first 5 records to understand the data glimpse
        df.head()
```

Out[5]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V23	V24	V2
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	...	-0.018307	0.277838	-0.110474	0.066928	0.12853
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	...	-0.225775	-0.638672	0.101288	-0.339846	0.16717
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	...	0.247998	0.771679	0.909412	-0.689281	-0.32764
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	...	-0.108300	0.005274	-0.190321	-1.175575	0.64737
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	...	-0.009431	0.798278	-0.137458	0.141267	-0.20601

5 rows × 31 columns

```
In [6]: # Lets check the shape of the data
        df.shape
```

Out[6]: (284807, 31)

Checked the df.info()

We did not see any **null values** in the data. Hence, we don't have to handle null values.

Checked Datatypes of columns

The Time column's data type is marked as float, but it should be an integer. The same is corrected.

Data Description Analysis

```
In [14]: # Lets check the description of the data

         df.describe()
```

Out[14]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	..
count	284807.000000	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	..
mean	94813.859575	1.759061e-12	-8.251130e-13	-9.654937e-13	8.321385e-13	1.649999e-13	4.248366e-13	-3.054600e-13	8.777971e-14	-1.179749e-12	..
std	47488.145955	1.958696e+00	1.651309e+00	1.516255e+00	1.415869e+00	1.380247e+00	1.332271e+00	1.237094e+00	1.194353e+00	1.098632e+00	..
min	0.000000	-5.640751e+01	-7.271573e+01	-4.832559e+01	-5.683171e+00	-1.137433e+02	-2.616051e+01	-4.355724e+01	-7.321672e+01	-1.343407e+01	..
25%	54201.500000	-9.203734e-01	-5.985499e-01	-8.903648e-01	-8.486401e-01	-6.915971e-01	-7.682956e-01	-5.540759e-01	-2.086297e-01	-6.430976e-01	..
50%	84692.000000	1.810880e-02	6.548556e-02	1.798463e-01	-1.984653e-02	-5.433583e-02	-2.741871e-01	4.010308e-02	2.235804e-02	-5.142873e-02	..
75%	139320.500000	1.315642e+00	8.037239e-01	1.027196e+00	7.433413e-01	6.119264e-01	3.985649e-01	5.704361e-01	3.273459e-01	5.971390e-01	..
max	172792.000000	2.454930e+00	2.205773e+01	9.382558e+00	1.687534e+01	3.480167e+01	7.330163e+01	1.205895e+02	2.000721e+01	1.559499e+01	..

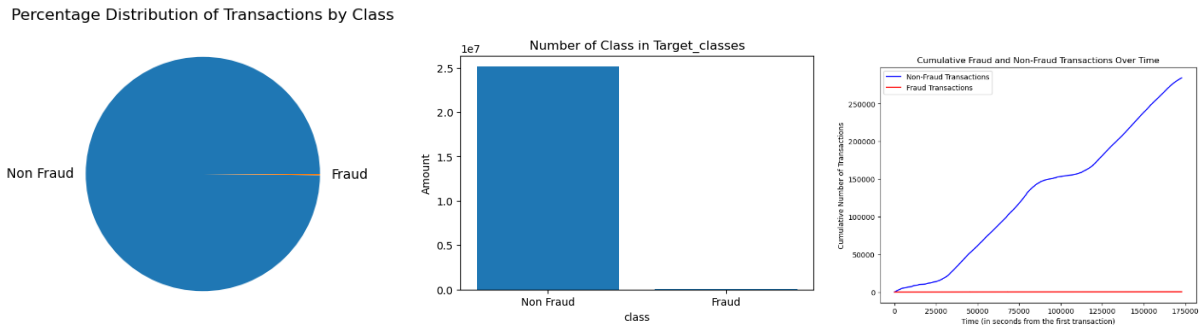
8 rows × 31 columns

Since the features V1 - V28 data is PCA transformed, we dont have to treat any outliers separeitly from these features. Lets work on non transformed Target Variable, Amount column ¶

Based on the nomenclature and Features in the dataset V1 - V28 looks PCA (Principal Component Analysis) transformed. we don't have to treat any outliers separately from these features. Lets work on non-transformed Time, Amount column, Class (Dependent Variable).

Target Variable Analysis

Visualize the Class in the Target Variable.



Above analysis indicates that there are only 2 Variables in binary format in the Target Variable and also most of FRAUD transactions are around low transaction value. And there is no pattern with time and Amount. So we can conclude to drop the time column and no transformation is required on Target Variable as it has binary values.

'Fraud_CumSum' and 'NonFraud_Cumsum' columns are used for EDA to understand time and amount. Hence it is dropped to avoid complex issues in deployment or test data prep

SPLIT DATA - Train Data and Test Data

We have split the data into 80:20 ratio randomly. 80% is used in training and the rest 20% is used in testing. Split files **X_train, X_test, y_test, y_train** are saved as a data frame for further data preparation and using it for training and testing model.

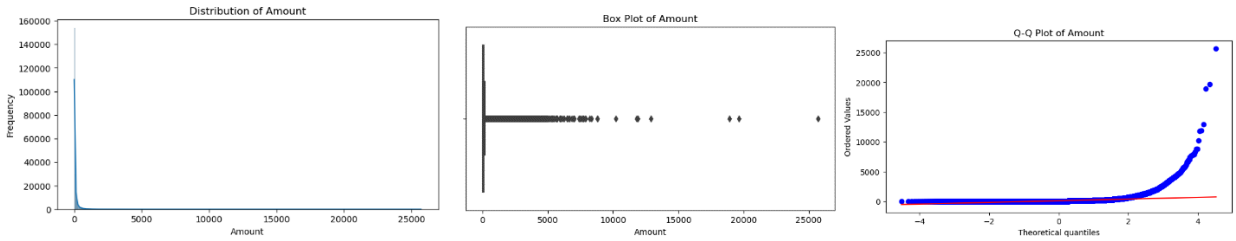
ENCODING & SCALING & TRANSFORMATION

We have done a detailed EDA & pre-processed the data. It's clear that all features except for Time, Amount and Class are already PCA Transformed. Below action was taken.

1. Time column is encoded with **OrdinalEncoder** since the data in the column has some order.
2. Amount column is a continuous data lets transform it after splitting to avoid data leakage to model.
3. Class column has binary values, hence no need to encode or transform

DISTRIBUTION OF DATA IN AMOUNT COLUMN

Scale the data - Since all columns are PCA transformed, we have scaled the left-out Amount column using the Standard Scaler. Saved the scaler file using joblib for future use on test data and in production deployment.



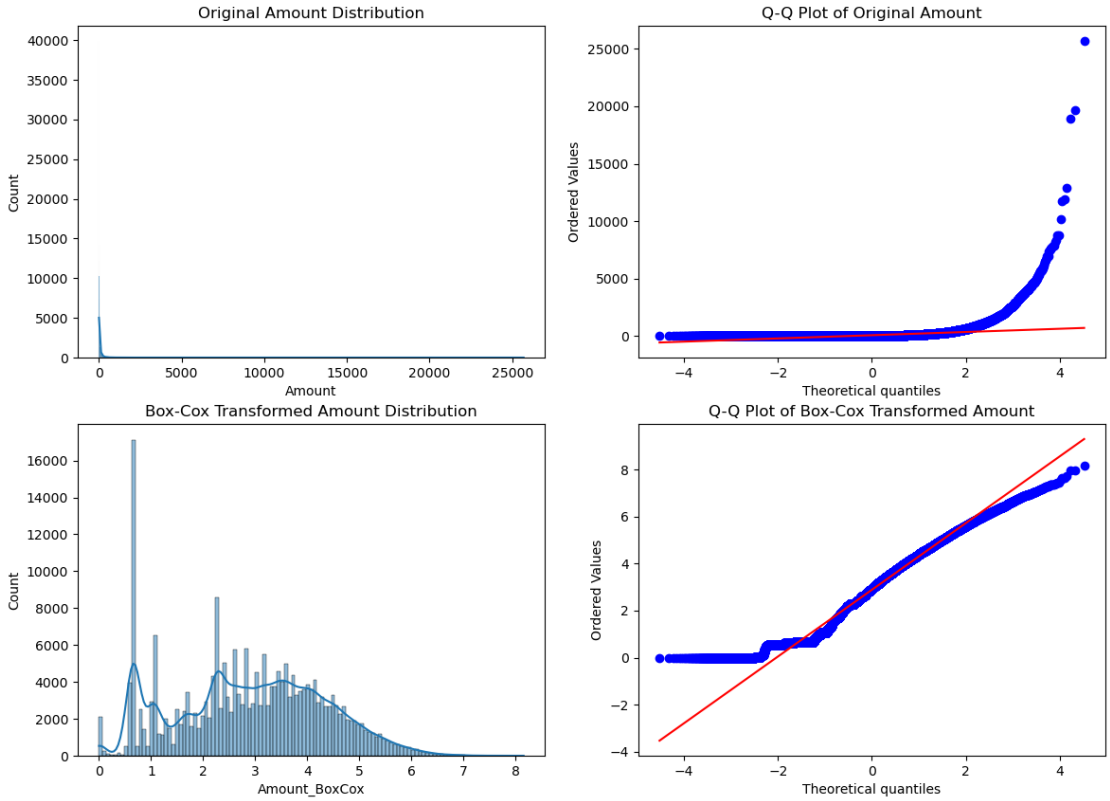
After checking the transformation of the scaled Amount column, we found that it was skewed. Therefore, we applied a Box-Cox transformation only to the Amount column, as the other columns are already PCA-transformed. **Lambda value used for Box-Cox transformation: -0.04497254555023551** is preserved and used in test data preparation pipeline to ensure consistency.

It's clearly visible with Histogram that Amount column is Right or positive Skewed. Some of the observations

1. Since the Amount column in the Credit card is a value of a transaction. It cannot be negatively skewed.
2. Amount Column - We have checked Histogram, since its too clear, plotted a Box Plot, it clearly shows Positively skewed data.
3. As its aware that V1-V28 are PCA transformed and skewness check of features show Amount column has high skewness of 18% lets transform the same using BOX COX TRANSFORATION.
4. Its also observed PCA transformed data V28, V7 and V21 is positively skewed but since its PCA transformed i'm NOT Transforming it as it could distort the principal component's nature.

Transformation - Box_Cox Transformation

Amount column feature is positively skewed let's transform it. (This data has no negative value so it's good to go). After transformation amount column data looks almost like a Normal Distribution.



Above histogram plot clearly show that the transformation is effective and its looks more symmetric.

Final Features selection for training

We have done following steps on the **X_train data frame** to prepare it for training the models. Above encoding, scaling, transformation executes as planned.

After **Ordinal encoding** of Time column new column Time_Encoded is created and hence duplicate **Time** column is **dropped** from X_train.

We have scaled the **amount column** after splitting and a new column **Amount_Scaled** is created. We have found the data is skewed by using few plots and hence transformed the Amount column using BoxCox Transformation as this is best for imbalanced data and it will help in getting a symmetrical structure to the data. After Transforming **Amount_scaled**, a new column **Amount_BoxCox** was created with transformed values.

To avoid duplication, **dropped Amount, Amount_Scaled.**

Below Features are kept ready for training the model :

```
Index(['V1', 'V2', 'V3', 'V4', 'V5', 'V6', 'V7', 'V8', 'V9', 'V10', 'V11',
       'V12', 'V13', 'V14', 'V15', 'V16', 'V17', 'V18', 'V19', 'V20', 'V21',
'V22', 'V23', 'V24', 'V25', 'V26', 'V27', 'V28', 'Time_Encoded',
'Amount_BoxCox'],
```

Amount_Scaled & Time_encoded will not be part of **X_test**. We need to scale the X_test and transform it before proceeding to train models.

Created Pipeline for preparing test data

Created the pipeline with below steps. A source code function file **'data_preparation.py'** is saved separately in the folder, it can be used for preparing the test data or in pipeline deployment

You can call the function **X_test = preprocess_data(X_test, is_dataframe=True)**. If you want to load a csv file mark has **"is_dataframe=False" if the data is a dataframe mark has "is_dataframe=True"**

Creates Undersampling and Oversampling Object

Now that the data is more consistent after all cleaning and transformation. We have created objects of Under sampling **(X_train_resampled, X_test_resampled)** and oversampling (SMOTE) **(y_train_smote, y_test_smote)** techniques

Save the processed data and objects

Since the data processing is complete. A final processed data file is saved in the data folder for future reference as required in the guidelines. Also objects are saved as a joblib file for quick reference in future. The list is as below,

```
joblib.dump(X_train, 'data/processed-data/X_train_v1.joblib')
joblib.dump(y_train, 'data/processed-data/y_train_v1.joblib')
joblib.dump(X_test, 'data/processed-data/X_test_v1.joblib')
joblib.dump(y_test, 'data/processed-data/y_test_v1.joblib')
joblib.dump(y_test, 'models/amount_scaler.joblib')
joblib.dump(X_train_resampled, 'data/processed-data/X_train_resampled_v1.joblib')
joblib.dump(y_train_resampled, 'data/processed-data/y_train_resampled_v1.joblib')
joblib.dump(X_train_smote, 'data/processed-data/X_train_smote_v1.joblib')
joblib.dump(y_train_smote, 'data/processed-data/y_train_smote_v1.joblib')
```

2.Model Training and Evaluation:

Design Choices

Choosing a Right Model and Right Metric

In the given problem statement, we can see the class is highly imbalanced due to very low fraud transactions. I want to handle the imbalanced data using RandomUndersampler and SMOTE approach. There is no standard model for classification, for the given problem of identifying the Fraud, its GBM & XGB is preferred model. Considering my computational resources available, I would use LightGBM instead of GBM. Also, I'm trying few algorithms along with XGBoost & LightGBM with above technique to handle imbalance data. Let's consolidate the metrics of every model and choose the best model.

Choosing Right Metric.

I will evaluate the model on test data using below evaluation metrics to have a comparison in the later stage to decide the best model. Though the problem statement wants to track Accuracy metric and a subjective result of at least 75%, even before building the model, we can see the chances of getting accuracy of 99% accuracy is high due to imbalanced data. Non fraud transactions in the dataset are more than 99%. I choose to track these metrics based on below justification.

Accuracy metrics : In the given problem more than 99% of data is legitimate hence accuracy will be high and considering this metric will be deceiving.

F1_score metrics : In the given problem identifying the fraud transactions is very critical than identifying non fraud transactions. Hence picking a combination of precision (identify fraud trans correctly) and recall (capturing all fraud trans) is wise choice. Instead of tracking multiple metrics like precision & recall, I consider f1 score a good metrics in this context as it's a harmonic mean of precision and recall.

ROC_AUC : Is the best metric in classifying fraud and non-fraud transaction especially in the imbalanced data. Hence let's give high importance to this metric in evaluating the best model.

Baseline Models: Logistic Regression, Gaussian Naive Bayes, Decision Trees, Random Forest, Suppot Vector Machine (classifier), KNN - K Nearest Neighbour, Light GBM and XGBoost are initially used to establish a baseline. Trained models are saved as a Joblib file in the models folder with clear nomenclature to identify.

List of trained Models - Before HyperParameter tuning.			
Logistic Regression	**Gaussian Naïve Bayes**	**Decision Tree**	**Random Forest**
logreg_base.joblib	gnb_base.joblib	dt_base.joblib	rf_base.joblib
logreg_rus.joblib	gnb_rus.joblib	dt_rus.joblib	rf_rus.joblib
logreg_smote.joblib	gnb_smote.joblib	dt_smote.joblib	rf_smote.joblib

Support Vector Machine	**XGBoost**	**LightGBM**	**KNN**
svm_base.joblib	xgb_base.joblib	lgbm_base.joblib	knn_base.joblib
svm_rus.joblib	xgb_rus.joblib	lgbm_rus.joblib	knn_rus.joblib
svm_smote.joblib	xgb_smote.joblib	lgbm_smote.joblib	knn_smote.joblib

Resampling Techniques: Undersampling, oversampling using SMOTE are applied to address class imbalance. Trained models under each algorithm and sampling technique is saved with clear nomenclature as joblib file for future reference.

List of Evaluation Metrics - for Model before Hyperparameter Tuning			
Logistic Regression	**Gaussian Naïve Bayes**	**Decision Tree**	**Random Forest**
logreg_base_metrics_v1.json	gnb_base_metrics_v1.json	dt_base_metrics_v1.json	rf_base_metrics_v1.json
logreg_rus_metrics_v1.json	gnb_rus_metrics_v1.json	dt_rus_metrics_v1.json	rf_rus_metrics_v1.json
logreg_smote_metrics_v1.json	gnb_smote_metrics_v1.json	dt_smote_metrics_v1.json	rf_smote_metrics_v1.json

Support Vector Machine	**XGBoost**	**LightGBM**	**KNN**
svm_base_metrics_v1.json	xgb_base_metrics_v1.json	lgbm_base_metrics_v1.json	knn_base_metrics_v1.json
svm_rus_metrics_v1.json	xgb_rus_metrics_v1.json	lgbm_rus_metrics_v1.json	knn_rus_metrics_v1.json
svm_smote_metrics_v1.json	xgb_smote_metrics_v1.json	lgbm_smote_metrics_v1.json	knn_smote_metrics_v1.json

4. Model Performance Evaluation

Model Evaluation: Models are evaluated using accuracy, F1 score, and ROC AUC. Every model is evaluated using above metrics and same is consolidated into a data frame. Below is the consolidated table.

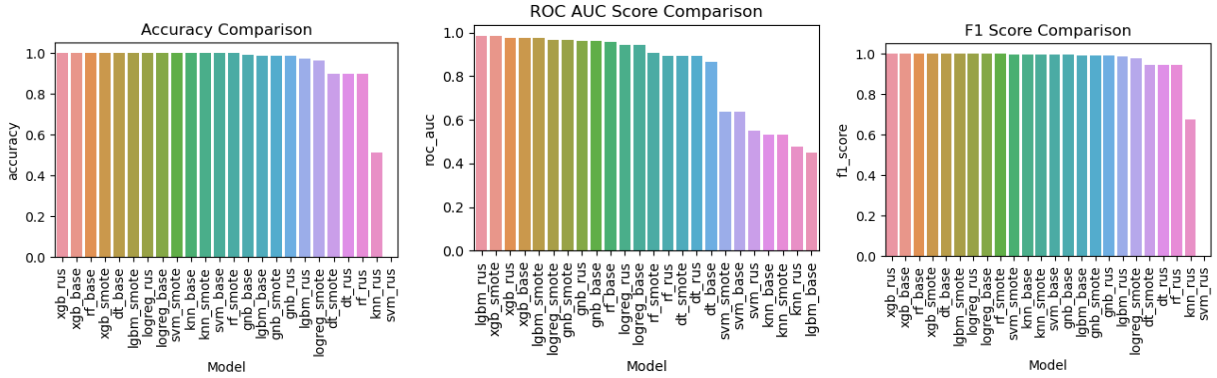
Top Models - Accuracy Metric wise					Top Models - F1 Score Metric wise					Top Models - ROC AUC Metric wise			
Model	**accuracy**	**f1_score**	**roc_auc**		**Model**	**accuracy**	**f1_score**	**roc_auc**		**Model**	**accuracy**	**f1_score**	**roc_auc**
xgb_base	**0.999596**	0.999578	0.975973		xgb_base	0.999596	**0.999578**	0.975973		xgb_smote	0.999561	0.99956	**0.987574**
xgb_rus	**0.999596**	0.999578	0.975973		xgb_rus	0.999596	**0.999578**	0.975973		lgbm_rus	0.971858	0.984314	**0.983165**
rf_base	**0.999561**	0.999541	0.959175		xgb_smote	0.999561	**0.99956**	0.987574		lgbm_smote	0.999105	0.999156	**0.98282**
xgb_smote	**0.999561**	0.99956	0.987574		rf_base	0.999561	**0.999541**	0.959175		logreg_smote	0.984024	0.990616	**0.981184**
dt_base	**0.999192**	0.999206	0.887412		dt_base	0.999192	**0.999206**	0.887412		xgb_base	0.999596	0.999578	**0.975973**
lgbm_smote	**0.999105**	0.999156	0.98282		lgbm_smote	0.999105	**0.999156**	0.98282		xgb_rus	0.999596	0.999578	**0.975973**
logreg_base	**0.998929**	0.998907	0.955558		logreg_base	0.998929	**0.998907**	0.955558		gnb_smote	0.996208	0.997079	**0.965628**
logreg_rus	**0.998929**	0.998907	0.955558		logreg_rus	0.998929	**0.998907**	0.955558		gnb_base	0.997174	0.997584	**0.964685**
knn_base	**0.998578**	0.997985	0.617392		rf_smote	0.997648	**0.998075**	0.909075		gnb_rus	0.99605	0.996985	**0.962788**
knn_smote	**0.998578**	0.997985	0.617392		knn_base	0.998578	**0.997985**	0.617392		rf_base	0.999561	0.999541	**0.959175**
svm_base	**0.998438**	0.997657	0.545068		knn_smote	0.998578	**0.997985**	0.617392		logreg_base	0.998929	0.998907	**0.955558**
svm_smote	**0.998438**	0.997657	0.545068		svm_base	0.998438	**0.997657**	0.545068		logreg_rus	0.998929	0.998907	**0.955558**
rf_smote	**0.997648**	0.998075	0.909075		svm_smote	0.998438	**0.997657**	0.545068		rf_smote	0.997648	0.998075	**0.909075**
gnb_base	**0.997174**	0.997584	0.964685		gnb_base	0.997174	**0.997584**	0.964685		dt_rus	0.913697	0.953394	**0.900689**
gnb_smote	**0.996208**	0.997079	0.965628		gnb_smote	0.996208	**0.997079**	0.965628		dt_smote	0.913697	0.953394	**0.900689**
gnb_rus	**0.99605**	0.996985	0.962788		gnb_rus	0.99605	**0.996985**	0.962788		rf_rus	0.913697	0.953394	**0.900689**
lgbm_base	**0.995523**	0.996717	0.827199		lgbm_base	0.995523	**0.996717**	0.827199		dt_base	0.999192	0.999206	**0.887412**
logreg_smote	**0.984024**	0.990616	0.981184		logreg_smote	0.984024	**0.990616**	0.981184		lgbm_base	0.995523	0.996717	**0.827199**
lgbm_rus	**0.971858**	0.984314	0.983165		lgbm_rus	0.971858	**0.984314**	0.983165		knn_rus	0.624627	0.76747	**0.686918**
dt_rus	**0.913697**	0.953394	0.900689		dt_rus	0.913697	**0.953394**	0.900689		knn_base	0.998578	0.997985	**0.617392**
dt_smote	**0.913697**	0.953394	0.900689		dt_smote	0.913697	**0.953394**	0.900689		knn_smote	0.998578	0.997985	**0.617392**
rf_rus	**0.913697**	0.953394	0.900689		rf_rus	0.913697	**0.953394**	0.900689		svm_rus	0.514694	0.67821	**0.555845**
knn_rus	**0.624627**	0.76747	0.686918		knn_rus	0.624627	**0.76747**	0.686918		svm_base	0.998438	0.997657	**0.545068**
svm_rus	**0.514694**	0.67821	0.555845		svm_rus	0.514694	**0.67821**	0.555845		svm_smote	0.998438	0.997657	**0.545068**

As discussed earlier, it's clear that the accuracy metric shows high performance in most models. F1 Score shows a slight difference in the performance metrics. But ROC_AUC clearly distinguishes the model performance more clearly as expected. It displays the strength of ROC_AUC in imbalanced data performance evaluation.

Since some of the metrics are having a very low difference, created a barchart as follows to see which model has performed better in every evaluation metric.

Top 5 Models as per metrics :

1. Accuracy : xgb_rus, xgb_base, xgb_smote, rf_base, dt_base.
2. F1_score : xgb_rus, xgb_base, xgb_smote, Rf_base, dt_base.
3. Roc_Auc : xgb_smote, lgbm_rus, lgbm_smote, logreg_smote, Xgb_rus.



Its clear that best results across all Algorithms are from XGB and LGBM. In Accuracy & F1_score we can see rf_base, dt_base are one of the top 5 contenders. I give more weightage to ROC_AUC metric since it is very robust in problems with imbalanced data.

Hence I'm proceeding with Hyper parameter Tuning or Cross-validation for only these 2 algorithms. Lets check if this improves the performance or help in understanding important hyperparameter.

Hyperparameter Tuning and Final Model Evaluation:

As observed, XGBoost & LBGB models have performed well on test data we will choose these 2 models to hyper parameter tuning using GridSearchCV to get the best possible params.

Based on the best params let's build a final best model under XGBoost & LightGBM, so that we can deploy one of them as for deployment.

GridSearchCV is Used to find the best hyperparameters for models like XGBoost and LightGBM. Using the best hyperparameter new XGBoost model was trained.

COMPARE XGBoost Base Model & XGBoost Best Model(hyperparameter tuned)

The first grid search found these hyperparameters for XGBoost:
Best parameters for XGBoost: {'learning_rate': 0.2, 'max_depth': 3, 'n_estimators': 300, 'subsample': 0.7}

Model	Accuracy	F1 Score	ROC_AUC
XGB_BASE	0.999579	0.999561	0.977255
XGB_BEST	0.999596	0.999585	0.973098

If you observe the best model metrics, we can see a very small improvement in Accuracy and F1Score values of XGB_best Model compared with the XGB_base Model. However, my main goal of improving the ROC AUC was not achieved. In fact, the best model underperformed slightly. But when compared with top Roc_AUC of XGB_smote value of 0.987574. It is clear the current result is Low. Though the difference in performance of the model is very negligible I'm still trying if the metrics can improve.

It is not a good idea to implement an XGB_Smote in production since it has worked on artifically created data. So, trying to get best parameters to improve Roc_Auc performance of base model. So, tuning some of the GridSearchCV values to see if we can get better parameters.

Run a XGBoost GridSearchCV for the 2nd time with slight modification to the parameter and found below best parameter.

Second Grid search on XGBoost model gave these best Hyper parameters has
{'learning_rate': 0.15, 'max_depth': 7, 'n_estimators': 400, 'subsample': 0.7}

Model	Accuracy	F1 Score	ROC_AUC
XGB_BASE	0.999579	0.999561	0.977255
XGB_BEST	0.999596	0.999585	0.973098
XGB_BEST1	0.999648	0.999636	0.974622

Original XGB_Base model has highly performed with ROC_AUC metric. But above results shows XGB_BEST and XGB_best1 have slight upper hand with regard to Accuracy and F1_Score Metrics.

2. Hyperparameter Tuning - GridsearchCV - LightGBM

GridseachCV was done on LightGBM Model and got below Best Hyperparameters :

```
{'learning_rate': 0.01, 'max_depth': 20, 'min_child_samples': 50, 'n_estimators':
100, 'num_leaves': 31, 'subsample': 0.8}
```

Model	Accuracy	F1 Score	ROC_AUC
LGBM_BASE	0.988413	0.992673	0.449614 (rus= 0.984544)
LGBM_BEST	0.999420	0.999368	0.970016

Hyper parameter tuned model lgbm_best model has perfomed better than the base model lgbm_base across all 3 metrics (accuracy, f1Score, Roc_auc). So further no tuning required and we can conclude **lgbm_best model is better than base model.**

3. Hyperparameter Tuning - RandomoizedsearchCV - RF - RANDOM FOREST (2nd Model for Gridsearch)

1. Due to computational resource constraint. Running RandomizedSearchCV instead of GridSearchCV for RF Model.
2. Initially planned to run hyperparameter tuning only on XGB & LGBM models based on ROC_AUC Metrics, values slightly changed while rerunning the model, hence tuning 3rd model – RF.
3. Since RF base models have performed much better in Accuracy and F1 Score, trying to check if RF tuning can improve performance.

Model	Accuracy	F1 Score	ROC_AUC
RF_BASE	0.999561	0.999541	0.959023
RF_BEST	0.999579	0.999560	0.977542

RF_BEST model has performed much better than the base model across all metrics.

6. Final Model Selection & conclusion

Justification for selecting the final model for deployment.

Let's compare top model metrics and decide on the best model for production.

:

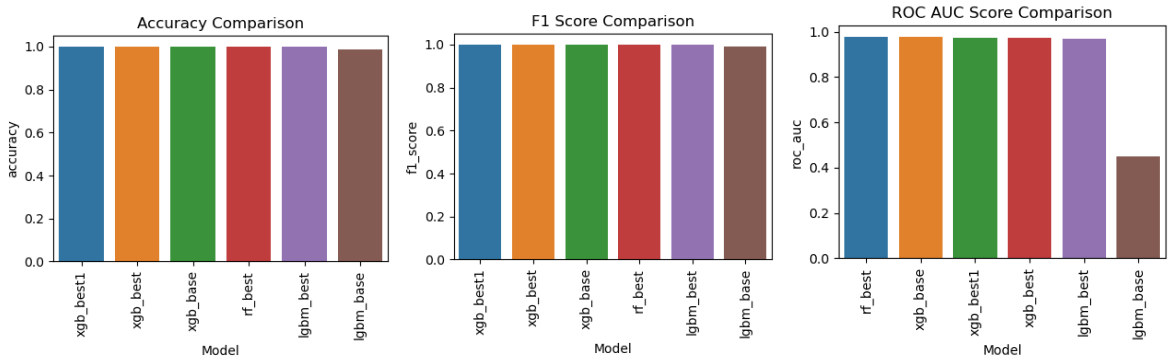
	Model	accuracy	f1_score	roc_auc
0	lgbm_base	0.988413	0.992673	0.449614
1	xgb_base	0.999579	0.999561	0.977255
2	xgb_best	0.999596	0.999585	0.973098
3	xgb_best1	0.999649	0.999636	0.974623
4	lgbm_best	0.999421	0.999369	0.970017
5	rf_best	0.999579	0.999561	0.977542

We can observe the data of above algorithms and compare their performance with base model and hyperparameter tuned models.

RANK	ACCURACY	F1_SCORE	ROC_AUC
1ST RANK	XGB_BEST1	LGBM_BASE	RF_BEST
2ND RANK	XGB_BEST	XGB_BEST1	XGB_BASE
3RD RANK	XGB_BASE & RF_BEST	XGB_BASE & RF_BEST	XGB_BASE & XGB_BEST1

By seeing above matrix, we can see XGBoost Algorithm has performed better than the LightGBM and Random Forest Algorithm. Though each model has top performed in different metrics. We can see XGB Model has performed better in all 3 metrics. Lets analyze the models

Visualization of the final metrics is as follows:



Cross Validation Score

Below are the Cross Validation Score of 3 hyper parameter tuned models. This clearly shows the generalization of the model. Low Standard Deviation indicates that the models are well generalization.

MOdel	CV1	CV2	CV3	CV4	CV5	MEAN	STD
XGB_BEST1	0.987820	0.976308	0.985477	0.986883	0.980190	0.983336	0.004394
Rf_BEST	0.986796	0.978242	0.964606	0.981180	0.961731	0.974511	0.009703
LGBM_BEST	0.982196	0.964279	0.961412	0.958605	0.957724	0.964843	0.008975

Based on the above summary of metrics of the top model. We can clearly see that **XGB Algorithm model Xgb_best1** has performed well across all evaluation matrix and the low Standard Deviation of the model clearly indicates that the model has very well generalized. It also indicates that the model is less likely to overfit and will perform very well on unseen data.

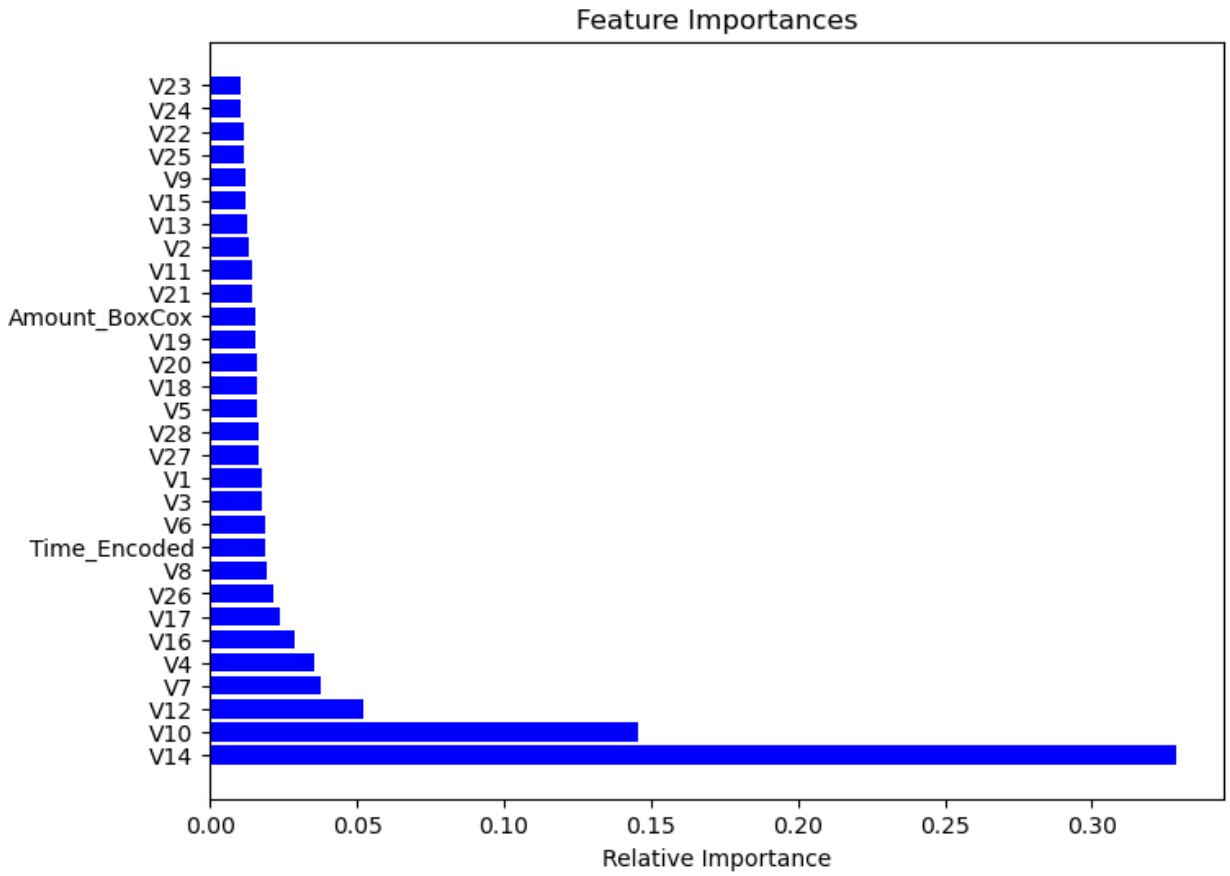
Though the difference in the metrics is minimal, we can clearly say that this model is the optimal choice for deployment. This model is expected to effectively detect fraudulent transactions and minimize both false positives and false negatives.

XGBoost model (XGB_BEST1) is an optimal choice for better prediction capacity, but lets remember that this models needs more Resource & time to train and infer. Considering the perfection trade off with resources is well justified. But if the time and resources are a concern, the next best model to choose will be LGBM_Best which is quite fast in computation and in handling Large dataset, specially sparse dataset.

Feature Importance :

Based on the above metrics and analysis we have found that the Model “xgb_best1” has the perfect model in the given situation. Below are the important Features which has influenced the model. Same is plotted in chronology to visualize the important feature in the dataset.

Since this is the Best Model choosen for deployments, lets find out the feature importance for the XGB_best1 Model before deployment. In the below plot we can **see Feature V14, V10, V12, V7, V4, V16** are the top features in ascertaining the model prediction.



Future Work

1. Though the model performance is good. We need to fit few of the deep learning models to check if it can improve performance.
2. We should continuously improve our model by reviewing the prediction and re-training the model wherever possible.
3. Every day fraud transactions are changing and our approach to identify the same also has to be upgraded based on regular learning.
4. We need to incorporate new data and improve the model at regular interval and watch if any pattern in missing

Deployment

Based on the above discussion we have finalized the best model. It's evaluated on multiple metrics. Before deploying in the production on cloud. I have deployed the model on the **STREAM LIT** for testing on multiple independent data.

This approach will help the non-technical users to test the model and understand the result. And infact it has helped me to find out the bugs in the process.

Features of the Deployed StreamLit App :

1. Option to upload the CSV File – User can upload a csv file with standard parameters.
2. After uploading if the features required and features in the uploaded file, an error will be thrown with the details.
3. Select 1 of 7 models to see the prediction. – User gets brief intro to the model.
4. On clicking on PREDICT button – Prediction will be done uploaded done using the selected model and result will be shown on the screen.
5. An option to download processed data in csv – After prediction, a file will be saved with prediction column and at this stage a row_id will be created in original file and will be indexed in the output file for easy identification of the transaction as the result columns will have processed data.
6. Output file will show the file name with the respective models selected for prediction, so the user will be able to understand which model prediction is saved in the particular file.
7. A reset button to clear all the processes and prepares for fresh prediction.