Credit Card Fraud Detection

Team members:

Abhijith SS - 961721104001

Adarsh LS - 961721104003

Anandhu S - 961721104302

Ashkar Muhammad AM - 961721104305

Akhil SS - 961721104301

Phase 5 Submission Document

Project: Credit Card Fraud Detection

Introduction:

* **Protecting Consumers: Prevents unauthorized transactions and identity theft.**
* **Protecting Businesses: Safeguards against revenue loss and reputational damage.**
* **Reducing Financial Losses: Minimizes financial impact on individuals and organizations.**
* **Maintaining Trust in the Financial System: Ensures confidence in payment networks.**
* **Future Developments :**we need credit card fraud detection techniques**to protect the cardholders from false activity.** India is on its way to becoming a developed country. To achieve this, the Government of India (GoI) has launched several initiatives and one of these is Digital India Campaign.

Content for Project Phase 2 :

For analyzing data, we need some libraries. In this section, we are importing all the required libraries like pandas, NumPy, matplotlib, plotly, seaborn, and word cloud that are required for data analysis. Check the below code to import all the required libraries.

Data Source

A good data source for credit card fraud detection should be Accurate, Complete, Covering the geographic area of interest, Accessible.

Dataset Link: (https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud)

# Data Collection and Pre-processing:

* Data collection : With Credit Card Fraud Detection, this project demonstrates the modelling of a data collection using **machine learning**. Modelling prior credit card transactions with data from those that turned out to be fraudulent is part of the Credit Card Fraud Detection Problem. The model is then used to determine whether or not a new transaction is fraudulent.

# Data pre-processing: Analysing the effect of data preprocessing techniques using machine learning algorithms on the diagnosis of fraud detection

# Exploratory Data Analysis ( EDA ):

# This case study is focused to give you an idea of applying Exploratory Data Analysis (EDA) in a real business scenario. In this case study, apart from applying the various Exploratory Data Analysis (EDA) techniques, you will also develop a basic understanding of risk analytics and understand how data can be utilized in order to minimise the risk of losing money while lending to customers.

# Feature Engineering:

**Feature engineering** is a crucial step in credit card fraud detection. It involves selecting and transforming the most relevant features from the dataset to improve the performance of machine learning models. [In credit card fraud detection, feature engineering can help identify patterns and anomalies in transaction data that are indicative of fraudulent activity](https://medium.com/dataman-in-ai/how-to-create-good-features-in-fraud-detection-de6562f249ef).

Advanced Regression Techniques:

# There are several machine learning techniques that can be used for credit card fraud detection. One such technique is **logistic regression**. [In a study, researchers investigated the use of logistic regression to detect fraudulent credit card transactions in an imbalanced dataset where only a small fraction of transactions are fraudulent](https://ieeexplore.ieee.org/document/10112302/).Another technique is **genetic algorithm (GA)** based feature selection. A recent paper proposed a machine learning based credit card fraud detection engine using the GA algorithm for feature selection. [After the optimized features are chosen, the proposed detection engine uses the following ML classifiers: Decision Tree (DT), Random Forest (RF), Logistic Regression (LR), Artificial Neural Network (ANN), and Naive Bayes (NB) 2](https://journalofbigdata.springeropen.com/articles/10.1186/s40537-022-00573-8). [The paper also demonstrated that their proposed approach outperforms existing systems](https://journalofbigdata.springeropen.com/articles/10.1186/s40537-022-00573-8).

# Model Interpretability:

Credit card fraud is a growing problem in the financial industry, with the potential to cause significant

financial losses to both customers and financial institutions. As a result, there has been a significant

amount of research in recent years on developing effective fraud detection systems. These systems rely on

a combination of statistical techniques, machine learning algorithms, and deep learning models to identify

fraudulent transactions.One of the most commonly used approaches for credit card fraud detection is

rule-based systems. These systems use predefined rules to identify transactions that are deemed suspicious.

However, rule-based systems have limitations, as they are only as good as the rules that have been

predefined, and they may not be able to detect new types of fraud. To overcome these limitations, machine

learning algorithms and statistical techniques have been applied to credit card fraud detection. These

techniques are based on analysing transaction-related data, such as the transaction amount, location, and

time, as well as other relevant factors, such as the customer’s transaction history and account details. In

recent years, deep learning models, such as convolutional neural networks (CNNs) and recurrent neural

networks (RNNs), have also been applied to credit card fraud detection. These models have shown

promising results in identifying fraudulent transactions by learning patterns in the data and improving the

accuracy of fraud detection. Overall, credit card fraud detection is a critical area of research in the financial

industry, with significant potential for improving fraud detection rates and reducing financial losse

Credit card fraud is a growing problem in the financial industry, with the potential to cause significant

financial losses to both customers and financial institutions. As a result, there has been a significant

amount of research in recent years on developing effective fraud detection systems. These systems rely on

a combination of statistical techniques, machine learning algorithms, and deep learning models to identify

fraudulent transactions.One of the most commonly used approaches for credit card fraud detection is

rule-based systems. These systems use predefined rules to identify transactions that are deemed suspicious.

However, rule-based systems have limitations, as they are only as good as the rules that have been

predefined, and they may not be able to detect new types of fraud. To overcome these limitations, machine

learning algorithms and statistical techniques have been applied to credit card fraud detection. These

techniques are based on analysing transaction-related data, such as the transaction amount, location, and

time, as well as other relevant factors, such as the customer’s transaction history and account details. In

recent years, deep learning models, such as convolutional neural networks (CNNs) and recurrent neural

networks (RNNs), have also been applied to credit card fraud detection. These models have shown

promising results in identifying fraudulent transactions by learning patterns in the data and improving the

accuracy of fraud detection. Overall, credit card fraud detection is a critical area of research in the financial

industry, with significant potential for improving fraud detection rates and reducing financial losse

Credit Card Fraud can be defined as a case where a person uses someone else’s credit card for personal reasons while the owner and the card-issuing authorities are unaware of the fact that the card is being used. Due to the rise and acceleration of E-Commerce, there has been a tremendous use of credit cards for online shopping which led to High amount of frauds related to credit cards. In the era of digitalization, the need to identify credit card frauds is necessary. Fraud detection involves monitoring and analysing the behaviour of various users to estimate detect or avoid undesirable behaviour. To identify credit card fraud detection effectively, we need to understand the various technologies, algorithms and types involved in detecting credit card frauds. The algorithm can differentiate transactions which are fraudulent or not. Find fraud, they need to passed dataset and knowledge of the fraudulent transaction. They analyze the dataset and classify all transactions. Fraud detection involves monitoring the activities of populations of users to estimate, perceive or avoid objectionable behaviour, which consist of fraud, intrusion, and defaulting. Machine learning algorithms are employed to analyses all the authorized transactions and report the suspicious ones. These reports are investigated by professionals who contact the cardholders to confirm if the transaction was genuine or fraudulent. The investigators provide feedback to the automated system which is used to train and update the algorithm to eventually improve the fraud-detection performance over time.

# Deployment and Prediction:

Deploy the chosen regression model to credit card fraud detection

Develop a user-friendly interface for users to input property features.

Program:

### Credit Card Fraud Detection

$ pip install sklearn==0.24.2 imbalanced-learn numpy pandas matplotlib seaborn

Let's import the necessary libraries:

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from matplotlib import gridspec

Now we read the data and try to understand each feature's meaning. The Python module [pandas](https://pandas.pydata.org/) provide us with the functions to read data. In the next step, we will read the data from our directory where the data is saved, and then we look at the first and last five rows of the data using head(), and tail() methods:

dataset = pd.read\_csv("creditcard.csv")

dataset.head().append(dataset.tail())

╔════════╤══════════╤════════════╤═══════════╤═══════════╤═══════════╤═══════════╤═══════════╤═══════════╤═══════════╤═══════════╤═════╤═══════════╤═══════════╤═══════════╤═══════════╤═══════════╤═══════════╤═══════════╤═══════════╤════════╤═══════╗

║ │ Time │ V1 │ V2 │ V3 │ V4 │ V5 │ V6 │ V7 │ V8 │ V9 │ ... │ V21 │ V22 │ V23 │ V24 │ V25 │ V26 │ V27 │ V28 │ Amount │ Class ║

╠════════╪══════════╪════════════╪═══════════╪═══════════╪═══════════╪═══════════╪═══════════╪═══════════╪═══════════╪═══════════╪═════╪═══════════╪═══════════╪═══════════╪═══════════╪═══════════╪═══════════╪═══════════╪═══════════╪════════╪═══════╣

║ 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 │ 0.128539 │ -0.189115 │ 0.133558 │ -0.021053 │ 149.62 │ 0 ║

╟────────┼──────────┼────────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼─────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼────────┼───────╢

║ 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 │ 0.167170 │ 0.125895 │ -0.008983 │ 0.014724 │ 2.69 │ 0 ║

╟────────┼──────────┼────────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼─────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼────────┼───────╢

║ 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 │ -0.327642 │ -0.139097 │ -0.055353 │ -0.059752 │ 378.66 │ 0 ║

╟────────┼──────────┼────────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼─────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼────────┼───────╢

║ 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 │ 0.647376 │ -0.221929 │ 0.062723 │ 0.061458 │ 123.50 │ 0 ║

╟────────┼──────────┼────────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼─────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼────────┼───────╢

║ 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 │ -0.206010 │ 0.502292 │ 0.219422 │ 0.215153 │ 69.99 │ 0 ║

╟────────┼──────────┼────────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼─────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼────────┼───────╢

║ 284802 │ 172786.0 │ -11.881118 │ 10.071785 │ -9.834783 │ -2.066656 │ -5.364473 │ -2.606837 │ -4.918215 │ 7.305334 │ 1.914428 │ ... │ 0.213454 │ 0.111864 │ 1.014480 │ -0.509348 │ 1.436807 │ 0.250034 │ 0.943651 │ 0.823731 │ 0.77 │ 0 ║

╟────────┼──────────┼────────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼─────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼────────┼───────╢

║ 284803 │ 172787.0 │ -0.732789 │ -0.055080 │ 2.035030 │ -0.738589 │ 0.868229 │ 1.058415 │ 0.024330 │ 0.294869 │ 0.584800 │ ... │ 0.214205 │ 0.924384 │ 0.012463 │ -1.016226 │ -0.606624 │ -0.395255 │ 0.068472 │ -0.053527 │ 24.79 │ 0 ║

╟────────┼──────────┼────────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼─────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼────────┼───────╢

║ 284804 │ 172788.0 │ 1.919565 │ -0.301254 │ -3.249640 │ -0.557828 │ 2.630515 │ 3.031260 │ -0.296827 │ 0.708417 │ 0.432454 │ ... │ 0.232045 │ 0.578229 │ -0.037501 │ 0.640134 │ 0.265745 │ -0.087371 │ 0.004455 │ -0.026561 │ 67.88 │ 0 ║

╟────────┼──────────┼────────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼─────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼────────┼───────╢

║ 284805 │ 172788.0 │ -0.240440 │ 0.530483 │ 0.702510 │ 0.689799 │ -0.377961 │ 0.623708 │ -0.686180 │ 0.679145 │ 0.392087 │ ... │ 0.265245 │ 0.800049 │ -0.163298 │ 0.123205 │ -0.569159 │ 0.546668 │ 0.108821 │ 0.104533 │ 10.00 │ 0 ║

╟────────┼──────────┼────────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼─────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼────────┼───────╢

║ 284806 │ 172792.0 │ -0.533413 │ -0.189733 │ 0.703337 │ -0.506271 │ -0.012546 │ -0.649617 │ 1.577006 │ -0.414650 │ 0.486180 │ ... │ 0.261057 │ 0.643078 │ 0.376777 │ 0.008797 │ -0.473649 │ -0.818267 │ -0.002415 │ 0.013649 │ 217.00 │ 0 ║

╚════════╧══════════╧════════════╧═══════════╧═══════════╧═══════════╧═══════════╧═══════════╧═══════════╧═══════════╧═══════════╧═════╧═══════════╧═══════════╧═══════════╧═══════════╧═══════════╧═══════════╧═══════════╧═══════════╧════════╧═══════╝

The Time is measured in seconds since the first transaction in the data collection. As a result, we may infer that this dataset contains all transactions recorded during two days. The features were prepared using PCA, so the physical interpretation of individual features does not make sense. 'Time' and 'Amount' are the only features that are not transformed to PCA. 'Class' is the response variable, and it has a value of 1 if there is fraud and 0 otherwise.

## Data Exploration and Visualization

Now we try to find out the relative proportion of valid and fraudulent credit card transactions:

print("Fraudulent Cases: " + str(len(dataset[dataset["Class"] == 1])))

print("Valid Transactions: " + str(len(dataset[dataset["Class"] == 0])))

print("Proportion of Fraudulent Cases: " + str(len(dataset[dataset["Class"] == 1])/ dataset.shape[0]))

data\_p = dataset.copy()

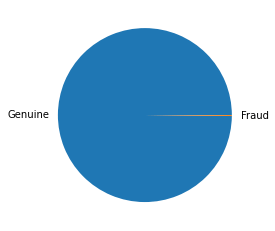
data\_p[" "] = np.where(data\_p["Class"] == 1 , "Fraud", "Genuine")

data\_p[" "].value\_counts().plot(kind="pie")

Fraudulent Cases: 492

Valid Transactions: 284315

Proportion of Fraudulent Cases: 0.001727485630620034



There is an imbalance in the data, with only 0.17% of the total cases being fraudulent.

Now we look at the distribution of the two named features in the dataset. For Time, it is clear that there was a particular duration in the day when most of the transactions took place:

f, axes = plt.subplots(1, 2, figsize=(18,4), sharex = True)

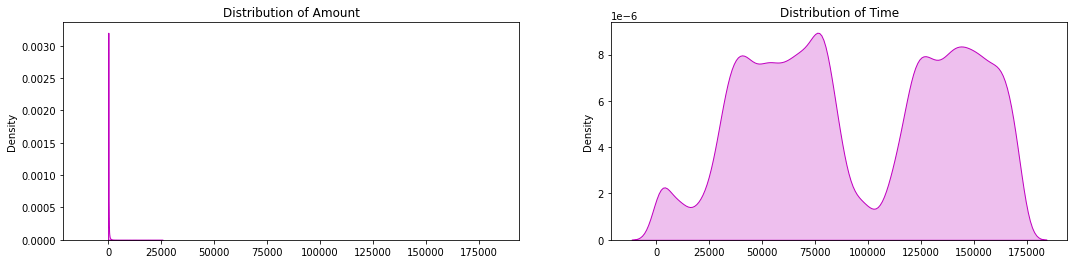
amount\_value = dataset['Amount'].values # values

time\_value = dataset['Time'].values # values

sns.distplot(amount\_value, hist=False, color="m", kde\_kws={"shade": True}, ax=axes[0]).set\_title('Distribution of Amount')

sns.distplot(time\_value, hist=False, color="m", kde\_kws={"shade": True}, ax=axes[1]).set\_title('Distribution of Time')

plt.show()



Let us check if there is any difference between valid transactions and fraudulent transactions:

print("Average Amount in a Fraudulent Transaction: " + str(dataset[dataset["Class"] == 1]["Amount"].mean()))

print("Average Amount in a Valid Transaction: " + str(dataset[dataset["Class"] == 0]["Amount"].mean()))

Average Amount in a Fraudulent Transaction: 122.21132113821133

Average Amount in a Valid Transaction: 88.29102242225574

As we can notice from this, the average money transaction for the fraudulent ones is more. It makes this problem crucial to deal with. Now let us try to understand the distribution of values in each feature. Let's start with the Amount:

print("Summary of the feature - Amount" + "\n-------------------------------")

print(dataset["Amount"].describe())

Summary of the feature - Amount

-------------------------------

count 284807.000000

mean 88.349619

std 250.120109

min 0.000000

25% 5.600000

50% 22.000000

75% 77.165000

max 25691.160000

Name: Amount, dtype: float64

The rest of the features don't have any physical interpretation and will be seen through histograms. Here the values are subgrouped according to class (valid or fraud):

data\_plot = dataset.copy()

amount = data\_plot['Amount']

data\_plot.drop(labels=['Amount'], axis=1, inplace = True)

data\_plot.insert(0, 'Amount', amount)

columns = data\_plot.iloc[:,0:30].columns

plt.figure(figsize=(12,30\*4))

grids = gridspec.GridSpec(30, 1)

for grid, index in enumerate(data\_plot[columns]):

ax = plt.subplot(grids[grid])

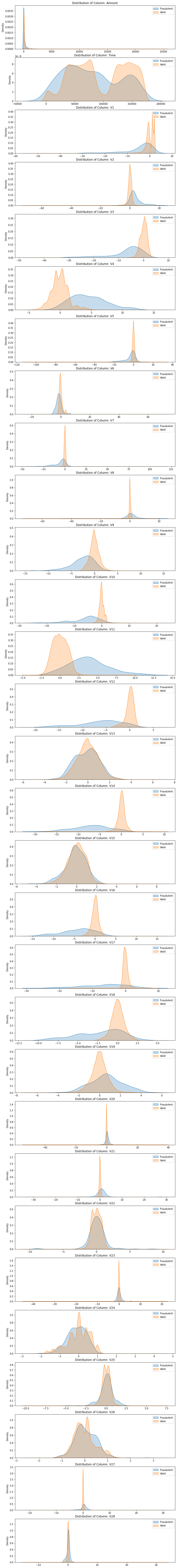
sns.distplot(data\_plot[index][data\_plot.Class == 1], hist=False, kde\_kws={"shade": True}, bins=50)

sns.distplot(data\_plot[index][data\_plot.Class == 0], hist=False, kde\_kws={"shade": True}, bins=50)

ax.set\_xlabel("")

ax.set\_title("Distribution of Column: " + str(index))

plt.show()



## Data Preparation

Since the features are created using PCA, feature selection is unnecessary as many features are tiny. Let's see if there are any missing values in the dataset:

dataset.isnull().shape[0]

print("Non-missing values: " + str(dataset.isnull().shape[0]))

print("Missing values: " + str(dataset.shape[0] - dataset.isnull().shape[0]))

Non-missing values: 284807

Missing values: 0

As there are no missing data, we turn to standardization. We standardize only Time and Amount using [RobustScaler](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.RobustScaler.html" \o "RobustScaler" \t "_blank):

from sklearn.preprocessing import RobustScaler

scaler = RobustScaler().fit(dataset[["Time", "Amount"]])

dataset[["Time", "Amount"]] = scaler.transform(dataset[["Time", "Amount"]])

dataset.head().append(dataset.tail())

As we saw previously, the Amount column has outliers, that's why we chose RobustScaler() as it's robust to outliers. Output:

╔════════╤═══════════╤════════════╤═══════════╤═══════════╤═══════════╤═══════════╤═══════════╤═══════════╤═══════════╤═══════════╤═════╤═══════════╤═══════════╤═══════════╤═══════════╤═══════════╤═══════════╤═══════════╤═══════════╤═══════════╤═══════╗

║ │ Time │ V1 │ V2 │ V3 │ V4 │ V5 │ V6 │ V7 │ V8 │ V9 │ ... │ V21 │ V22 │ V23 │ V24 │ V25 │ V26 │ V27 │ V28 │ Amount │ Class ║

╠════════╪═══════════╪════════════╪═══════════╪═══════════╪═══════════╪═══════════╪═══════════╪═══════════╪═══════════╪═══════════╪═════╪═══════════╪═══════════╪═══════════╪═══════════╪═══════════╪═══════════╪═══════════╪═══════════╪═══════════╪═══════╣

║ 0 │ -0.994983 │ -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 │ 0.128539 │ -0.189115 │ 0.133558 │ -0.021053 │ 1.783274 │ 0 ║

╟────────┼───────────┼────────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼─────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────╢

║ 1 │ -0.994983 │ 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 │ 0.167170 │ 0.125895 │ -0.008983 │ 0.014724 │ -0.269825 │ 0 ║

╟────────┼───────────┼────────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼─────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────╢

║ 2 │ -0.994972 │ -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 │ -0.327642 │ -0.139097 │ -0.055353 │ -0.059752 │ 4.983721 │ 0 ║

╟────────┼───────────┼────────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼─────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────╢

║ 3 │ -0.994972 │ -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 │ 0.647376 │ -0.221929 │ 0.062723 │ 0.061458 │ 1.418291 │ 0 ║

╟────────┼───────────┼────────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼─────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────╢

║ 4 │ -0.994960 │ -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 │ -0.206010 │ 0.502292 │ 0.219422 │ 0.215153 │ 0.670579 │ 0 ║

╟────────┼───────────┼────────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼─────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────╢

║ 284802 │ 1.034951 │ -11.881118 │ 10.071785 │ -9.834783 │ -2.066656 │ -5.364473 │ -2.606837 │ -4.918215 │ 7.305334 │ 1.914428 │ ... │ 0.213454 │ 0.111864 │ 1.014480 │ -0.509348 │ 1.436807 │ 0.250034 │ 0.943651 │ 0.823731 │ -0.296653 │ 0 ║

╟────────┼───────────┼────────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼─────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────╢

║ 284803 │ 1.034963 │ -0.732789 │ -0.055080 │ 2.035030 │ -0.738589 │ 0.868229 │ 1.058415 │ 0.024330 │ 0.294869 │ 0.584800 │ ... │ 0.214205 │ 0.924384 │ 0.012463 │ -1.016226 │ -0.606624 │ -0.395255 │ 0.068472 │ -0.053527 │ 0.038986 │ 0 ║

╟────────┼───────────┼────────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼─────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────╢

║ 284804 │ 1.034975 │ 1.919565 │ -0.301254 │ -3.249640 │ -0.557828 │ 2.630515 │ 3.031260 │ -0.296827 │ 0.708417 │ 0.432454 │ ... │ 0.232045 │ 0.578229 │ -0.037501 │ 0.640134 │ 0.265745 │ -0.087371 │ 0.004455 │ -0.026561 │ 0.641096 │ 0 ║

╟────────┼───────────┼────────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼─────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────╢

║ 284805 │ 1.034975 │ -0.240440 │ 0.530483 │ 0.702510 │ 0.689799 │ -0.377961 │ 0.623708 │ -0.686180 │ 0.679145 │ 0.392087 │ ... │ 0.265245 │ 0.800049 │ -0.163298 │ 0.123205 │ -0.569159 │ 0.546668 │ 0.108821 │ 0.104533 │ -0.167680 │ 0 ║

╟────────┼───────────┼────────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼─────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────╢

║ 284806 │ 1.035022 │ -0.533413 │ -0.189733 │ 0.703337 │ -0.506271 │ -0.012546 │ -0.649617 │ 1.577006 │ -0.414650 │ 0.486180 │ ... │ 0.261057 │ 0.643078 │ 0.376777 │ 0.008797 │ -0.473649 │ -0.818267 │ -0.002415 │ 0.013649 │ 2.724796 │ 0 ║

╚════════╧═══════════╧════════════╧═══════════╧═══════════╧═══════════╧═══════════╧═══════════╧═══════════╧═══════════╧═══════════╧═════╧═══════════╧═══════════╧═══════════╧═══════════╧═══════════╧═══════════╧═══════════╧═══════════╧═══════════╧═══════╝

10 rows × 31 columns

Next, let's divide the data into features and targets. We also make the train-test split of the data:

y = dataset["Class"] # target

X = dataset.iloc[:,0:30]

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X, y, test\_size = 0.2, random\_state = 42)

X\_train.shape, X\_test.shape, y\_train.shape, y\_test.shape

Output:

((227845, 30), (56962, 30), (227845,), (56962,))

Let's import all the necessary libraries for the tutorial:

from sklearn.model\_selection import StratifiedKFold

from sklearn.model\_selection import GridSearchCV, cross\_val\_score, RandomizedSearchCV

kf = StratifiedKFold(n\_splits=5, random\_state = None, shuffle = False)

from imblearn.pipeline import make\_pipeline ## Create a Pipeline using the provided estimators .

from imblearn.under\_sampling import NearMiss ## perform Under-sampling based on NearMiss methods.

from imblearn.over\_sampling import SMOTE ## PerformOver-sampling class that uses SMOTE.

from sklearn.metrics import roc\_curve, roc\_auc\_score, accuracy\_score, recall\_score, precision\_score, f1\_score

from sklearn.linear\_model import LogisticRegression

from sklearn.svm import SVC

from sklearn.neighbors import KNeighborsClassifier

from sklearn.tree import DecisionTreeClassifier

from sklearn.ensemble import RandomForestClassifier

For analysing data, we need some libraries. In this section, we are importing all the required libraries like pandas, NumPy, matplotlib, plotly, seaborn, and word cloud that are required for data analysis. Check the below code to import all the required libraries.

Data Source

A good data source for credit card fraud detection should be Accurate, Complete, Covering the geographic area of interest, Accessible.

# Data Collection and Preprocessing:

* Data collection : With Credit Card Fraud Detection, this project demonstrates the modelling of a data collection using **machine learning**. Modeling prior credit card transactions with data from those that turned out to be fraudulent is part of the Credit Card Fraud Detection Problem. The model is then used to determine whether or not a new transaction is fraudulent.

# Data preprocessing: Analyzing the effect of data preprocessing techniques using machine learning algorithms on the diagnosis of fraud detection

# Exploratory Data Analysis ( EDA ):

# This case study is focused to give you an idea of applying Exploratory Data Analysis (EDA) in a real business scenario. In this case study, apart from applying the various Exploratory Data Analysis (EDA) techniques, you will also develop a basic understanding of risk analytics and understand how data can be utilized in order to minimise the risk of losing money while lending to customers.

# Feature Engineering:

**Feature engineering** is a crucial step in credit card fraud detection. It involves selecting and transforming the most relevant features from the dataset to improve the performance of machine learning models. [In credit card fraud detection, feature engineering can help identify patterns and anomalies in transaction data that are indicative of fraudulent activity](https://medium.com/dataman-in-ai/how-to-create-good-features-in-fraud-detection-de6562f249ef).

Advanced Regression Techniques:

# There are several machine learning techniques that can be used for credit card fraud detection. One such technique is **logistic regression**. [In a study, researchers investigated the use of logistic regression to detect fraudulent credit card transactions in an imbalanced dataset where only a small fraction of transactions are fraudulent](https://ieeexplore.ieee.org/document/10112302/).Another technique is **genetic algorithm (GA)** based feature selection. A recent paper proposed a machine learning based credit card fraud detection engine using the GA algorithm for feature selection. [After the optimized features are chosen, the proposed detection engine uses the following ML classifiers: Decision Tree (DT), Random Forest (RF), Logistic Regression (LR), Artificial Neural Network (ANN), and Naive Bayes (NB) 2](https://journalofbigdata.springeropen.com/articles/10.1186/s40537-022-00573-8). [The paper also demonstrated that their proposed approach outperforms existing systems](https://journalofbigdata.springeropen.com/articles/10.1186/s40537-022-00573-8).

# Model Interpretability:

Credit card fraud is a growing problem in the financial industry, with the potential to cause significant

financial losses to both customers and financial institutions. As a result, there has been a significant

amount of research in recent years on developing effective fraud detection systems. These systems rely on

a combination of statistical techniques, machine learning algorithms, and deep learning models to identify

fraudulent transactions.One of the most commonly used approaches for credit card fraud detection is

rule-based systems. These systems use predefined rules to identify transactions that are deemed suspicious.

However, rule-based systems have limitations, as they are only as good as the rules that have been

predefined, and they may not be able to detect new types of fraud. To overcome these limitations, machine

learning algorithms and statistical techniques have been applied to credit card fraud detection. These

techniques are based on analysing transaction-related data, such as the transaction amount, location, and

time, as well as other relevant factors, such as the customer’s transaction history and account details. In

recent years, deep learning models, such as convolutional neural networks (CNNs) and recurrent neural

networks (RNNs), have also been applied to credit card fraud detection. These models have shown

promising results in identifying fraudulent transactions by learning patterns in the data and improving the

accuracy of fraud detection. Overall, credit card fraud detection is a critical area of research in the financial

industry, with significant potential for improving fraud detection rates and reducing financial losse

Credit card fraud is a growing problem in the financial industry, with the potential to cause significant

financial losses to both customers and financial institutions. As a result, there has been a significant

amount of research in recent years on developing effective fraud detection systems. These systems rely on

a combination of statistical techniques, machine learning algorithms, and deep learning models to identify

fraudulent transactions.One of the most commonly used approaches for credit card fraud detection is

rule-based systems. These systems use predefined rules to identify transactions that are deemed suspicious.

However, rule-based systems have limitations, as they are only as good as the rules that have been

predefined, and they may not be able to detect new types of fraud. To overcome these limitations, machine

learning algorithms and statistical techniques have been applied to credit card fraud detection. These

techniques are based on analysing transaction-related data, such as the transaction amount, location, and

time, as well as other relevant factors, such as the customer’s transaction history and account details. In

recent years, deep learning models, such as convolutional neural networks (CNNs) and recurrent neural

networks (RNNs), have also been applied to credit card fraud detection. These models have shown

promising results in identifying fraudulent transactions by learning patterns in the data and improving the

accuracy of fraud detection. Overall, credit card fraud detection is a critical area of research in the financial

industry, with significant potential for improving fraud detection rates and reducing financial losse

Credit Card Fraud can be defined as a case where a person uses someone else’s credit card for personal reasons while the owner and the card-issuing authorities are unaware of the fact that the card is being used. Due to the rise and acceleration of E-Commerce, there has been a tremendous use of credit cards for online shopping which led to High amount of frauds related to credit cards. In the era of digitalization, the need to identify credit card frauds is necessary. Fraud detection involves monitoring and analyzing the behaviour of various users to estimate detect or avoid undesirable behaviour. To identify credit card fraud detection effectively, we need to understand the various technologies, algorithms and types involved in detecting credit card frauds. The algorithm can differentiate transactions which are fraudulent or not. Find fraud, they need to passed dataset and knowledge of the fraudulent transaction. They analyze the dataset and classify all transactions. Fraud detection involves monitoring the activities of populations of users to estimate, perceive or avoid objectionable behaviour, which consist of fraud, intrusion, and defaulting. Machine learning algorithms are employed to analyses all the authorized transactions and report the suspicious ones. These reports are investigated by professionals who contact the cardholders to confirm if the transaction was genuine or fraudulent. The investigators provide feedback to the automated system which is used to train and update the algorithm to eventually improve the fraud-detection performance over time.

Feature Importance Analysis:

Understanding which features or variables have the most significant impact on the model's predictions is often essential. Techniques like feature importance scores (e.g., Gini importance for decision trees or SHAP values) can help highlight the most influential features.

Partial Dependency Plots (PDP):

PDPs illustrate how a model's output changes with variations in a specific feature while holding other features constant. This allows users to see the relationship between individual features and the model's predictions.

LIME (Local Interpretable Model-Agnostic Explanations):

LIME is a technique that provides local interpretability for black-box models. It generates locally faithful explanations by training a simple, interpretable model on a subset of data around a specific prediction. This helps understand the model's behaviour in the vicinity of a particular data point.

SHAP (Shapley Additive Explanations):

SHAP values provide a unified measure of feature importance. They can explain how each feature contributes to a specific prediction and can also give a global view of feature importance.

Decision Trees and Rule-Based Models:

Decision trees are inherently interpretable. By visualizing the tree structure, users can see how decisions are made at each node based on specific features. Rule-based models, like decision sets, provide a set of easy-to-understand rules for predictions.

Counterfactual Explanations:

Counterfactual explanations offer insights into how changing specific feature values would have led to different model predictions. They help users understand what they can do differently to achieve a desired outcome.

Model-Agnostic Methods:

Techniques like SHAP, LIME, and anchor explanations work with any type of model. They provide a way to explain complex models, even if you don't know the internal workings of the model.

Visual Explanations:

Visualization tools can be used to present model interpretations in a more understandable and user-friendly manner. For instance, you can use heatmap visualizations to show how feature values influence predictions.

Global vs. Local Interpretability:

Models can provide global explanations that apply to the entire model or local explanations that focus on a specific prediction. Both types are important depending on the context and audience.

Regulatory Compliance:

In regulated industries like finance, ensuring that a model's behaviour aligns with legal and compliance requirements is critical. Transparent and interpretable models are often favoured in such scenarios.

Stakeholder Collaboration:

Collaboration between data scientists, domain experts, and business stakeholders can help bridge the gap between technical understanding and practical interpretation.

Documented Model Behaviour:

Clearly documenting a model's behaviour, including its input features and how it makes predictions, can aid in understanding and interpretation.

Model interpretability is vital not only for understanding model decisions but also for building trust and ensuring that models behave as expected, especially in applications where errors or misclassifications can have significant consequences, such as credit card fraud detection.

# Deployment and Prediction:

Deploy the chosen regression model to credit card fraud detection

Develop a user-friendly interface for users to input property features.

Program:

### Credit Card Fraud Detection

$ pip install sklearn==0.24.2 imbalanced-learn numpy pandas matplotlib seaborn

Let's import the necessary libraries:

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from matplotlib import gridspec

Now we read the data and try to understand each feature's meaning. The Python module [pandas](https://pandas.pydata.org/) provide us with the functions to read data. In the next step, we will read the data from our directory where the data is saved, and then we look at the first and last five rows of the data using head(), and tail() methods:

dataset = pd.read\_csv("creditcard.csv")

dataset.head().append(dataset.tail())

╔════════╤══════════╤════════════╤═══════════╤═══════════╤═══════════╤═══════════╤═══════════╤═══════════╤═══════════╤═══════════╤═════╤═══════════╤═══════════╤═══════════╤═══════════╤═══════════╤═══════════╤═══════════╤═══════════╤════════╤═══════╗

║ │ Time │ V1 │ V2 │ V3 │ V4 │ V5 │ V6 │ V7 │ V8 │ V9 │ ... │ V21 │ V22 │ V23 │ V24 │ V25 │ V26 │ V27 │ V28 │ Amount │ Class ║

╠════════╪══════════╪════════════╪═══════════╪═══════════╪═══════════╪═══════════╪═══════════╪═══════════╪═══════════╪═══════════╪═════╪═══════════╪═══════════╪═══════════╪═══════════╪═══════════╪═══════════╪═══════════╪═══════════╪════════╪═══════╣

║ 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 │ 0.128539 │ -0.189115 │ 0.133558 │ -0.021053 │ 149.62 │ 0 ║

╟────────┼──────────┼────────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼─────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼────────┼───────╢

║ 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 │ 0.167170 │ 0.125895 │ -0.008983 │ 0.014724 │ 2.69 │ 0 ║

╟────────┼──────────┼────────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼─────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼────────┼───────╢

║ 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 │ -0.327642 │ -0.139097 │ -0.055353 │ -0.059752 │ 378.66 │ 0 ║

╟────────┼──────────┼────────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼─────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼────────┼───────╢

║ 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 │ 0.647376 │ -0.221929 │ 0.062723 │ 0.061458 │ 123.50 │ 0 ║

╟────────┼──────────┼────────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼─────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼────────┼───────╢

║ 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 │ -0.206010 │ 0.502292 │ 0.219422 │ 0.215153 │ 69.99 │ 0 ║

╟────────┼──────────┼────────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼─────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼────────┼───────╢

║ 284802 │ 172786.0 │ -11.881118 │ 10.071785 │ -9.834783 │ -2.066656 │ -5.364473 │ -2.606837 │ -4.918215 │ 7.305334 │ 1.914428 │ ... │ 0.213454 │ 0.111864 │ 1.014480 │ -0.509348 │ 1.436807 │ 0.250034 │ 0.943651 │ 0.823731 │ 0.77 │ 0 ║

╟────────┼──────────┼────────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼─────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼────────┼───────╢

║ 284803 │ 172787.0 │ -0.732789 │ -0.055080 │ 2.035030 │ -0.738589 │ 0.868229 │ 1.058415 │ 0.024330 │ 0.294869 │ 0.584800 │ ... │ 0.214205 │ 0.924384 │ 0.012463 │ -1.016226 │ -0.606624 │ -0.395255 │ 0.068472 │ -0.053527 │ 24.79 │ 0 ║

╟────────┼──────────┼────────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼─────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼────────┼───────╢

║ 284804 │ 172788.0 │ 1.919565 │ -0.301254 │ -3.249640 │ -0.557828 │ 2.630515 │ 3.031260 │ -0.296827 │ 0.708417 │ 0.432454 │ ... │ 0.232045 │ 0.578229 │ -0.037501 │ 0.640134 │ 0.265745 │ -0.087371 │ 0.004455 │ -0.026561 │ 67.88 │ 0 ║

╟────────┼──────────┼────────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼─────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼────────┼───────╢

║ 284805 │ 172788.0 │ -0.240440 │ 0.530483 │ 0.702510 │ 0.689799 │ -0.377961 │ 0.623708 │ -0.686180 │ 0.679145 │ 0.392087 │ ... │ 0.265245 │ 0.800049 │ -0.163298 │ 0.123205 │ -0.569159 │ 0.546668 │ 0.108821 │ 0.104533 │ 10.00 │ 0 ║

╟────────┼──────────┼────────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼─────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼────────┼───────╢

║ 284806 │ 172792.0 │ -0.533413 │ -0.189733 │ 0.703337 │ -0.506271 │ -0.012546 │ -0.649617 │ 1.577006 │ -0.414650 │ 0.486180 │ ... │ 0.261057 │ 0.643078 │ 0.376777 │ 0.008797 │ -0.473649 │ -0.818267 │ -0.002415 │ 0.013649 │ 217.00 │ 0 ║

╚════════╧══════════╧════════════╧═══════════╧═══════════╧═══════════╧═══════════╧═══════════╧═══════════╧═══════════╧═══════════╧═════╧═══════════╧═══════════╧═══════════╧═══════════╧═══════════╧═══════════╧═══════════╧═══════════╧════════╧═══════╝

The Time is measured in seconds since the first transaction in the data collection. As a result, we may infer that this dataset contains all transactions recorded during two days. The features were prepared using PCA, so the physical interpretation of individual features does not make sense. 'Time' and 'Amount' are the only features that are not transformed to PCA. 'Class' is the response variable, and it has a value of 1 if there is fraud and 0 otherwise.

## Data Exploration and Visualization

Now we try to find out the relative proportion of valid and fraudulent credit card transactions:

print("Fraudulent Cases: " + str(len(dataset[dataset["Class"] == 1])))

print("Valid Transactions: " + str(len(dataset[dataset["Class"] == 0])))

print("Proportion of Fraudulent Cases: " + str(len(dataset[dataset["Class"] == 1])/ dataset.shape[0]))

data\_p = dataset.copy()

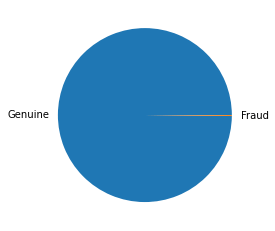
data\_p[" "] = np.where(data\_p["Class"] == 1 , "Fraud", "Genuine")

data\_p[" "].value\_counts().plot(kind="pie")

Fraudulent Cases: 492

Valid Transactions: 284315

Proportion of Fraudulent Cases: 0.001727485630620034



There is an imbalance in the data, with only 0.17% of the total cases being fraudulent.

Now we look at the distribution of the two named features in the dataset. For Time, it is clear that there was a particular duration in the day when most of the transactions took place:

f, axes = plt.subplots(1, 2, figsize=(18,4), sharex = True)

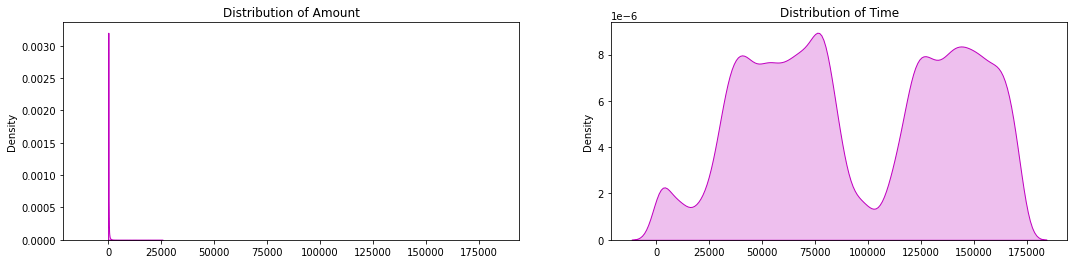
amount\_value = dataset['Amount'].values # values

time\_value = dataset['Time'].values # values

sns.distplot(amount\_value, hist=False, color="m", kde\_kws={"shade": True}, ax=axes[0]).set\_title('Distribution of Amount')

sns.distplot(time\_value, hist=False, color="m", kde\_kws={"shade": True}, ax=axes[1]).set\_title('Distribution of Time')

plt.show()



Let us check if there is any difference between valid transactions and fraudulent transactions:

print("Average Amount in a Fraudulent Transaction: " + str(dataset[dataset["Class"] == 1]["Amount"].mean()))

print("Average Amount in a Valid Transaction: " + str(dataset[dataset["Class"] == 0]["Amount"].mean()))

Average Amount in a Fraudulent Transaction: 122.21132113821133

Average Amount in a Valid Transaction: 88.29102242225574

As we can notice from this, the average money transaction for the fraudulent ones is more. It makes this problem crucial to deal with. Now let us try to understand the distribution of values in each feature. Let's start with the Amount:

print("Summary of the feature - Amount" + "\n-------------------------------")

print(dataset["Amount"].describe())

Summary of the feature - Amount

-------------------------------

count 284807.000000

mean 88.349619

std 250.120109

min 0.000000

25% 5.600000

50% 22.000000

75% 77.165000

max 25691.160000

Name: Amount, dtype: float64

The rest of the features don't have any physical interpretation and will be seen through histograms. Here the values are subgrouped according to class (valid or fraud):

data\_plot = dataset.copy()

amount = data\_plot['Amount']

data\_plot.drop(labels=['Amount'], axis=1, inplace = True)

data\_plot.insert(0, 'Amount', amount)

columns = data\_plot.iloc[:,0:30].columns

plt.figure(figsize=(12,30\*4))

grids = gridspec.GridSpec(30, 1)

for grid, index in enumerate(data\_plot[columns]):

ax = plt.subplot(grids[grid])

sns.distplot(data\_plot[index][data\_plot.Class == 1], hist=False, kde\_kws={"shade": True}, bins=50)

sns.distplot(data\_plot[index][data\_plot.Class == 0], hist=False, kde\_kws={"shade": True}, bins=50)

ax.set\_xlabel("")

ax.set\_title("Distribution of Column: " + str(index))

plt.show()

Some common data processing tasks include:

Credit card fraud detection involves a range of data processing tasks to identify and prevent fraudulent transactions. Here are some common data processing tasks in credit card fraud detection:

1. Data Collection:
   * Acquiring transaction data from various sources, including point-of-sale terminals, online payment gateways, and mobile applications.
2. Data Pre-processing:
   * Data cleaning to handle missing or inconsistent data.
   * Data transformation, such as normalizing or standardizing features.
   * Feature engineering to create new variables that may aid in fraud detection.
3. Data Integration:
   * Combining transaction data from multiple sources to create a comprehensive dataset.
4. Data Sampling:
   * Creating a balanced dataset by under sampling the majority class (legitimate transactions) or oversampling the minority class (fraudulent transactions) to avoid class imbalance issues.
5. Data Splitting:
   * Splitting the dataset into training, validation, and testing sets for model development and evaluation.
6. Exploratory Data Analysis (EDA):
   * Exploring the data to understand its distribution, patterns, and relationships.
   * Visualizing transaction statistics, such as transaction amounts, timestamps, and card usage.
7. Feature Selection:
   * Identifying and selecting the most relevant features for the model to reduce dimensionality and improve model performance.
8. Model Training:
   * Building and training machine learning models, such as logistic regression, decision trees, random forests, or deep learning models like neural networks.
   * Hyperparameter tuning to optimize model performance.
9. Anomaly Detection:
   * Applying anomaly detection techniques to identify unusual or suspicious patterns in transactions.
10. Real-time Monitoring:
    * Processing incoming transaction data in real-time to detect and flag potentially fraudulent transactions as they occur.
11. Model Evaluation:
    * Assessing model performance using metrics like accuracy, precision, recall, F1-score, and ROC AUC.
    * Cross-validation to ensure model robustness.
12. Threshold Optimization:
    * Setting decision thresholds to balance false positives and false negatives, depending on the desired level of fraud detection and false positive tolerance.
13. Post-processing:
    * Applying additional post-processing techniques to further refine model predictions, like clustering or outlier removal.
14. Reporting and Alerting:
    * Generating alerts and reports for transactions suspected of fraud for manual review by fraud analysts.
15. Continuous Improvement:
    * Regularly retraining and updating models to adapt to evolving fraud patterns and minimize false positives.
16. Data Storage and Archiving:
    * Storing historical transaction data for auditing, compliance, and future analysis.
17. Compliance:
    * Ensuring that the data processing and fraud detection processes comply with relevant regulations, such as GDPR or PCI DSS.

These data processing tasks play a crucial role in building effective credit card fraud detection systems that can protect cardholders and financial institutions from fraudulent activities.

How to overcome the challenges of loading and pre-processing a

detection dataset:

There are a number of things that can be done to overcome the

challenges of loading and pre-processing a house price dataset, including:

 Use a data pre-processing library:

There are a number of libraries available that can help with data

pre-processing tasks, such as handling missing values, encoding

categorical variables, and scaling the features.

 Carefully consider the specific needs of your model:

The best way to pre-process the data will depend on the specific

machine learning algorithm that you are using. It is important to

carefully consider the requirements of the algorithm and to pre-process

the data in a way that is compatible with the algorithm.

 Validate the pre-processed data:

It is important to validate the pre-processed data to ensure that it is

in a format that can be used by the machine learning algorithm and that

it is of high quality. This can be done by inspecting the data visually or

by using statistical methods.

 Data pre-processing is the process of cleaning, transforming, and

integrating data in order to make it ready for analysis.

 This may involve removing errors and inconsistencies, handling

missing values, transforming the data into a consistent format, and

scaling the data to a suitable range.

Certainly, let's delve into the development phase of credit card fraud detection in even more detail:

Data Collection and Pre-processing:

Data Gathering:

Collect historical transaction data, which may span several years. This data should include various attributes like transaction amount, time, location, merchant information, and cardholder details.

Data Sources:

Data can come from a variety of sources, including internal records, payment processors, and industry databases.

Data Sampling:

Depending on the size of the dataset, it might be necessary to take a random or stratified sample to manage computational resources effectively.

Data Cleaning and Transformation:

Data Cleaning:

Address issues such as missing values, outliers, and inconsistencies. This may involve imputing missing values, removing duplicates, and correcting erroneous entries.

Data Transformation:

Normalize or standardize numerical features to ensure that they are on the same scale. Categorical data may need to be one-hot encoded.

Time-based features may require feature engineering to extract relevant information, like day of the week or hour of the day.

Exploratory Data Analysis (EDA):

Conduct thorough EDA to gain insights into the data, understand the distribution of features, and identify potential patterns or anomalies that could be indicative of fraud.

Feature Engineering:

Feature Selection:

Use domain knowledge and statistical tests to select the most relevant features for fraud detection.

Feature Creation:

Generate new features that capture relationships or patterns in the data, such as aggregations, ratios, or time-based features.

Feature Scaling:

Apply scaling methods like Standardization (Z-score scaling) to numerical features.

Data Splitting:

Stratified Sampling:

Ensure that the training, validation, and test datasets have a balanced representation of both legitimate and fraudulent transactions.

Time-Based Split:

When dealing with chronological data, split the data based on time, ensuring that the model is tested on more recent data.

Model Selection:

Consider a variety of models including:

Logistic Regression

Decision Trees

Random Forests

Support Vector Machines

Neural Networks

Gradient Boosting Algorithms (XGBoost, LightGBM)

Experiment with ensemble methods for combining the strengths of multiple models.

Model Training:

Train the chosen model(s) on the training dataset using an appropriate loss function (e.g., binary cross-entropy for binary classification).

Hyperparameter Tuning:

Use techniques like grid search, random search, or Bayesian optimization to find the optimal hyperparameters for each model.

Model Evaluation:

Evaluate model performance on the validation dataset using various metrics, including but not limited to:

Accuracy

Precision

Recall

F1 Score

ROC-AUC

Generate a confusion matrix to understand the model's performance in terms of true positives, true negatives, false positives, and false negatives.

Threshold Selection:

Choose an appropriate threshold for the model's prediction probabilities to classify transactions as legitimate or fraudulent. The choice may involve considering business needs, such as minimizing false positives or maximizing true positives.

Model Deployment:

Deploy the trained model into a production environment where it can process real-time credit card transactions.

Implement necessary infrastructure for low-latency predictions and scalability.

Monitoring and Alerts:

Implement real-time monitoring of the fraud detection system and set up alerts for suspicious patterns or system failures.

Continuously analyze model drift and retrain the model as needed.

Feedback Loop:

Collect and incorporate feedback from fraud analysts and investigators to improve the model's performance and adapt to evolving fraud tactics.

Integration with Fraud Prevention Tools:

Integrate the fraud detection system with other security measures, such as two-factor authentication, biometric authentication, and real-time SMS or email notifications for cardholders.

Compliance and Reporting:

Ensure that the fraud detection system complies with legal and regulatory requirements (e.g., GDPR, PCI DSS).

Maintain detailed audit logs and reporting capabilities to track transactions and model performance.

User Education and Communication:

Educate cardholders about the importance of safeguarding their card information and teach them to recognize and report potential fraud.

Establish channels for customers to report suspicious transactions.

The development phase of credit card fraud detection is a complex and ongoing process that requires constant vigilance and adaptation to new fraud techniques. The quality of data, the choice of algorithms, and the effectiveness of monitoring are all critical factors in building a robust fraud detection system.

Key Features and Components:

Input Layer:

Fraud Guard takes a wide range of transaction attributes as input, which may include:

Transaction amount

Transaction date and time

Merchant information (e.g., merchant category, location)

Cardholder information (e.g., card type, country)

Historical transaction behaviour (e.g., card usage patterns)

Additional features derived from feature engineering

Feature Engineering:

Before input, the model processes data through a feature engineering pipeline to create new features and transform the raw data into a suitable format for analysis.

This may involve normalizing numerical features, one-hot encoding categorical features, and extracting time-based features.

Model Architecture:

Fraud Guard employs a gradient boosting ensemble model, specifically XGBoost.

It consists of an ensemble of decision trees that work together to make predictions. Each tree learns different aspects of the data, and their predictions are combined to make a final decision.

XGBoost is chosen for its ability to handle imbalanced datasets and adapt to changing patterns in fraud.

Training:

Fraud Guard is trained on a historical dataset of credit card transactions, which includes both legitimate and fraudulent examples.

The model learns to distinguish between the two classes by optimizing a binary cross-entropy loss function during the training process.

Hyperparameter tuning is employed to find the best model configuration.

Threshold Setting:

After training, a decision threshold is set to classify transactions as either legitimate or fraudulent. The threshold can be adjusted to achieve a balance between minimizing false positives and false negatives, depending on the risk tolerance of the financial institution.

Real-time Prediction:

Fraud Guard is deployed in a real-time processing environment, where it evaluates incoming credit card transactions as they occur.

For each transaction, it computes a probability score indicating the likelihood of fraud.

Alert System:

When a transaction is flagged as potentially fraudulent (i.e., the probability score exceeds the threshold), the system generates an alert for further investigation.

These alerts are sent to fraud analysts or cardholders, depending on the situation, allowing for timely action.

Feedback Loop:

The model continuously collects feedback from analysts and investigators to improve its performance and adapt to new fraud tactics.

Periodic retraining is scheduled to ensure the model remains up-to-date.

Evaluation and Performance Metrics:

Fraud Guard’s performance is assessed using various evaluation metrics, including:

Accuracy

Precision

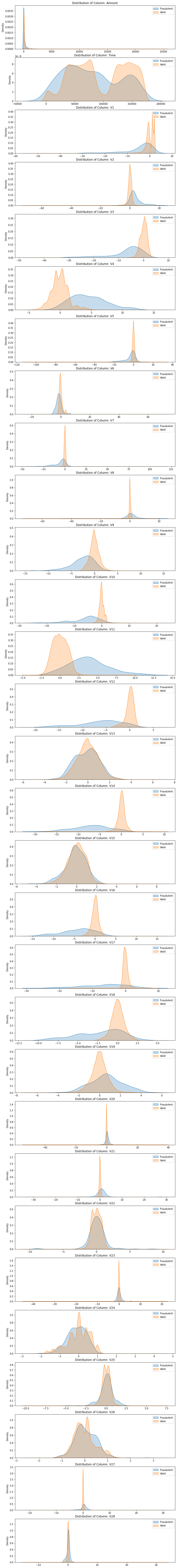
Recall

F1 Score

ROC-AUC

These metrics are monitored to ensure the model maintains a balance between minimizing false positives (legitimate transactions incorrectly classified as fraud) and false negatives (fraudulent transactions not detected).

Fraud Guard is a robust and adaptive model that helps financial institutions protect their customers from credit card fraud while minimizing disruptions to legitimate transactions. Its performance and effectiveness are continuously monitored and improved to stay ahead of evolving fraud tactics and maintain high levels of security.



Importance of loading and processing dataset:

Loading and processing datasets is a critical step in many data-driven tasks, especially in the fields of machine learning, data analysis, and artificial intelligence. The importance of this step can be summarized as follows:

1. **Data Quality Assurance:** Loading and processing datasets allow you to examine and clean the data. This is crucial because real-world data is often messy, containing missing values, outliers, and errors. Data pre-processing helps ensure the quality and reliability of the data used for analysis or modelling.
2. **Feature Engineering:** Processing data involves selecting relevant features and transforming them into a suitable format. Proper feature selection and engineering can significantly impact the performance of machine learning models. This step is critical for enhancing the predictive power of models.
3. **Normalization and Standardization:** Data pre-processing often includes techniques like normalization and standardization, which can make the data more suitable for various machine learning algorithms. This step helps ensure that the scales of different features are comparable.
4. **Dimensionality Reduction:** High-dimensional data can be challenging to work with. Dimensionality reduction techniques, like Principal Component Analysis (PCA) or feature selection, help reduce the number of features while retaining the most critical information. This can lead to faster training times and better model performance.
5. **Data Integration:** In many cases, data comes from various sources and in different formats. Loading and processing data enable you to integrate diverse datasets and prepare them for analysis or modelling. This is essential for holistic insights and decision-making.
6. **Data Understanding:** As you load and process a dataset, you gain a better understanding of its characteristics, such as the distribution of values, statistical properties, and relationships between variables. This understanding is crucial for choosing appropriate algorithms and making informed decisions.
7. **Data Visualization:** Pre-processing often includes data visualization, which helps you identify patterns, trends, and correlations in the data. Visualizing the data can provide insights that guide subsequent analysis and modelling tasks.
8. **Computational Efficiency:** Efficient data processing can significantly reduce the time and resources required for training machine learning models. Properly processed data can lead to faster convergence and reduced computational costs.
9. **Model Performance:** The quality of data pre-processing can directly impact the performance of machine learning models. Well-pre-processed data can lead to more accurate, robust, and interpretable models.
10. **Ethical Considerations:** Data pre-processing also plays a role in addressing ethical and privacy concerns. It includes steps to anonymize or de-identify sensitive information and ensure that models are not biased or discriminatory.

In summary, loading and processing datasets are foundational steps in the data analysis and machine learning pipeline. The quality and effectiveness of these steps can have a profound impact on the success of data-driven projects, making them crucial for extracting valuable insights, making informed decisions, and building accurate predictive models.

## Data Preparation

Since the features are created using PCA, feature selection is unnecessary as many features are tiny. Let's see if there are any missing values in the dataset:

dataset.isnull().shape[0]

print("Non-missing values: " + str(dataset.isnull().shape[0]))

print("Missing values: " + str(dataset.shape[0] - dataset.isnull().shape[0]))

Non-missing values: 284807

Missing values: 0

As there are no missing data, we turn to standardization. We standardize only Time and Amount using [RobustScaler](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.RobustScaler.html" \o "RobustScaler" \t "_blank):

from sklearn.preprocessing import RobustScaler

scaler = RobustScaler().fit(dataset[["Time", "Amount"]])

dataset[["Time", "Amount"]] = scaler.transform(dataset[["Time", "Amount"]])

dataset.head().append(dataset.tail())

As we saw previously, the Amount column has outliers, that's why we chose RobustScaler() as it's robust to outliers. Output:

╔════════╤═══════════╤════════════╤═══════════╤═══════════╤═══════════╤═══════════╤═══════════╤═══════════╤═══════════╤═══════════╤═════╤═══════════╤═══════════╤═══════════╤═══════════╤═══════════╤═══════════╤═══════════╤═══════════╤═══════════╤═══════╗

║ │ Time │ V1 │ V2 │ V3 │ V4 │ V5 │ V6 │ V7 │ V8 │ V9 │ ... │ V21 │ V22 │ V23 │ V24 │ V25 │ V26 │ V27 │ V28 │ Amount │ Class ║

╠════════╪═══════════╪════════════╪═══════════╪═══════════╪═══════════╪═══════════╪═══════════╪═══════════╪═══════════╪═══════════╪═════╪═══════════╪═══════════╪═══════════╪═══════════╪═══════════╪═══════════╪═══════════╪═══════════╪═══════════╪═══════╣

║ 0 │ -0.994983 │ -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 │ 0.128539 │ -0.189115 │ 0.133558 │ -0.021053 │ 1.783274 │ 0 ║

╟────────┼───────────┼────────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼─────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────╢

║ 1 │ -0.994983 │ 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 │ 0.167170 │ 0.125895 │ -0.008983 │ 0.014724 │ -0.269825 │ 0 ║

╟────────┼───────────┼────────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼─────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────╢

║ 2 │ -0.994972 │ -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 │ -0.327642 │ -0.139097 │ -0.055353 │ -0.059752 │ 4.983721 │ 0 ║

╟────────┼───────────┼────────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼─────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────╢

║ 3 │ -0.994972 │ -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 │ 0.647376 │ -0.221929 │ 0.062723 │ 0.061458 │ 1.418291 │ 0 ║

╟────────┼───────────┼────────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼─────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────╢

║ 4 │ -0.994960 │ -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 │ -0.206010 │ 0.502292 │ 0.219422 │ 0.215153 │ 0.670579 │ 0 ║

╟────────┼───────────┼────────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼─────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────╢

║ 284802 │ 1.034951 │ -11.881118 │ 10.071785 │ -9.834783 │ -2.066656 │ -5.364473 │ -2.606837 │ -4.918215 │ 7.305334 │ 1.914428 │ ... │ 0.213454 │ 0.111864 │ 1.014480 │ -0.509348 │ 1.436807 │ 0.250034 │ 0.943651 │ 0.823731 │ -0.296653 │ 0 ║

╟────────┼───────────┼────────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼─────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────╢

║ 284803 │ 1.034963 │ -0.732789 │ -0.055080 │ 2.035030 │ -0.738589 │ 0.868229 │ 1.058415 │ 0.024330 │ 0.294869 │ 0.584800 │ ... │ 0.214205 │ 0.924384 │ 0.012463 │ -1.016226 │ -0.606624 │ -0.395255 │ 0.068472 │ -0.053527 │ 0.038986 │ 0 ║

╟────────┼───────────┼────────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼─────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────╢

║ 284804 │ 1.034975 │ 1.919565 │ -0.301254 │ -3.249640 │ -0.557828 │ 2.630515 │ 3.031260 │ -0.296827 │ 0.708417 │ 0.432454 │ ... │ 0.232045 │ 0.578229 │ -0.037501 │ 0.640134 │ 0.265745 │ -0.087371 │ 0.004455 │ -0.026561 │ 0.641096 │ 0 ║

╟────────┼───────────┼────────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼─────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────╢

║ 284805 │ 1.034975 │ -0.240440 │ 0.530483 │ 0.702510 │ 0.689799 │ -0.377961 │ 0.623708 │ -0.686180 │ 0.679145 │ 0.392087 │ ... │ 0.265245 │ 0.800049 │ -0.163298 │ 0.123205 │ -0.569159 │ 0.546668 │ 0.108821 │ 0.104533 │ -0.167680 │ 0 ║

╟────────┼───────────┼────────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼─────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────────┼───────╢

║ 284806 │ 1.035022 │ -0.533413 │ -0.189733 │ 0.703337 │ -0.506271 │ -0.012546 │ -0.649617 │ 1.577006 │ -0.414650 │ 0.486180 │ ... │ 0.261057 │ 0.643078 │ 0.376777 │ 0.008797 │ -0.473649 │ -0.818267 │ -0.002415 │ 0.013649 │ 2.724796 │ 0 ║

╚════════╧═══════════╧════════════╧═══════════╧═══════════╧═══════════╧═══════════╧═══════════╧═══════════╧═══════════╧═══════════╧═════╧═══════════╧═══════════╧═══════════╧═══════════╧═══════════╧═══════════╧═══════════╧═══════════╧═══════════╧═══════╝

10 rows × 31 columns

Next, let's divide the data into features and targets. We also make the train-test split of the data:

y = dataset["Class"] # target

X = dataset.iloc[:,0:30]

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X, y, test\_size = 0.2, random\_state = 42)

X\_train.shape, X\_test.shape, y\_train.shape, y\_test.shape

Output:

((227845, 30), (56962, 30), (227845,), (56962,))

Let's import all the necessary libraries for the tutorial:

from sklearn.model\_selection import StratifiedKFold

from sklearn.model\_selection import GridSearchCV, cross\_val\_score, RandomizedSearchCV

kf = StratifiedKFold(n\_splits=5, random\_state = None, shuffle = False)

from imblearn.pipeline import make\_pipeline ## Create a Pipeline using the provided estimators .

from imblearn.under\_sampling import NearMiss ## perform Under-sampling based on NearMiss methods.

from imblearn.over\_sampling import SMOTE ## PerformOver-sampling class that uses SMOTE.

from sklearn.metrics import roc\_curve, roc\_auc\_score, accuracy\_score, recall\_score, precision\_score, f1\_score

from sklearn.linear\_model import LogisticRegression

from sklearn.svm import SVC

from sklearn.neighbors import KNeighborsClassifier

from sklearn.tree import DecisionTreeClassifier

from sklearn.ensemble import RandomForestClassifier

# Model Interpretability:

Credit card fraud is a growing problem in the financial industry, with the potential to cause significant

financial losses to both customers and financial institutions. As a result, there has been a significant

amount of research in recent years on developing effective fraud detection systems. These systems rely on

a combination of statistical techniques, machine learning algorithms, and deep learning models to identify

fraudulent transactions.One of the most commonly used approaches for credit card fraud detection is

rule-based systems. These systems use predefined rules to identify transactions that are deemed suspicious.

However, rule-based systems have limitations, as they are only as good as the rules that have been

predefined, and they may not be able to detect new types of fraud. To overcome these limitations, machine

learning algorithms and statistical techniques have been applied to credit card fraud detection. These

techniques are based on analysing transaction-related data, such as the transaction amount, location, and

time, as well as other relevant factors, such as the customer’s transaction history and account details. In

recent years, deep learning models, such as convolutional neural networks (CNNs) and recurrent neural

networks (RNNs), have also been applied to credit card fraud detection. These models have shown

promising results in identifying fraudulent transactions by learning patterns in the data and improving the

accuracy of fraud detection. Overall, credit card fraud detection is a critical area of research in the financial

industry, with significant potential for improving fraud detection rates and reducing financial losse

Credit card fraud is a growing problem in the financial industry, with the potential to cause significant

financial losses to both customers and financial institutions. As a result, there has been a significant

amount of research in recent years on developing effective fraud detection systems. These systems rely on

a combination of statistical techniques, machine learning algorithms, and deep learning models to identify

fraudulent transactions.One of the most commonly used approaches for credit card fraud detection is

rule-based systems. These systems use predefined rules to identify transactions that are deemed suspicious.

However, rule-based systems have limitations, as they are only as good as the rules that have been

predefined, and they may not be able to detect new types of fraud. To overcome these limitations, machine

learning algorithms and statistical techniques have been applied to credit card fraud detection. These

techniques are based on analysing transaction-related data, such as the transaction amount, location, and

time, as well as other relevant factors, such as the customer’s transaction history and account details. In

recent years, deep learning models, such as convolutional neural networks (CNNs) and recurrent neural

networks (RNNs), have also been applied to credit card fraud detection. These models have shown

promising results in identifying fraudulent transactions by learning patterns in the data and improving the

accuracy of fraud detection. Overall, credit card fraud detection is a critical area of research in the financial

industry, with significant potential for improving fraud detection rates and reducing financial losse

Credit Card Fraud can be defined as a case where a person uses someone else’s credit card for personal reasons while the owner and the card-issuing authorities are unaware of the fact that the card is being used. Due to the rise and acceleration of E-Commerce, there has been a tremendous use of credit cards for online shopping which led to High amount of frauds related to credit cards. In the era of digitalization, the need to identify credit card frauds is necessary. Fraud detection involves monitoring and analysing the behaviour of various users to estimate detect or avoid undesirable behaviour. To identify credit card fraud detection effectively, we need to understand the various technologies, algorithms and types involved in detecting credit card frauds. The algorithm can differentiate transactions which are fraudulent or not. Find fraud, they need to passed dataset and knowledge of the fraudulent transaction. They analyse the dataset and classify all transactions. Fraud detection involves monitoring the activities of populations of users to estimate, perceive or avoid objectionable behaviour, which consist of fraud, intrusion, and defaulting. Machine learning algorithms are employed to analyses all the authorized transactions and report the suspicious ones. These reports are investigated by professionals who contact the cardholders to confirm if the transaction was genuine or fraudulent. The investigators provide feedback to the automated system which is used to train and update the algorithm to eventually improve the fraud-detection performance over time.

### Business Understanding

Credit Card Fraud Detection is a classic class-imbalance problem where the number of fraud transactions is much lesser than the number of legitimate transaction for any bank. Most of the approaches involve building model on such imbalanced data, and thus fails to produce results on real-time new data because of overfitting on training data and a bias towards the majoritarian class of legitimate transactions. Thus, we can see this as an anomaly detection problem.

1. What time does the Credit Card Frauds usually take place?
2. What are the general trends of amounts for Credit Card Fraud Transactions?
3. How do we balance the data to not let the model overfit on legitimate transactions?

# Importing Required Libraries  
import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
import seaborn as sns  
  
from sklearn.linear\_model import LogisticRegression  
from sklearn.svm import SVC  
from sklearn.neighbors import KNeighborsClassifier  
from sklearn.tree import DecisionTreeClassifier  
from sklearn.ensemble import RandomForestClassifier  
from sklearn.linear\_model import SGDClassifier  
  
from mlxtend.plotting import plot\_learning\_curves  
from sklearn.model\_selection import train\_test\_split  
from imblearn.over\_sampling import SMOTE  
from sklearn.metrics import precision\_score, recall\_score, f1\_score, roc\_auc\_score, accuracy\_score, classification\_report  
from sklearn.model\_selection import KFold, StratifiedKFold  
from sklearn.preprocessing import StandardScaler  
from sklearn.pipeline import Pipeline  
from sklearn.model\_selection import GridSearchCV  
from sklearn.metrics import make\_scorer, matthews\_corrcoef  
  
import warnings  
warnings.filterwarnings("ignore")

### Data Understanding

The Dataset we use is the Kaggle Credit Card Fraud Detection Dataset enlisted in the following link: Link

* The Data has 32 features from V1-V28 which are unknown for confidentiality, TIme, Amount and Class
* The input features are V1-V28, Time and Amount
* The target variable is Class
* The Data does not have any missing values as evident from the below mentioned code, thus need not be handled
* The Data consists of all numerical features, and only the Target Variable Class is a categorical feature.
  + Class 0: Legitimate Transaction
  + Class 1: Fraud Transaction

# Read Data into a Dataframe  
df = pd.read\_csv('creditcard.csv')

df

Time V1 V2 V3 V4 V5 \  
0 0.0 -1.359807 -0.072781 2.536347 1.378155 -0.338321   
1 0.0 1.191857 0.266151 0.166480 0.448154 0.060018   
2 1.0 -1.358354 -1.340163 1.773209 0.379780 -0.503198   
3 1.0 -0.966272 -0.185226 1.792993 -0.863291 -0.010309   
4 2.0 -1.158233 0.877737 1.548718 0.403034 -0.407193   
... ... ... ... ... ... ...   
284802 172786.0 -11.881118 10.071785 -9.834783 -2.066656 -5.364473   
284803 172787.0 -0.732789 -0.055080 2.035030 -0.738589 0.868229   
284804 172788.0 1.919565 -0.301254 -3.249640 -0.557828 2.630515   
284805 172788.0 -0.240440 0.530483 0.702510 0.689799 -0.377961   
284806 172792.0 -0.533413 -0.189733 0.703337 -0.506271 -0.012546   
  
 V6 V7 V8 V9 ... V21 V22 \  
0 0.462388 0.239599 0.098698 0.363787 ... -0.018307 0.277838   
1 -0.082361 -0.078803 0.085102 -0.255425 ... -0.225775 -0.638672   
2 1.800499 0.791461 0.247676 -1.514654 ... 0.247998 0.771679   
3 1.247203 0.237609 0.377436 -1.387024 ... -0.108300 0.005274   
4 0.095921 0.592941 -0.270533 0.817739 ... -0.009431 0.798278   
... ... ... ... ... ... ... ...   
284802 -2.606837 -4.918215 7.305334 1.914428 ... 0.213454 0.111864   
284803 1.058415 0.024330 0.294869 0.584800 ... 0.214205 0.924384   
284804 3.031260 -0.296827 0.708417 0.432454 ... 0.232045 0.578229   
284805 0.623708 -0.686180 0.679145 0.392087 ... 0.265245 0.800049   
284806 -0.649617 1.577006 -0.414650 0.486180 ... 0.261057 0.643078   
  
 V23 V24 V25 V26 V27 V28 Amount \  
0 -0.110474 0.066928 0.128539 -0.189115 0.133558 -0.021053 149.62   
1 0.101288 -0.339846 0.167170 0.125895 -0.008983 0.014724 2.69   
2 0.909412 -0.689281 -0.327642 -0.139097 -0.055353 -0.059752 378.66   
3 -0.190321 -1.175575 0.647376 -0.221929 0.062723 0.061458 123.50   
4 -0.137458 0.141267 -0.206010 0.502292 0.219422 0.215153 69.99   
... ... ... ... ... ... ... ...   
284802 1.014480 -0.509348 1.436807 0.250034 0.943651 0.823731 0.77   
284803 0.012463 -1.016226 -0.606624 -0.395255 0.068472 -0.053527 24.79   
284804 -0.037501 0.640134 0.265745 -0.087371 0.004455 -0.026561 67.88   
284805 -0.163298 0.123205 -0.569159 0.546668 0.108821 0.104533 10.00   
284806 0.376777 0.008797 -0.473649 -0.818267 -0.002415 0.013649 217.00   
  
 Class   
0 0   
1 0   
2 0   
3 0   
4 0   
... ...   
284802 0   
284803 0   
284804 0   
284805 0   
284806 0   
  
[284807 rows x 31 columns]

### Data Preparation

* The Data does not have any missing values and hence, need not be handled.
* The Data has only Target Variable Class as the categorical variable.
* Remaining Features are numerical and need to be only standardized for comparison after balancing the dataset
* The mean of the amount of money in transactions is 88.34
* The standard deviation of amount of money in transactions is 250.12
* The time is distributed throughout the data equitably and hence, serves as an independent feature
* It is best to not remove or drop any data or features in this case and try to tune the model assuming them as independent features initially

# Describe Data  
df.describe()

Time V1 V2 V3 V4 \  
count 284807.000000 2.848070e+05 2.848070e+05 2.848070e+05 2.848070e+05   
mean 94813.859575 1.165980e-15 3.416908e-16 -1.373150e-15 2.086869e-15   
std 47488.145955 1.958696e+00 1.651309e+00 1.516255e+00 1.415869e+00   
min 0.000000 -5.640751e+01 -7.271573e+01 -4.832559e+01 -5.683171e+00   
25% 54201.500000 -9.203734e-01 -5.985499e-01 -8.903648e-01 -8.486401e-01   
50% 84692.000000 1.810880e-02 6.548556e-02 1.798463e-01 -1.984653e-02   
75% 139320.500000 1.315642e+00 8.037239e-01 1.027196e+00 7.433413e-01   
max 172792.000000 2.454930e+00 2.205773e+01 9.382558e+00 1.687534e+01   
  
 V5 V6 V7 V8 V9 \  
count 2.848070e+05 2.848070e+05 2.848070e+05 2.848070e+05 2.848070e+05   
mean 9.604066e-16 1.490107e-15 -5.556467e-16 1.177556e-16 -2.406455e-15   
std 1.380247e+00 1.332271e+00 1.237094e+00 1.194353e+00 1.098632e+00   
min -1.137433e+02 -2.616051e+01 -4.355724e+01 -7.321672e+01 -1.343407e+01   
25% -6.915971e-01 -7.682956e-01 -5.540759e-01 -2.086297e-01 -6.430976e-01   
50% -5.433583e-02 -2.741871e-01 4.010308e-02 2.235804e-02 -5.142873e-02   
75% 6.119264e-01 3.985649e-01 5.704361e-01 3.273459e-01 5.971390e-01   
max 3.480167e+01 7.330163e+01 1.205895e+02 2.000721e+01 1.559499e+01   
  
 ... V21 V22 V23 V24 \  
count ... 2.848070e+05 2.848070e+05 2.848070e+05 2.848070e+05   
mean ... 1.656562e-16 -3.444850e-16 2.578648e-16 4.471968e-15   
std ... 7.345240e-01 7.257016e-01 6.244603e-01 6.056471e-01   
min ... -3.483038e+01 -1.093314e+01 -4.480774e+01 -2.836627e+00   
25% ... -2.283949e-01 -5.423504e-01 -1.618463e-01 -3.545861e-01   
50% ... -2.945017e-02 6.781943e-03 -1.119293e-02 4.097606e-02   
75% ... 1.863772e-01 5.285536e-01 1.476421e-01 4.395266e-01   
max ... 2.720284e+01 1.050309e+01 2.252841e+01 4.584549e+00   
  
 V25 V26 V27 V28 Amount \  
count 2.848070e+05 2.848070e+05 2.848070e+05 2.848070e+05 284807.000000   
mean 5.340915e-16 1.687098e-15 -3.666453e-16 -1.220404e-16 88.349619   
std 5.212781e-01 4.822270e-01 4.036325e-01 3.300833e-01 250.120109   
min -1.029540e+01 -2.604551e+00 -2.256568e+01 -1.543008e+01 0.000000   
25% -3.171451e-01 -3.269839e-01 -7.083953e-02 -5.295979e-02 5.600000   
50% 1.659350e-02 -5.213911e-02 1.342146e-03 1.124383e-02 22.000000   
75% 3.507156e-01 2.409522e-01 9.104512e-02 7.827995e-02 77.165000   
max 7.519589e+00 3.517346e+00 3.161220e+01 3.384781e+01 25691.160000   
  
 Class   
count 284807.000000   
mean 0.001727   
std 0.041527   
min 0.000000   
25% 0.000000   
50% 0.000000   
75% 0.000000   
max 1.000000   
  
[8 rows x 31 columns]

df.columns

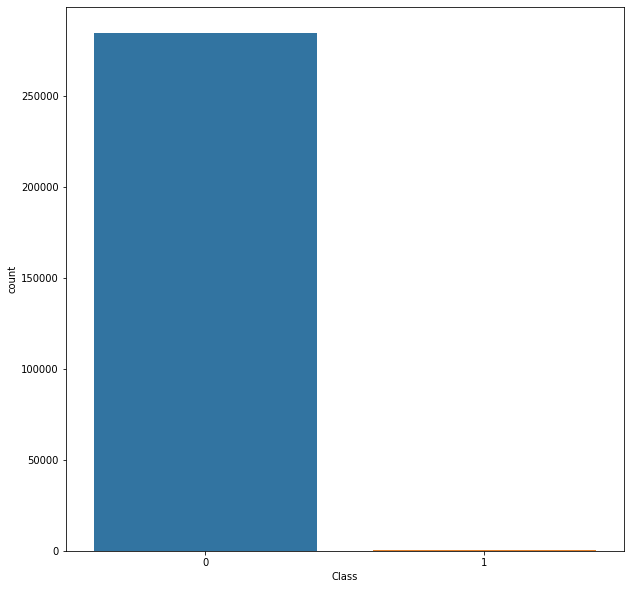
Index(['Time', 'V1', 'V2', 'V3', 'V4', 'V5', 'V6', 'V7', 'V8', 'V9', 'V10',  
 'V11', 'V12', 'V13', 'V14', 'V15', 'V16', 'V17', 'V18', 'V19', 'V20',  
 'V21', 'V22', 'V23', 'V24', 'V25', 'V26', 'V27', 'V28', 'Amount',  
 'Class'],  
 dtype='object')

df.isna().sum()

Time 0  
V1 0  
V2 0  
V3 0  
V4 0  
V5 0  
V6 0  
V7 0  
V8 0  
V9 0  
V10 0  
V11 0  
V12 0  
V13 0  
V14 0  
V15 0  
V16 0  
V17 0  
V18 0  
V19 0  
V20 0  
V21 0  
V22 0  
V23 0  
V24 0  
V25 0  
V26 0  
V27 0  
V28 0  
Amount 0  
Class 0  
dtype: int64

def countplot\_data(data, feature):  
 '''  
 Method to compute countplot of given dataframe  
 Parameters:  
 data(pd.Dataframe): Input Dataframe  
 feature(str): Feature in Dataframe  
 '''  
 plt.figure(figsize=(10,10))  
 sns.countplot(x=feature, data=data)  
 plt.show()  
  
def pairplot\_data\_grid(data, feature1, feature2, target):  
 '''  
 Method to construct pairplot of the given feature wrt data  
 Parameters:  
 data(pd.DataFrame): Input Dataframe  
 feature1(str): First Feature for Pair Plot  
 feature2(str): Second Feature for Pair Plot  
 target: Target or Label (y)  
 '''  
  
 sns.FacetGrid(data, hue=target, size=6).map(plt.scatter, feature1, feature2).add\_legend()  
 plt.show()

countplot\_data(df, df.Class)



### Insights:

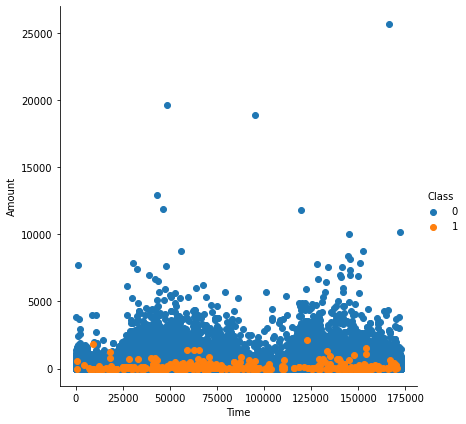
* The Dataset has 32 columns with unknown features labelled V1 to V28, Time, Amount and Class
* The target variable is 'Class' and rest of the variables are input features
* The Class has the following values:
  + 0: Legitimate Transactions
  + 1: Fraud Transactions
* The Dataset is highly imbalanced as evident from the countplot with majoritarian class label '0' and minority class label '1'
* Thus, if we run the model on such imbalanced data we may end up highly overfitting it on the data and resulting in non-deployable model
* Hence, we will perform Synthetic Minority Oversampling on the data to balance it out as shown later after exploring other features.

### What is relationship of fraud transactions with amount of money?

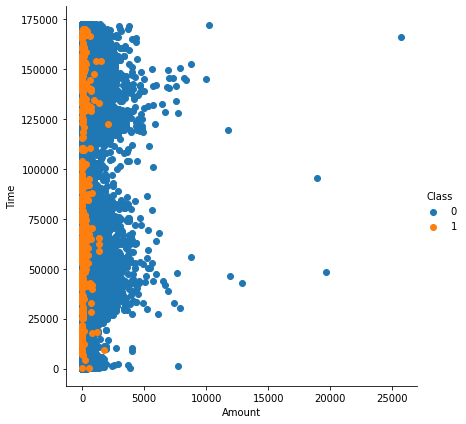
Let us try to determine the nature of transactions which are fraud and obtain a relevant set of the same with respect to their amount.

* We hypothesise based on our scatter plot that all fraud transactions occur for an amount less than 2500.

pairplot\_data\_grid(df, "Time", "Amount", "Class")



pairplot\_data\_grid(df, "Amount", "Time", "Class")



### Insights:

* It can be observed that the fraud transactions are generally not above an amount of 2500.
* It can also be observed that the fraud transactions are evenly distributed about time.
* Let us try to prove it

amount\_more = 0  
amount\_less = 0  
for i in range(df\_refine.shape[0]):  
 if(df\_refine.iloc[i]["Amount"] < 2500):  
 amount\_less += 1  
 else:  
 amount\_more += 1  
print(amount\_more)  
print(amount\_less)

449  
284358

percentage\_less = (amount\_less/df.shape[0])\*100  
percentage\_less

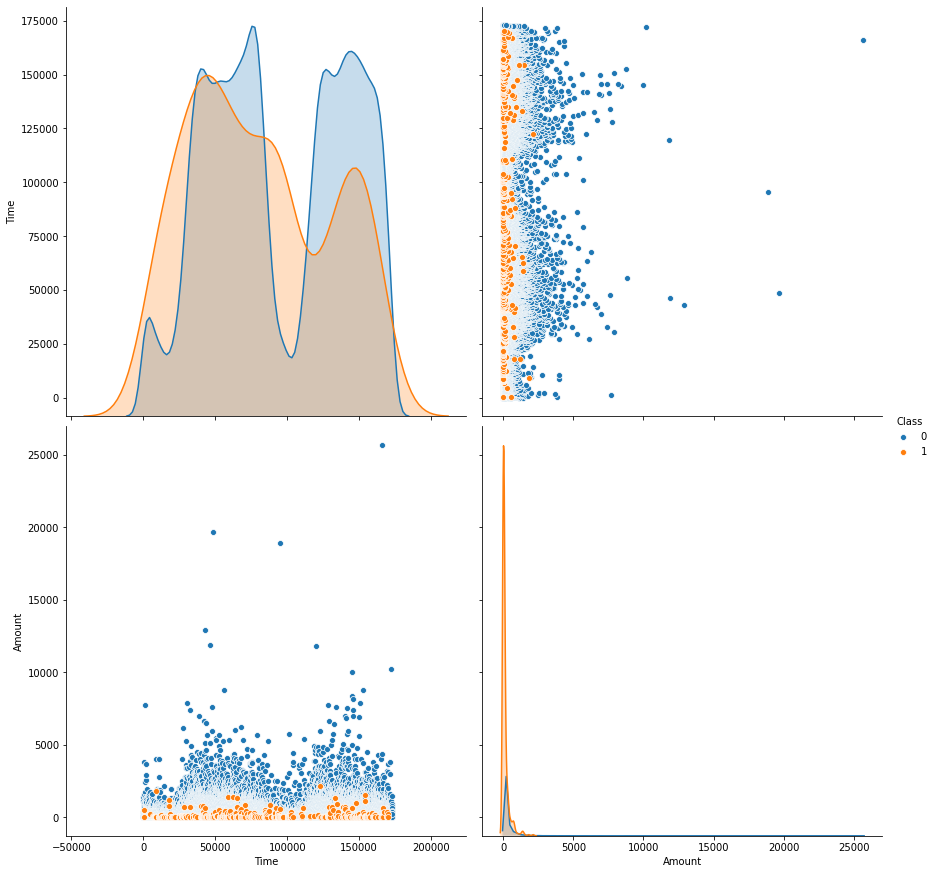
99.84234938045763

Hence, we observe that the 99.85% of transactions amount to less than 2500. Let us see how many of these are fraud and others legitimate

fraud = 0  
legitimate = 1  
for i in range(df\_refine.shape[0]):  
 if(df\_refine.iloc[i]["Amount"]<2500):  
 if(df\_refine.iloc[i]["Class"] == 0):  
 legitimate += 1  
 else:  
 fraud+=1  
print(fraud)  
print(legitimate)

492  
283867

df\_refine = df[["Time", "Amount", "Class"]]  
sns.pairplot(df\_refine, hue="Class", size=6)  
plt.show()



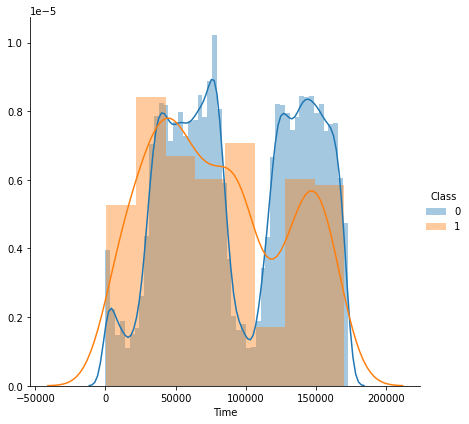
df.Class.value\_counts()

0 284315  
1 492  
Name: Class, dtype: int64

Thus, we can conclude that since the number of fraud transaction below the amount of 2500 is same as the number of total fraud transactions. Hence, all fraud transactions are less than 2500.

### What is the relationship between Time and Transactions?

sns.FacetGrid(df\_refine, hue="Class", size=6).map(sns.distplot,"Time").add\_legend()  
plt.show()



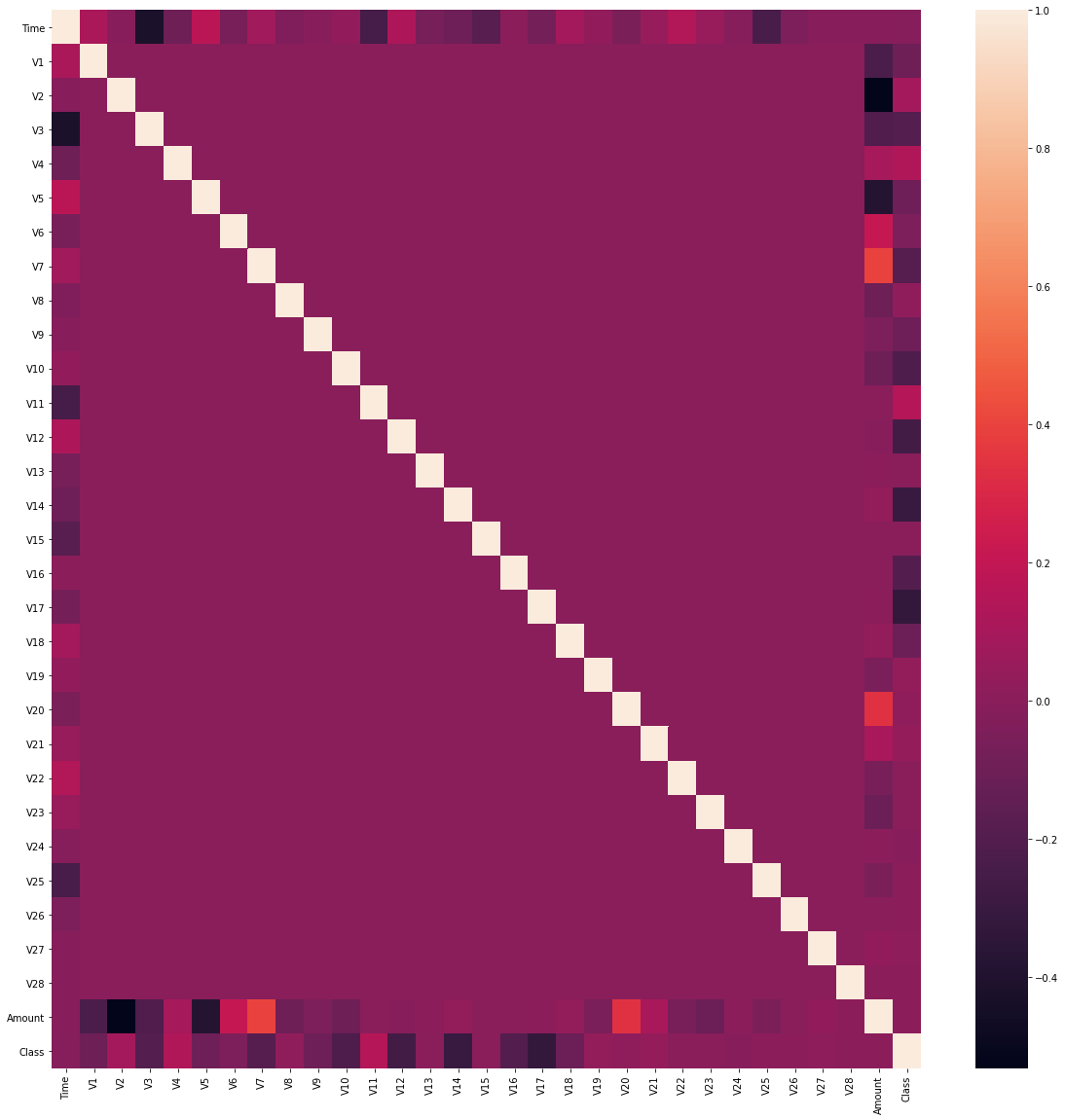
From the above distribution plot, it is clear that the fraudulent transactions are spread throughout the time period

### Modelling

* Study the Feature Correlations of the given data
* Plot a Heatmap
* Run Grid Search on the Data
* Fine Tune the Classifiers
* Create Pipelines for evaluation

plt.figure(figsize=(20,20))  
df\_corr = df.corr()  
sns.heatmap(df\_corr)

<matplotlib.axes.\_subplots.AxesSubplot at 0x7f397f77d700>



# Create Train and Test Data in ratio 70:30  
X = df.drop(labels='Class', axis=1) # Features  
y = df.loc[:,'Class'] # Target Variable  
  
  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=1, stratify=y)

### How will you balance the fraud and legitimate transactions in data?

# Use Synthetic Minority Oversampling  
sm = SMOTE(random\_state=42)  
X\_res, y\_res = sm.fit\_resample(X\_train, y\_train)

from sklearn.feature\_selection import mutual\_info\_classif  
mutual\_infos = pd.Series(data=mutual\_info\_classif(X\_res, y\_res, discrete\_features=False, random\_state=1), index=X\_train.columns)

mutual\_infos.sort\_values(ascending=False)

V14 0.535037  
V10 0.464777  
V12 0.456051  
V17 0.438193  
V4 0.427426  
V11 0.404044  
Amount 0.392941  
V3 0.387191  
V16 0.335318  
V7 0.304175  
V2 0.291492  
V9 0.256679  
Time 0.247989  
V21 0.235031  
V27 0.229915  
V1 0.220743  
V18 0.198264  
V8 0.174393  
V6 0.171974  
V28 0.170493  
V5 0.157362  
V20 0.107488  
V19 0.099837  
V23 0.067332  
V24 0.063567  
V26 0.046973  
V25 0.031607  
V22 0.031539  
V13 0.024931  
V15 0.022442  
dtype: float64

sns.countplot(y\_res)

<matplotlib.axes.\_subplots.AxesSubplot at 0x7f1e74ae2ac0>



Hence, we can say that the most correlated features after resolving class imbalance using Synthetic Minority Oversampling are V14, V10, V4, V12 and V17.

### Evaluation

We make use of AUC-ROC Score, Classification Report, Accuracy and F1-Score to evaluate the performance of the classifiers

# Evaluation of Classifiers  
def grid\_eval(grid\_clf):  
 """  
 Method to Compute the best score and parameters computed by grid search  
 Parameter:  
 grid\_clf: The Grid Search Classifier   
 """  
 print("Best Score", grid\_clf.best\_score\_)  
 print("Best Parameter", grid\_clf.best\_params\_)  
   
def evaluation(y\_test, grid\_clf, X\_test):  
 """  
 Method to compute the following:  
 1. Classification Report  
 2. F1-score  
 3. AUC-ROC score  
 4. Accuracy  
 Parameters:  
 y\_test: The target variable test set  
 grid\_clf: Grid classifier selected  
 X\_test: Input Feature Test Set  
 """  
 y\_pred = grid\_clf.predict(X\_test)  
 print('CLASSIFICATION REPORT')  
 print(classification\_report(y\_test, y\_pred))  
   
 print('AUC-ROC')  
 print(roc\_auc\_score(y\_test, y\_pred))  
   
 print('F1-Score')  
 print(f1\_score(y\_test, y\_pred))  
   
 print('Accuracy')  
 print(accuracy\_score(y\_test, y\_pred))

# The parameters of each classifier are different  
# Hence, we do not make use of a single method and this is not to violate DRY Principles  
# We set pipelines for each classifier unique with parameters  
param\_grid\_sgd = [{  
 'model\_\_loss': ['log'],  
 'model\_\_penalty': ['l1', 'l2'],  
 'model\_\_alpha': np.logspace(start=-3, stop=3, num=20)  
}, {  
 'model\_\_loss': ['hinge'],  
 'model\_\_alpha': np.logspace(start=-3, stop=3, num=20),  
 'model\_\_class\_weight': [None, 'balanced']  
}]  
  
pipeline\_sgd = Pipeline([  
 ('scaler', StandardScaler(copy=False)),  
 ('model', SGDClassifier(max\_iter=1000, tol=1e-3, random\_state=1, warm\_start=True))  
])  
  
MCC\_scorer = make\_scorer(matthews\_corrcoef)  
grid\_sgd = GridSearchCV(estimator=pipeline\_sgd, param\_grid=param\_grid\_sgd, scoring=MCC\_scorer, n\_jobs=-1, pre\_dispatch='2\*n\_jobs', cv=5, verbose=1, return\_train\_score=False)  
  
  
grid\_sgd.fit(X\_res, y\_res)

Fitting 5 folds for each of 80 candidates, totalling 400 fits

[Parallel(n\_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.  
[Parallel(n\_jobs=-1)]: Done 42 tasks | elapsed: 48.4s  
[Parallel(n\_jobs=-1)]: Done 192 tasks | elapsed: 3.2min  
[Parallel(n\_jobs=-1)]: Done 400 out of 400 | elapsed: 6.0min finished

GridSearchCV(cv=5, error\_score=nan,  
 estimator=Pipeline(memory=None,  
 steps=[('scaler',  
 StandardScaler(copy=False,  
 with\_mean=True,  
 with\_std=True)),  
 ('model',  
 SGDClassifier(alpha=0.0001,  
 average=False,  
 class\_weight=None,  
 early\_stopping=False,  
 epsilon=0.1, eta0=0.0,  
 fit\_intercept=True,  
 l1\_ratio=0.15,  
 learning\_rate='optimal',  
 loss='hinge',  
 max\_iter=1000,  
 n\_iter\_no\_change=...  
 3.35981829e-01, 6.95192796e-01, 1.43844989e+00, 2.97635144e+00,  
 6.15848211e+00, 1.27427499e+01, 2.63665090e+01, 5.45559478e+01,  
 1.12883789e+02, 2.33572147e+02, 4.83293024e+02, 1.00000000e+03]),  
 'model\_\_class\_weight': [None, 'balanced'],  
 'model\_\_loss': ['hinge']}],  
 pre\_dispatch='2\*n\_jobs', refit=True, return\_train\_score=False,  
 scoring=make\_scorer(matthews\_corrcoef), verbose=1)

grid\_eval(grid\_sgd)

Best Score 0.9560162686072134  
Best Parameter {'model\_\_alpha': 0.001, 'model\_\_loss': 'log', 'model\_\_penalty': 'l1'}

evaluation(y\_test, grid\_sgd, X\_test)

CLASSIFICATION REPORT  
 precision recall f1-score support  
  
 0 1.00 0.99 1.00 85295  
 1 0.14 0.91 0.25 148  
  
 accuracy 0.99 85443  
 macro avg 0.57 0.95 0.62 85443  
weighted avg 1.00 0.99 0.99 85443  
  
AUC-ROC  
0.9479720619851928  
F1-Score  
0.2460973370064279  
Accuracy  
0.990391254988706

pipeline\_rf = Pipeline([  
 ('model', RandomForestClassifier(n\_jobs=-1, random\_state=1))  
])  
param\_grid\_rf = {'model\_\_n\_estimators': [75]}  
grid\_rf = GridSearchCV(estimator=pipeline\_rf, param\_grid=param\_grid\_rf, scoring=MCC\_scorer, n\_jobs=-1, pre\_dispatch='2\*n\_jobs', cv=5, verbose=1, return\_train\_score=False)  
grid\_rf.fit(X\_res, y\_res)

Fitting 5 folds for each of 1 candidates, totalling 5 fits

[Parallel(n\_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.  
[Parallel(n\_jobs=-1)]: Done 5 out of 5 | elapsed: 10.5min finished

GridSearchCV(cv=5, error\_score=nan,  
 estimator=Pipeline(memory=None,  
 steps=[('model',  
 RandomForestClassifier(bootstrap=True,  
 ccp\_alpha=0.0,  
 class\_weight=None,  
 criterion='gini',  
 max\_depth=None,  
 max\_features='auto',  
 max\_leaf\_nodes=None,  
 max\_samples=None,  
 min\_impurity\_decrease=0.0,  
 min\_impurity\_split=None,  
 min\_samples\_leaf=1,  
 min\_samples\_split=2,  
 min\_weight\_fraction\_leaf=0.0,  
 n\_estimators=100,  
 n\_jobs=-1,  
 oob\_score=False,  
 random\_state=1,  
 verbose=0,  
 warm\_start=False))],  
 verbose=False),  
 iid='deprecated', n\_jobs=-1,  
 param\_grid={'model\_\_n\_estimators': [75]}, pre\_dispatch='2\*n\_jobs',  
 refit=True, return\_train\_score=False,  
 scoring=make\_scorer(matthews\_corrcoef), verbose=1)

grid\_eval(grid\_rf)

Best Score 0.9997538267139271  
Best Parameter {'model\_\_n\_estimators': 75}

evaluation(y\_test, grid\_rf, X\_test)

CLASSIFICATION REPORT  
 precision recall f1-score support  
  
 0 1.00 1.00 1.00 85295  
 1 0.90 0.86 0.88 148  
  
 accuracy 1.00 85443  
 macro avg 0.95 0.93 0.94 85443  
weighted avg 1.00 1.00 1.00 85443  
  
AUC-ROC  
0.9323445023075716  
F1-Score  
0.879725085910653  
Accuracy  
0.9995903701883126

pipeline\_lr = Pipeline([  
 ('model', LogisticRegression(random\_state=1))  
])  
param\_grid\_lr = {'model\_\_penalty': ['l2'],  
 'model\_\_class\_weight': [None, 'balanced']}  
grid\_lr = GridSearchCV(estimator=pipeline\_lr, param\_grid=param\_grid\_lr, scoring=MCC\_scorer, n\_jobs=-1, pre\_dispatch='2\*n\_jobs', cv=5, verbose=1, return\_train\_score=False)  
grid\_lr.fit(X\_res, y\_res)

Fitting 5 folds for each of 2 candidates, totalling 10 fits

[Parallel(n\_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.  
[Parallel(n\_jobs=-1)]: Done 10 out of 10 | elapsed: 28.0s finished

GridSearchCV(cv=5, error\_score=nan,  
 estimator=Pipeline(memory=None,  
 steps=[('model',  
 LogisticRegression(C=1.0,  
 class\_weight=None,  
 dual=False,  
 fit\_intercept=True,  
 intercept\_scaling=1,  
 l1\_ratio=None,  
 max\_iter=100,  
 multi\_class='auto',  
 n\_jobs=None,  
 penalty='l2',  
 random\_state=1,  
 solver='lbfgs',  
 tol=0.0001,  
 verbose=0,  
 warm\_start=False))],  
 verbose=False),  
 iid='deprecated', n\_jobs=-1,  
 param\_grid={'model\_\_class\_weight': [None, 'balanced'],  
 'model\_\_penalty': ['l2']},  
 pre\_dispatch='2\*n\_jobs', refit=True, return\_train\_score=False,  
 scoring=make\_scorer(matthews\_corrcoef), verbose=1)

grid\_eval(grid\_lr)

Best Score 0.959816277887179  
Best Parameter {'model\_\_class\_weight': None, 'model\_\_penalty': 'l2'}

evaluation(y\_test, grid\_lr, X\_test)

CLASSIFICATION REPORT  
 precision recall f1-score support  
  
 0 1.00 0.99 1.00 85295  
 1 0.15 0.91 0.26 148  
  
 accuracy 0.99 85443  
 macro avg 0.57 0.95 0.63 85443  
weighted avg 1.00 0.99 0.99 85443  
  
AUC-ROC  
0.948212404326479  
F1-Score  
0.2557251908396946  
Accuracy  
0.9908711070538253

pipeline\_knn = Pipeline([  
 ('model', KNeighborsClassifier(n\_neighbors=5))  
])  
param\_grid\_knn = {'model\_\_p': [2]}  
grid\_knn = GridSearchCV(estimator=pipeline\_knn, param\_grid=param\_grid\_knn, scoring=MCC\_scorer, n\_jobs=-1, pre\_dispatch='2\*n\_jobs', cv=5, verbose=1, return\_train\_score=False)  
grid\_knn.fit(X\_res, y\_res)

Fitting 5 folds for each of 1 candidates, totalling 5 fits

[Parallel(n\_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.  
[Parallel(n\_jobs=-1)]: Done 5 out of 5 | elapsed: 15.9min finished

GridSearchCV(cv=5, error\_score=nan,  
 estimator=Pipeline(memory=None,  
 steps=[('model',  
 KNeighborsClassifier(algorithm='auto',  
 leaf\_size=30,  
 metric='minkowski',  
 metric\_params=None,  
 n\_jobs=None,  
 n\_neighbors=5, p=2,  
 weights='uniform'))],  
 verbose=False),  
 iid='deprecated', n\_jobs=-1, param\_grid={'model\_\_p': [2]},  
 pre\_dispatch='2\*n\_jobs', refit=True, return\_train\_score=False,  
 scoring=make\_scorer(matthews\_corrcoef), verbose=1)

grid\_eval(grid\_knn)

Best Score 0.9980623930056313  
Best Parameter {'model\_\_p': 2}

evaluation(y\_test, grid\_knn, X\_test)

CLASSIFICATION REPORT  
 precision recall f1-score support  
  
 0 1.00 1.00 1.00 85295  
 1 0.50 0.86 0.63 148  
  
 accuracy 1.00 85443  
 macro avg 0.75 0.93 0.82 85443  
weighted avg 1.00 1.00 1.00 85443  
  
AUC-ROC  
0.9283095789968995  
F1-Score  
0.6318407960199005  
Accuracy  
0.9982678510820079

# The number of fraud transactions are very few comparted to legitimate transactions and it has to be balanced in order for a fair comparison to prevent the model from overfitting.Credit Card Fraud Detection

## Data frames

import pandas as pd

df = pd.read\_csv('creditcard.csv')

df

Time V1 V2 V3 V4 V5 \  
0 0.0 -1.359807 -0.072781 2.536347 1.378155 -0.338321   
1 0.0 1.191857 0.266151 0.166480 0.448154 0.060018   
2 1.0 -1.358354 -1.340163 1.773209 0.379780 -0.503198   
3 1.0 -0.966272 -0.185226 1.792993 -0.863291 -0.010309   
4 2.0 -1.158233 0.877737 1.548718 0.403034 -0.407193   
... ... ... ... ... ... ...   
284802 172786.0 -11.881118 10.071785 -9.834783 -2.066656 -5.364473   
284803 172787.0 -0.732789 -0.055080 2.035030 -0.738589 0.868229   
284804 172788.0 1.919565 -0.301254 -3.249640 -0.557828 2.630515   
284805 172788.0 -0.240440 0.530483 0.702510 0.689799 -0.377961   
284806 172792.0 -0.533413 -0.189733 0.703337 -0.506271 -0.012546   
  
 V6 V7 V8 V9 ... V21 V22 \  
0 0.462388 0.239599 0.098698 0.363787 ... -0.018307 0.277838   
1 -0.082361 -0.078803 0.085102 -0.255425 ... -0.225775 -0.638672   
2 1.800499 0.791461 0.247676 -1.514654 ... 0.247998 0.771679   
3 1.247203 0.237609 0.377436 -1.387024 ... -0.108300 0.005274   
4 0.095921 0.592941 -0.270533 0.817739 ... -0.009431 0.798278   
... ... ... ... ... ... ... ...   
284802 -2.606837 -4.918215 7.305334 1.914428 ... 0.213454 0.111864   
284803 1.058415 0.024330 0.294869 0.584800 ... 0.214205 0.924384   
284804 3.031260 -0.296827 0.708417 0.432454 ... 0.232045 0.578229   
284805 0.623708 -0.686180 0.679145 0.392087 ... 0.265245 0.800049   
284806 -0.649617 1.577006 -0.414650 0.486180 ... 0.261057 0.643078   
  
 V23 V24 V25 V26 V27 V28 Amount \  
0 -0.110474 0.066928 0.128539 -0.189115 0.133558 -0.021053 149.62   
1 0.101288 -0.339846 0.167170 0.125895 -0.008983 0.014724 2.69   
2 0.909412 -0.689281 -0.327642 -0.139097 -0.055353 -0.059752 378.66   
3 -0.190321 -1.175575 0.647376 -0.221929 0.062723 0.061458 123.50   
4 -0.137458 0.141267 -0.206010 0.502292 0.219422 0.215153 69.99   
... ... ... ... ... ... ... ...   
284802 1.014480 -0.509348 1.436807 0.250034 0.943651 0.823731 0.77   
284803 0.012463 -1.016226 -0.606624 -0.395255 0.068472 -0.053527 24.79   
284804 -0.037501 0.640134 0.265745 -0.087371 0.004455 -0.026561 67.88   
284805 -0.163298 0.123205 -0.569159 0.546668 0.108821 0.104533 10.00   
284806 0.376777 0.008797 -0.473649 -0.818267 -0.002415 0.013649 217.00   
  
 Class   
0 0   
1 0   
2 0   
3 0   
4 0   
... ...   
284802 0   
284803 0   
284804 0   
284805 0   
284806 0   
  
[284807 rows x 31 columns]

df['Class'].value\_counts()

0 284315  
1 492  
Name: Class, dtype: int64

### Data Pre-processing

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

scalar = StandardScaler()

X = df.drop('Class', axis=1)  
y = df.Class

X = scalar.fit\_transform(X)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=1)

## Modeling

from sklearn.svm import SVC

model\_svc = SVC()

model\_svc.fit(X\_train, y\_train)

SVC()

model\_svc.score(X\_train,y\_train)

0.9996752178015756

model\_svc.score(X\_test,y\_test)

0.999385555282469

y\_predict = model\_svc.predict(X\_test)

## Implementing Report

from sklearn.metrics import classification\_report , confusion\_matrix

import numpy as np

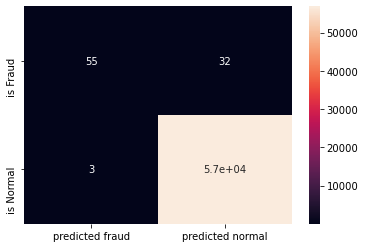
cm = np.array(confusion\_matrix(y\_test, y\_predict, labels=[1,0]))  
confusion = pd.DataFrame(cm, index=['is Fraud', 'is Normal'],columns=['predicted fraud','predicted normal'])  
confusion

predicted fraud predicted normal  
is Fraud 55 32  
is Normal 3 56872

import seaborn as sns

sns.heatmap(confusion, annot=True)

<matplotlib.axes.\_subplots.AxesSubplot at 0x1a0374fe948>



print(classification\_report(y\_test, y\_predict))

precision recall f1-score support  
  
 0 1.00 1.00 1.00 56875  
 1 0.95 0.63 0.76 87  
  
 accuracy 1.00 56962  
 macro avg 0.97 0.82 0.88 56962  
weighted avg 1.00 1.00 1.00 56962

from google.colab import drive  
drive.mount('/content/drive')

Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client\_id=947318989803-6bn6qk8qdgf4n4g3pfee6491hc0brc4i.apps.googleusercontent.com&redirect\_uri=urn%3aietf%3awg%3aoauth%3a2.0%3aoob&response\_type=code&scope=email%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdocs.test%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdrive%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdrive.photos.readonly%20https%3a%2f%2fwww.googleapis.com%2fauth%2fpeopleapi.readonly  
  
Enter your authorization code:  
··········  
Mounted at /content/drive

from matplotlib import pyplot as plt  
import warnings  
warnings.filterwarnings("ignore")  
import seaborn as sns  
import pandas as pd  
import numpy as np  
from imblearn.over\_sampling import SMOTE  
from sklearn.preprocessing import StandardScaler  
from sklearn.model\_selection import train\_test\_split  
import tensorflow as tf  
from tensorflow.keras.layers import Dense  
from tensorflow.keras.layers import Input  
from tensorflow.keras.models import Model

df = pd.read\_csv("/content/drive/My Drive/creditcard.csv", encoding="utf-8")

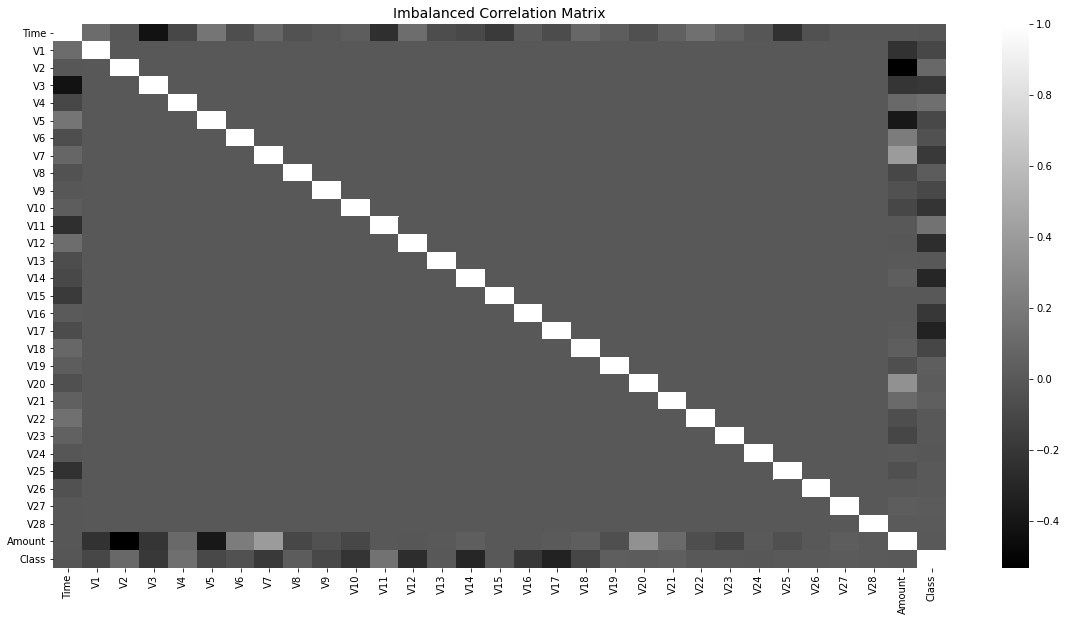
df

Time V1 V2 ... V28 Amount Class  
0 0.0 -1.359807 -0.072781 ... -0.021053 149.62 0  
1 0.0 1.191857 0.266151 ... 0.014724 2.69 0  
2 1.0 -1.358354 -1.340163 ... -0.059752 378.66 0  
3 1.0 -0.966272 -0.185226 ... 0.061458 123.50 0  
4 2.0 -1.158233 0.877737 ... 0.215153 69.99 0  
... ... ... ... ... ... ... ...  
284802 172786.0 -11.881118 10.071785 ... 0.823731 0.77 0  
284803 172787.0 -0.732789 -0.055080 ... -0.053527 24.79 0  
284804 172788.0 1.919565 -0.301254 ... -0.026561 67.88 0  
284805 172788.0 -0.240440 0.530483 ... 0.104533 10.00 0  
284806 172792.0 -0.533413 -0.189733 ... 0.013649 217.00 0  
  
[284807 rows x 31 columns]

df.head()

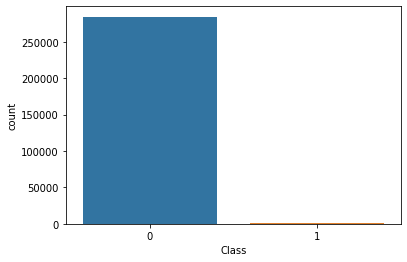
Time V1 V2 V3 ... V27 V28 Amount Class  
0 0.0 -1.359807 -0.072781 2.536347 ... 0.133558 -0.021053 149.62 0  
1 0.0 1.191857 0.266151 0.166480 ... -0.008983 0.014724 2.69 0  
2 1.0 -1.358354 -1.340163 1.773209 ... -0.055353 -0.059752 378.66 0  
3 1.0 -0.966272 -0.185226 1.792993 ... 0.062723 0.061458 123.50 0  
4 2.0 -1.158233 0.877737 1.548718 ... 0.219422 0.215153 69.99 0  
  
[5 rows x 31 columns]

fig, ax = plt.subplots(figsize=(20,10))  
corr = df.corr()  
sns.heatmap(corr, cmap="gray", ax=ax)  
ax.set\_title("Imbalanced Correlation Matrix", fontsize=14)  
plt.show()



sns.countplot(x='Class',data=df)

<matplotlib.axes.\_subplots.AxesSubplot at 0x7f690eb54f60>



sm = SMOTE(sampling\_strategy='minority', random\_state=7)  
resampled\_X, resampled\_Y = sm.fit\_resample(df.drop('Class', axis=1), df['Class'])  
oversampled\_df = pd.concat([pd.DataFrame(resampled\_X), pd.DataFrame(resampled\_Y)], axis=1)  
oversampled\_df.columns = df.columns  
oversampled\_df['Class'].value\_counts()

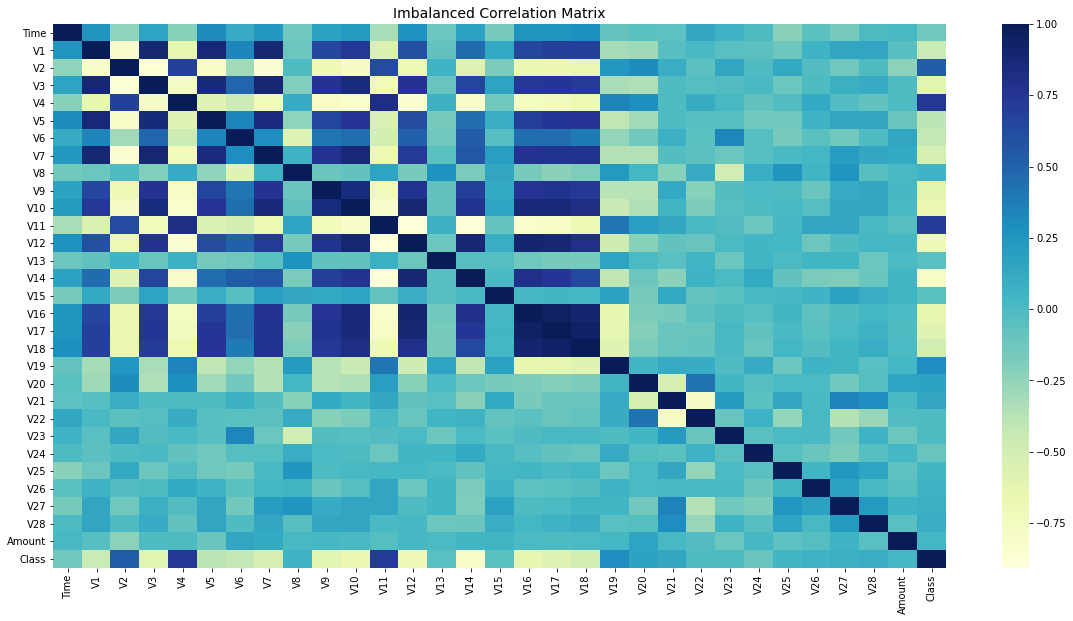
1 284315  
0 284315  
Name: Class, dtype: int64

sns.countplot(x='Class', data=oversampled\_df)

<matplotlib.axes.\_subplots.AxesSubplot at 0x7f6908f63a20>



fig, ax = plt.subplots(figsize=(20,10))   
corr = oversampled\_df.corr()  
sns.heatmap(corr, cmap='YlGnBu', annot\_kws={'size':30}, ax=ax)  
ax.set\_title("Imbalanced Correlation Matrix", fontsize=14)  
plt.show()



sc = StandardScaler()  
X = oversampled\_df.iloc[:, 1:-1].values  
y = oversampled\_df.iloc[:, -1].values  
y = y.reshape(-1, 1)  
print(X.shape, y.shape)  
  
X = sc.fit\_transform(X)  
print(X[0])

(568630, 29) (568630, 1)  
[ 0.20495125 -0.54573636 1.0045184 -0.30168224 0.31098779 0.6920209  
 0.55516986 -0.03644768 0.76125221 0.67923829 -0.92061643 0.57128753  
 -0.94634073 0.71704508 1.64724241 0.48636415 0.62706212 0.50679008  
 0.04977222 0.06379384 -0.14589103 0.24649907 -0.10441593 0.22564242  
 0.16596895 -0.48756152 0.05492719 -0.14984388 0.2455859 ]

x\_train, x\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

adam = tf.keras.optimizers.Adam(learning\_rate=0.0005)

x\_features = X.shape[1]  
y\_features = y.shape[1]

i = Input(shape=(x\_features,))  
  
x = Dense(64, activation='relu')(i)  
x = Dense(64, activation='relu')(x)  
o = Dense(y\_features, activation='sigmoid')(x)  
  
model = Model(i,o)  
model.compile(loss="binary\_crossentropy", metrics=['accuracy'], optimizer=adam)  
print(model.summary())  
callback = tf.keras.callbacks.EarlyStopping(  
 monitor='val\_loss', min\_delta=0, patience=10, verbose=0, mode='auto',  
 baseline=None, restore\_best\_weights=True  
)

Model: "model"  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
Layer (type) Output Shape Param #   
=================================================================  
input\_1 (InputLayer) [(None, 29)] 0   
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
dense (Dense) (None, 64) 1920   
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
dense\_1 (Dense) (None, 64) 4160   
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
dense\_2 (Dense) (None, 1) 65   
=================================================================  
Total params: 6,145  
Trainable params: 6,145  
Non-trainable params: 0  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
None

r = model.fit(x\_train, y\_train, epochs=100, batch\_size=512, verbose=1, validation\_data=(x\_test, y\_test), callbacks=[callback])

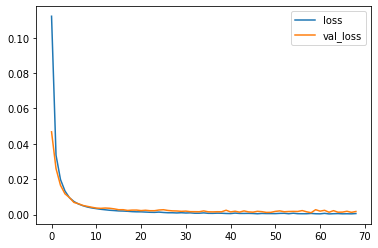
Epoch 1/100  
778/778 [==============================] - 3s 4ms/step - loss: 0.1122 - accuracy: 0.9588 - val\_loss: 0.0468 - val\_accuracy: 0.9844  
Epoch 2/100  
778/778 [==============================] - 3s 4ms/step - loss: 0.0336 - accuracy: 0.9883 - val\_loss: 0.0260 - val\_accuracy: 0.9929  
Epoch 3/100  
778/778 [==============================] - 3s 4ms/step - loss: 0.0198 - accuracy: 0.9939 - val\_loss: 0.0165 - val\_accuracy: 0.9945  
Epoch 4/100  
778/778 [==============================] - 3s 4ms/step - loss: 0.0133 - accuracy: 0.9965 - val\_loss: 0.0117 - val\_accuracy: 0.9974  
Epoch 5/100  
778/778 [==============================] - 3s 4ms/step - loss: 0.0095 - accuracy: 0.9978 - val\_loss: 0.0096 - val\_accuracy: 0.9977  
Epoch 6/100  
778/778 [==============================] - 3s 4ms/step - loss: 0.0074 - accuracy: 0.9984 - val\_loss: 0.0068 - val\_accuracy: 0.9986  
Epoch 7/100  
778/778 [==============================] - 3s 4ms/step - loss: 0.0060 - accuracy: 0.9987 - val\_loss: 0.0061 - val\_accuracy: 0.9986  
Epoch 8/100  
778/778 [==============================] - 3s 4ms/step - loss: 0.0050 - accuracy: 0.9989 - val\_loss: 0.0051 - val\_accuracy: 0.9989  
Epoch 9/100  
778/778 [==============================] - 3s 4ms/step - loss: 0.0042 - accuracy: 0.9991 - val\_loss: 0.0046 - val\_accuracy: 0.9989  
Epoch 10/100  
778/778 [==============================] - 3s 4ms/step - loss: 0.0037 - accuracy: 0.9992 - val\_loss: 0.0041 - val\_accuracy: 0.9990  
Epoch 11/100  
778/778 [==============================] - 3s 4ms/step - loss: 0.0033 - accuracy: 0.9993 - val\_loss: 0.0037 - val\_accuracy: 0.9993  
Epoch 12/100  
778/778 [==============================] - 3s 4ms/step - loss: 0.0029 - accuracy: 0.9994 - val\_loss: 0.0035 - val\_accuracy: 0.9992  
Epoch 13/100  
778/778 [==============================] - 3s 4ms/step - loss: 0.0027 - accuracy: 0.9994 - val\_loss: 0.0037 - val\_accuracy: 0.9991  
Epoch 14/100  
778/778 [==============================] - 3s 4ms/step - loss: 0.0024 - accuracy: 0.9995 - val\_loss: 0.0035 - val\_accuracy: 0.9990  
Epoch 15/100  
778/778 [==============================] - 3s 4ms/step - loss: 0.0022 - accuracy: 0.9995 - val\_loss: 0.0032 - val\_accuracy: 0.9992  
Epoch 16/100  
778/778 [==============================] - 3s 4ms/step - loss: 0.0020 - accuracy: 0.9996 - val\_loss: 0.0027 - val\_accuracy: 0.9994  
Epoch 17/100  
778/778 [==============================] - 3s 4ms/step - loss: 0.0020 - accuracy: 0.9995 - val\_loss: 0.0027 - val\_accuracy: 0.9993  
Epoch 18/100  
778/778 [==============================] - 3s 4ms/step - loss: 0.0018 - accuracy: 0.9996 - val\_loss: 0.0023 - val\_accuracy: 0.9996  
Epoch 19/100  
778/778 [==============================] - 3s 4ms/step - loss: 0.0016 - accuracy: 0.9996 - val\_loss: 0.0025 - val\_accuracy: 0.9994  
Epoch 20/100  
778/778 [==============================] - 3s 4ms/step - loss: 0.0015 - accuracy: 0.9997 - val\_loss: 0.0025 - val\_accuracy: 0.9993  
Epoch 21/100  
778/778 [==============================] - 3s 4ms/step - loss: 0.0015 - accuracy: 0.9996 - val\_loss: 0.0022 - val\_accuracy: 0.9994  
Epoch 22/100  
778/778 [==============================] - 3s 4ms/step - loss: 0.0014 - accuracy: 0.9997 - val\_loss: 0.0024 - val\_accuracy: 0.9994  
Epoch 23/100  
778/778 [==============================] - 3s 4ms/step - loss: 0.0013 - accuracy: 0.9997 - val\_loss: 0.0022 - val\_accuracy: 0.9996  
Epoch 24/100  
778/778 [==============================] - 3s 4ms/step - loss: 0.0012 - accuracy: 0.9997 - val\_loss: 0.0022 - val\_accuracy: 0.9995  
Epoch 25/100  
778/778 [==============================] - 3s 4ms/step - loss: 0.0014 - accuracy: 0.9997 - val\_loss: 0.0025 - val\_accuracy: 0.9993  
Epoch 26/100  
778/778 [==============================] - 3s 4ms/step - loss: 0.0011 - accuracy: 0.9997 - val\_loss: 0.0027 - val\_accuracy: 0.9993  
Epoch 27/100  
778/778 [==============================] - 3s 4ms/step - loss: 9.9902e-04 - accuracy: 0.9997 - val\_loss: 0.0023 - val\_accuracy: 0.9995  
Epoch 28/100  
778/778 [==============================] - 3s 4ms/step - loss: 9.9616e-04 - accuracy: 0.9997 - val\_loss: 0.0021 - val\_accuracy: 0.9995  
Epoch 29/100  
778/778 [==============================] - 3s 4ms/step - loss: 8.8531e-04 - accuracy: 0.9998 - val\_loss: 0.0020 - val\_accuracy: 0.9995  
Epoch 30/100  
778/778 [==============================] - 3s 4ms/step - loss: 0.0010 - accuracy: 0.9997 - val\_loss: 0.0018 - val\_accuracy: 0.9996  
Epoch 31/100  
778/778 [==============================] - 3s 4ms/step - loss: 8.5445e-04 - accuracy: 0.9998 - val\_loss: 0.0019 - val\_accuracy: 0.9995  
Epoch 32/100  
778/778 [==============================] - 3s 4ms/step - loss: 9.3442e-04 - accuracy: 0.9997 - val\_loss: 0.0016 - val\_accuracy: 0.9997  
Epoch 33/100  
778/778 [==============================] - 3s 4ms/step - loss: 7.1790e-04 - accuracy: 0.9998 - val\_loss: 0.0016 - val\_accuracy: 0.9996  
Epoch 34/100  
778/778 [==============================] - 3s 4ms/step - loss: 7.2516e-04 - accuracy: 0.9998 - val\_loss: 0.0016 - val\_accuracy: 0.9996  
Epoch 35/100  
778/778 [==============================] - 3s 4ms/step - loss: 9.1578e-04 - accuracy: 0.9997 - val\_loss: 0.0020 - val\_accuracy: 0.9994  
Epoch 36/100  
778/778 [==============================] - 3s 4ms/step - loss: 6.7762e-04 - accuracy: 0.9998 - val\_loss: 0.0015 - val\_accuracy: 0.9997  
Epoch 37/100  
778/778 [==============================] - 3s 4ms/step - loss: 6.5826e-04 - accuracy: 0.9998 - val\_loss: 0.0015 - val\_accuracy: 0.9996  
Epoch 38/100  
778/778 [==============================] - 3s 4ms/step - loss: 7.5355e-04 - accuracy: 0.9998 - val\_loss: 0.0016 - val\_accuracy: 0.9996  
Epoch 39/100  
778/778 [==============================] - 3s 4ms/step - loss: 7.1149e-04 - accuracy: 0.9998 - val\_loss: 0.0015 - val\_accuracy: 0.9996  
Epoch 40/100  
778/778 [==============================] - 3s 4ms/step - loss: 5.8838e-04 - accuracy: 0.9999 - val\_loss: 0.0024 - val\_accuracy: 0.9994  
Epoch 41/100  
778/778 [==============================] - 3s 4ms/step - loss: 5.2743e-04 - accuracy: 0.9999 - val\_loss: 0.0015 - val\_accuracy: 0.9997  
Epoch 42/100  
778/778 [==============================] - 3s 4ms/step - loss: 7.9482e-04 - accuracy: 0.9998 - val\_loss: 0.0019 - val\_accuracy: 0.9995  
Epoch 43/100  
778/778 [==============================] - 3s 4ms/step - loss: 6.0553e-04 - accuracy: 0.9998 - val\_loss: 0.0013 - val\_accuracy: 0.9997  
Epoch 44/100  
778/778 [==============================] - 3s 4ms/step - loss: 6.0704e-04 - accuracy: 0.9998 - val\_loss: 0.0020 - val\_accuracy: 0.9995  
Epoch 45/100  
778/778 [==============================] - 3s 4ms/step - loss: 6.3886e-04 - accuracy: 0.9998 - val\_loss: 0.0015 - val\_accuracy: 0.9996  
Epoch 46/100  
778/778 [==============================] - 3s 4ms/step - loss: 5.7343e-04 - accuracy: 0.9998 - val\_loss: 0.0013 - val\_accuracy: 0.9997  
Epoch 47/100  
778/778 [==============================] - 3s 4ms/step - loss: 4.2137e-04 - accuracy: 0.9999 - val\_loss: 0.0018 - val\_accuracy: 0.9996  
Epoch 48/100  
778/778 [==============================] - 3s 4ms/step - loss: 6.0087e-04 - accuracy: 0.9998 - val\_loss: 0.0016 - val\_accuracy: 0.9997  
Epoch 49/100  
778/778 [==============================] - 3s 4ms/step - loss: 5.2967e-04 - accuracy: 0.9999 - val\_loss: 0.0012 - val\_accuracy: 0.9998  
Epoch 50/100  
778/778 [==============================] - 3s 4ms/step - loss: 5.2438e-04 - accuracy: 0.9998 - val\_loss: 0.0012 - val\_accuracy: 0.9997  
Epoch 51/100  
778/778 [==============================] - 3s 4ms/step - loss: 5.0218e-04 - accuracy: 0.9998 - val\_loss: 0.0018 - val\_accuracy: 0.9996  
Epoch 52/100  
778/778 [==============================] - 3s 4ms/step - loss: 6.3864e-04 - accuracy: 0.9998 - val\_loss: 0.0020 - val\_accuracy: 0.9995  
Epoch 53/100  
778/778 [==============================] - 3s 4ms/step - loss: 6.7778e-04 - accuracy: 0.9998 - val\_loss: 0.0016 - val\_accuracy: 0.9996  
Epoch 54/100  
778/778 [==============================] - 3s 4ms/step - loss: 4.3683e-04 - accuracy: 0.9999 - val\_loss: 0.0017 - val\_accuracy: 0.9996  
Epoch 55/100  
778/778 [==============================] - 3s 4ms/step - loss: 7.2916e-04 - accuracy: 0.9998 - val\_loss: 0.0018 - val\_accuracy: 0.9996  
Epoch 56/100  
778/778 [==============================] - 3s 4ms/step - loss: 4.6164e-04 - accuracy: 0.9999 - val\_loss: 0.0017 - val\_accuracy: 0.9996  
Epoch 57/100  
778/778 [==============================] - 3s 4ms/step - loss: 4.3985e-04 - accuracy: 0.9999 - val\_loss: 0.0022 - val\_accuracy: 0.9995  
Epoch 58/100  
778/778 [==============================] - 3s 4ms/step - loss: 4.4691e-04 - accuracy: 0.9999 - val\_loss: 0.0015 - val\_accuracy: 0.9996  
Epoch 59/100  
778/778 [==============================] - 3s 4ms/step - loss: 6.0507e-04 - accuracy: 0.9998 - val\_loss: 0.0010 - val\_accuracy: 0.9998  
Epoch 60/100  
778/778 [==============================] - 3s 4ms/step - loss: 4.2673e-04 - accuracy: 0.9999 - val\_loss: 0.0027 - val\_accuracy: 0.9995  
Epoch 61/100  
778/778 [==============================] - 3s 4ms/step - loss: 3.7599e-04 - accuracy: 0.9999 - val\_loss: 0.0019 - val\_accuracy: 0.9996  
Epoch 62/100  
778/778 [==============================] - 3s 4ms/step - loss: 6.4678e-04 - accuracy: 0.9998 - val\_loss: 0.0024 - val\_accuracy: 0.9995  
Epoch 63/100  
778/778 [==============================] - 3s 4ms/step - loss: 3.3452e-04 - accuracy: 0.9999 - val\_loss: 0.0013 - val\_accuracy: 0.9997  
Epoch 64/100  
778/778 [==============================] - 3s 4ms/step - loss: 4.5338e-04 - accuracy: 0.9999 - val\_loss: 0.0022 - val\_accuracy: 0.9995  
Epoch 65/100  
778/778 [==============================] - 3s 4ms/step - loss: 5.2037e-04 - accuracy: 0.9998 - val\_loss: 0.0013 - val\_accuracy: 0.9997  
Epoch 66/100  
778/778 [==============================] - 3s 4ms/step - loss: 3.9358e-04 - accuracy: 0.9999 - val\_loss: 0.0014 - val\_accuracy: 0.9997  
Epoch 67/100  
778/778 [==============================] - 3s 4ms/step - loss: 4.2975e-04 - accuracy: 0.9999 - val\_loss: 0.0018 - val\_accuracy: 0.9997  
Epoch 68/100  
778/778 [==============================] - 3s 4ms/step - loss: 3.7875e-04 - accuracy: 0.9999 - val\_loss: 0.0012 - val\_accuracy: 0.9997  
Epoch 69/100  
778/778 [==============================] - 3s 4ms/step - loss: 5.4449e-04 - accuracy: 0.9998 - val\_loss: 0.0017 - val\_accuracy: 0.9997

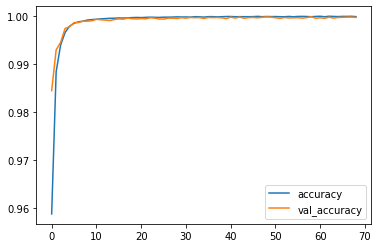
results = model.evaluate(x\_test, y\_test, batch\_size=5, verbose=1)  
print("Loss: %.2f" % results[0])  
print("Acc: %.2f" % results[1])

34118/34118 [==============================] - 74s 2ms/step - loss: 0.0010 - accuracy: 0.9998  
Loss: 0.00  
Acc: 1.00

print(r.history.keys())  
plt.plot(r.history['loss'])  
plt.plot(r.history['val\_loss'])  
plt.legend(['loss', 'val\_loss'])  
plt.show()  
  
plt.plot(r.history['accuracy'])  
plt.plot(r.history['val\_accuracy'])  
plt.legend(['accuracy', 'val\_accuracy'])  
plt.show()

dict\_keys(['loss', 'accuracy', 'val\_loss', 'val\_accuracy'])





y\_pred = model.predict(x\_test)  
y\_pred = np.round(y\_pred, decimals=0).astype(int)

df\_pred = pd.concat([pd.DataFrame(x\_test), pd.DataFrame(y\_test)], axis=1)  
df\_pred.columns = df.drop('Time', axis=1).columns  
df\_pred.rename(columns={"Class":"Old\_class"}, inplace=True)  
df\_pred['New\_class'] = y\_pred  
cm = pd.crosstab(df\_pred["New\_class"], df\_pred['Old\_class'])  
true\_pos = np.sum(np.diag(cm))  
false\_pos = cm[0][1]  
false\_neg = cm[1][0]  
precision = true\_pos / (true\_pos + false\_pos) \* 100  
recall = true\_pos / (true\_pos + false\_neg) \* 100  
f1 = 2 \* (precision \* recall) / (precision + recall)  
print("Precision: %.3f%%" % (precision))  
print("Recall: %.3f%%" % (recall))  
print("F1: %.3f%%" % (f1))

Precision: 99.981%  
Recall: 100.000%  
F1: 99.991%

### Conclusion

* The K-Nearest Neighbors Classifier tuned with Grid Search with the best parameter being the Euclidean Distance (p=2) outperforms its counterparts to give a test accuracy of nearly 99.8% and a perfect F1-Score with minimal overfitting
* SMOTE overcomes overfitting by synthetically oversampling minority class labels and is successful to a great degree

### Summary

* All Fraud Transactions occur for an amount below 2500. Thus, the bank can infer clearly that the fraud committers try to commit frauds of smaller amounts to avoid suspicion.
* The fraud transactions are equitable distributed throughout time and there is no clear relationship of time with committing of fraud.