

CAPSTONE PROJECT – Credit Card Fraud

Hari Santosh

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

Mounted at /content/drive

Mounting the google drive where the dataset is present.

```
import numpy as np
import pandas as pd

pd.set_option('display.max_columns', None)

[ ] df_original = pd.read_csv("/content/drive/MyDrive/creditcard.csv")
df_original
```

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	V13	V14	V15
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	0.090794	-0.551600	-0.617801	-0.991390	-0.311169	1.468177
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	-0.166974	1.612727	1.065235	0.489095	-0.143772	0.635558

Importing numpy and pandas.

We use `pd.set_option(display.max_columns)` parameter to display all the rows which would otherwise be hidden from us.

```
[ ] # Count the number of duplicate rows
num_duplicates = duplicate_rows.sum()

# Print the count of duplicate rows
print("Number of Duplicate Rows:", num_duplicates)

Number of Duplicate Rows: 1081

[ ] fraud = df_original[df_original['Class'] == 1 ]
fraud.describe()
```

	Amount	Class
count	492.000000	492.0
mean	122.211321	1.0
std	256.683288	0.0
min	0.000000	1.0
25%	1.000000	1.0
50%	9.250000	1.0
75%	105.890000	1.0
max	2125.870000	1.0

We are checking for Duplicate rows in the above code, and also describing the minority class (Fraud). It is a minority class because there's only 492 rows out of 284807 rows. The above table is the descriptive statistics of the "Fraud" Class with respect to "Amount" Feature.

```
[ ] print(df_original.duplicated().sum())

1081


▶ df_original.drop_duplicates(inplace=True)
print(df_original.duplicated().sum())


➞ 0

[ ] np.mean(df_original['Amount'])

88.47268731099724
```

We check the mean of the Feature "Amount" and also drop duplicates if any.

 `df_original.isna().sum()`

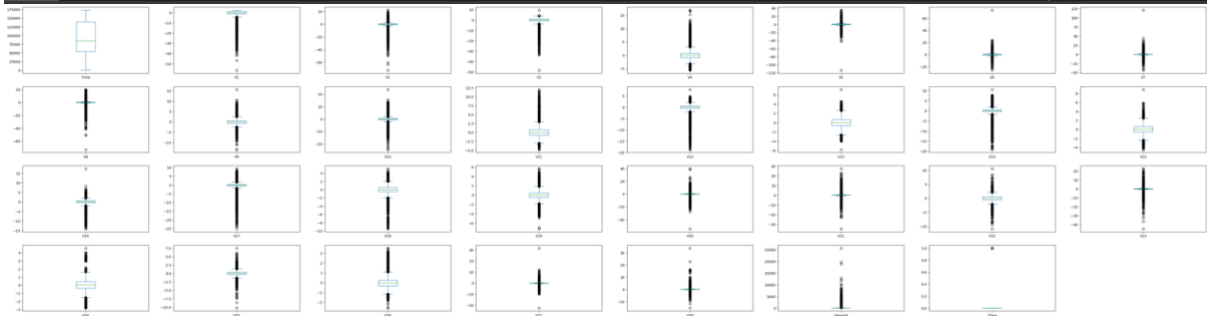


Time	0
V1	0
V2	0
V3	0
V4	0
V5	0
V6	0
V7	0
V8	0
V9	0
V10	0
V11	0
V12	0
V13	0
V14	0
V15	0
V16	0
V17	0
V18	0
V19	0
V20	0
V21	0
V22	0
V23	0
V24	0
V25	0
V26	0
V27	0
V28	0
Amount	0
Class	0
dtype:	int64

As we can see there are no null values in our Dataset

```
[ ] import matplotlib.pyplot as plt
import seaborn as sns
```

```
df_original.plot(kind = "box" , subplots = True , figsize = (60,60) , layout = (15,8))
```



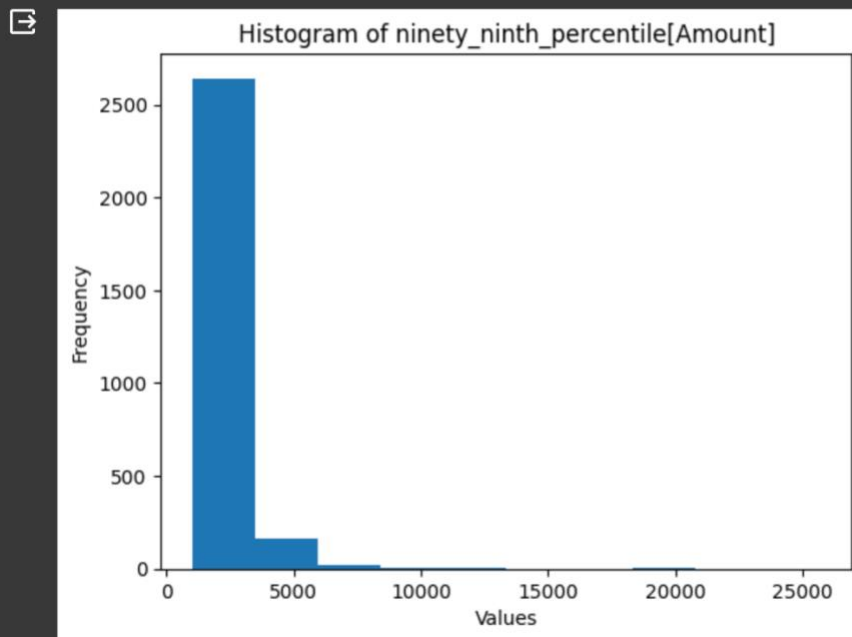
Above code to generate boxplots for each feature in our dataset using seaborn.

▼ OUTLIER TREATMENT FOR 'AMOUNT' FEATURE

```
# Calculate the 1st percentile
first_percentile = df_original[df_original['Amount'] <= df_original['Amount'].quantile(0.01)]
|
# Calculate the 99th percentile
ninety_ninth_percentile = df_original[df_original['Amount'] >= df_original['Amount'].quantile(0.99)]
```

Here we are checking the 1st and 99th percentile of the dataset based on “Amount” feature. We are doing this because this is the threshold value chosen as the limit for outliers. Anything Below or Above respectively, can be treated as outliers. But to remove, cap or leave them untreated is up to us.

```
plt.hist(ninety_ninth_percentile['Amount'])
plt.xlabel('Values')
plt.ylabel('Frequency')
plt.title('Histogram of ninety_ninth_percentile[Amount]')
plt.show()
```



Generating a histogram plot to check the distribution of “Amount” Feature in the 99th percentile range only. It is Heavily skewed to the left. Around 3000 rows of outliers.

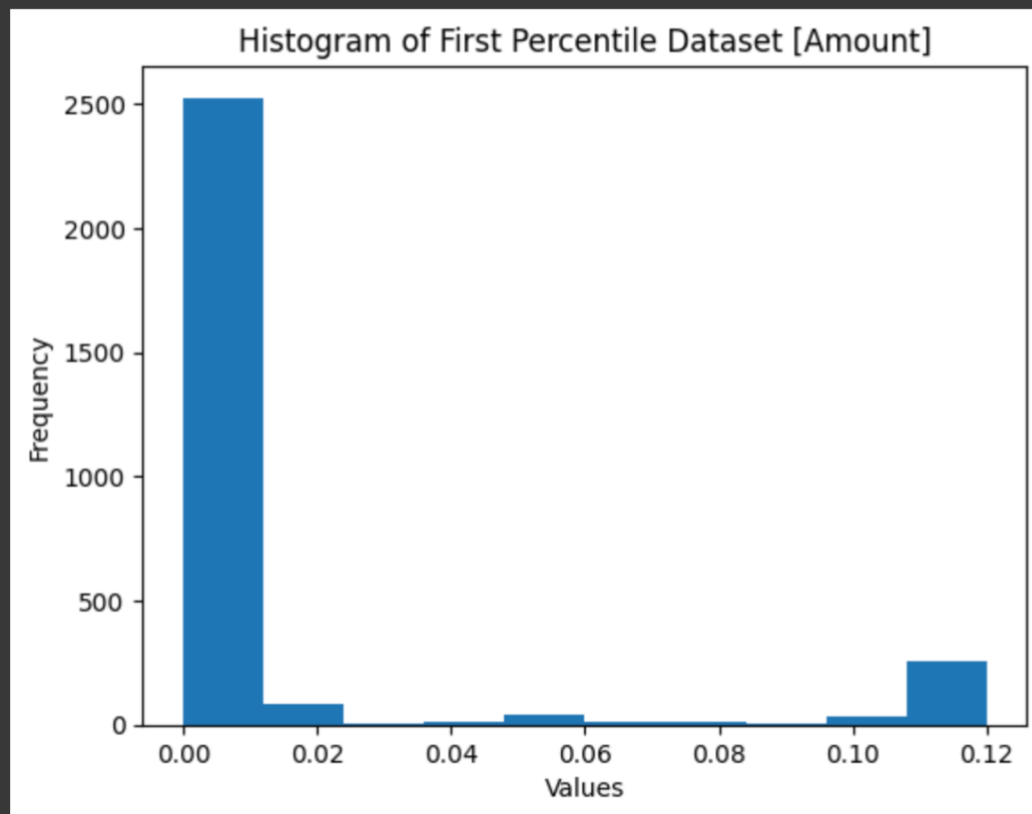
```
a = ninety_ninth_percentile[ninety_ninth_percentile['Amount'] > 2000]
a[a['Class'] == 1]
```

	Time	V1	V2	V3	V4	V5	V6	V7
176049	122608.0	-2.00346	-7.159042	-4.050976	1.30958	-2.058102	-0.098621	2.880083

```
[ ] np.median(ninety_ninth_percentile['Amount'])
1457.9499999999998
```

Checking fraud transactions in 99th percentile, but only 1 row. This may seem like we can drop 99th percentile data, but the median is 1457, while the max fraud transaction value is 2125.

```
plt.hist(first_percentile['Amount'])
plt.xlabel('Values')
plt.ylabel('Frequency')
plt.title('Histogram of First Percentile Dataset [Amount]')
plt.show()
```



This histogram is to check the Amount feature distribution in the 1st percentile data.

```
from scipy.stats import mstats

# Specify the column you want to winsorize
column_to_winsorize = 'Amount'

# Winsorize the column at the 1st percentile only
winsorized_data = mstats.winsorize(df_original[column_to_winsorize], limits=[0.01, 0.00])

# Assign the winsorized values back to the original DataFrame
df_original[column_to_winsorize] = winsorized_data
```

Here I've not capped outliers at the >99 percentile quantile because the mean of df_original drops from ~88 to 80 after capping. It is a 10% effect, therefore only the bottom 1 percentile data is capped, barely affecting the mean of df_original. The extreme values in the >99 percentile quantile feel like genuine extreme values, probably not a data entry mistake

Another reason to not drop 99th percentile is given above, and we are capping (winsorizing) the 1st percentile data only.

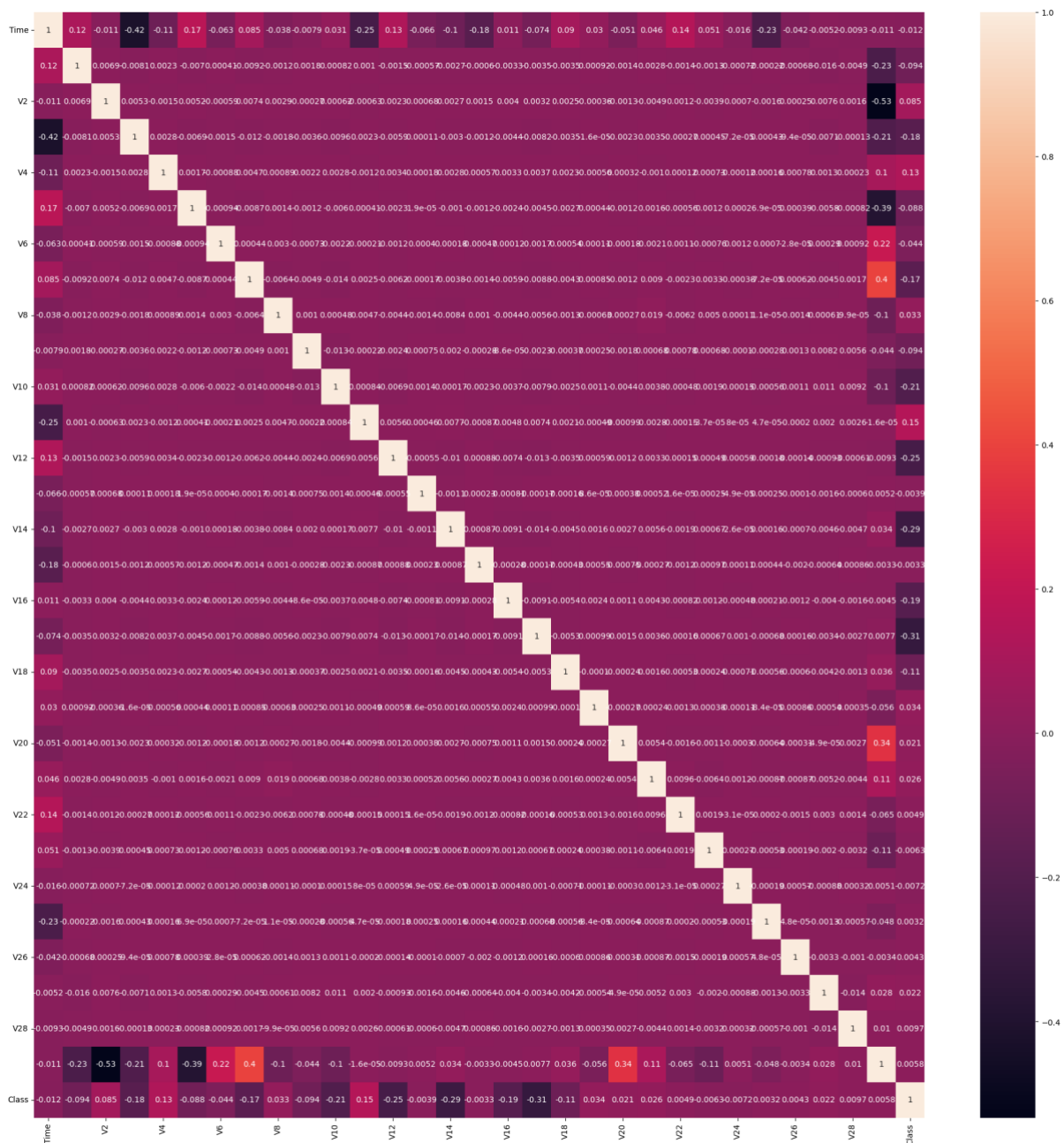
```
# Calculate the correlation matrix
correlation_matrix = df_original.corr()

# Print the correlation matrix
correlation_matrix
```

	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	V29	V30	V31
Time	1.000000	0.117927	-0.010556	-0.422054	-0.105845	0.173223	-0.063279	0.085335	-0.038203	-0.007861	0.031068	-0.248536	0.125500	-0.065958	-0.100010	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000
V1	0.117927	1.000000	0.006875	-0.008112	0.002257	-0.007036	0.000413	-0.009173	-0.001168	0.001828	0.000815	0.001028	-0.001524	-0.000568	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000
V2	-0.010556	0.006875	1.000000	0.005278	-0.001495	0.005210	-0.000594	0.007425	0.002899	-0.000274	0.000620	-0.000633	0.002266	0.000680	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000
V3	-0.422054	-0.008112	0.005278	1.000000	0.002829	-0.006879	-0.001511	-0.011721	-0.001815	-0.003579	-0.009632	0.002339	-0.005900	0.000113	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000
V4	-0.105845	0.002257	-0.001495	0.002829	1.000000	0.001744	-0.000880	0.004657	0.000890	0.002154	0.002753	-0.001223	0.003366	0.000177	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000
V5	0.173223	-0.007036	0.005210	-0.006879	0.001744	1.000000	-0.000938	-0.008709	0.001430	-0.001213	-0.006050	0.000411	-0.002342	0.000019	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000
V6	-0.063279	0.000413	-0.000594	-0.001511	-0.000880	-0.000938	1.000000	0.000436	0.003036	-0.000734	-0.002180	-0.000211	-0.001185	0.000397	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000
V7	0.085335	-0.009173	0.007425	-0.011721	0.004657	-0.008709	0.000436	1.000000	-0.006419	-0.004921	-0.013617	0.002454	-0.006153	-0.000170	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000
V8	-0.038203	-0.001168	0.002899	-0.001815	0.000890	0.001430	0.003036	-0.006419	1.000000	0.001038	0.000481	0.004688	-0.004414	-0.001381	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000
V9	-0.007861	0.001828	-0.000274	-0.003579	0.002154	-0.001213	-0.000734	-0.004921	0.001038	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
V10	0.031068	0.000815	0.000620	-0.009632	0.002753	-0.006050	-0.002180	-0.013617	0.000481	0.000000	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
V11	-0.248536	0.001028	-0.000633	0.002339	-0.001223	0.000411	-0.000211	0.002454	0.004688	0.000000	0.000000	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
V12	0.125500	-0.001524	0.002266	-0.005900	0.003366	-0.002342	-0.001185	-0.006153	-0.004414	0.000000	0.000000	0.000000	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
V13	-0.065958	-0.000568	0.000680	0.000113	0.000177	0.000019	0.000397	-0.000170	-0.001381	0.000000	0.000000	0.000000	0.000000	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000

Making the correlation matrix for all 31 features. But since the dataset has already undergone PCA transformation, the maximum variance has been captured for each feature and values have been transformed, therefore no features are highly correlated. We will see this in the heatmap below.

```
# Visualize the correlation matrix using a heatmap:
ax = sns.heatmap(correlation_matrix, annot=True)
plt.gcf().set_size_inches(25, 25)
```



The heatmap generated from the above correlation matrix. We can see the features are barely correlated with each other.

Handling very high class imbalance

```
# Select the column you want to create a pie chart for
feature_column = 'Class'

# Calculate the value counts and percentages
value_counts = df_original[feature_column].value_counts()
percentages = value_counts / len(df_original) * 100

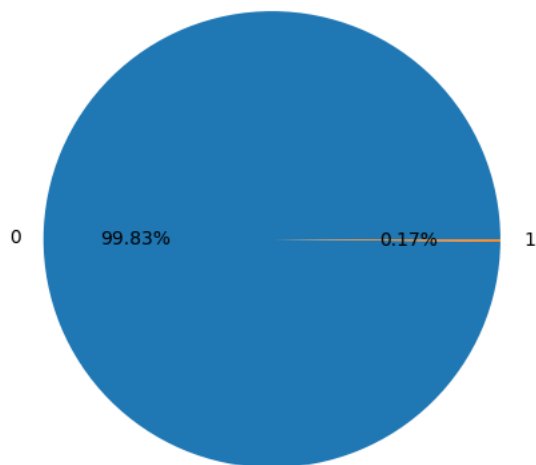
# Add a title to the chart
plt.title("Pie Chart of {}".format(feature_column))

# Add the percentages as text outside each slice
_, _ = plt.pie(value_counts, labels=value_counts.index, autopct='%0.2f%%')
for autotext in autotexts:
    autotext.set_horizontalalignment('center')
    autotext.set_verticalalignment('center')

# Display the chart
plt.tight_layout()

# 1 Means FRAUD
# 0 means LEGIT
```

Pie Chart of Class



Using a pie chart we can visualize the class distribution difference between majority and minority, 99.83% vs 0.17%. Extremely imbalanced. We will need to handle this imbalance before making any ML model.

```
from imblearn.under_sampling import TomekLinks

# Separate the features (X) and the target variable (Y)
X = df_original.drop('Class', axis=1)
Y = df_original['Class']

# Undersample the dataset using Tomek links
tl = TomekLinks()
X_resampled, Y_resampled = tl.fit_resample(X, Y)

# Print the shapes of the resampled datasets
print("Resampled X shape:", X_resampled.shape)
print("Resampled Y shape:", Y_resampled.shape)
```

⇒ Resampled X shape: (283653, 30)
Resampled Y shape: (283653,)

As we could see from the above code, Tomek Links had increased our minority class from 473 samples to 283653, which was too much considering the size of the dataset which could impact the model training time. The `fit_resample` method itself took 6+ hours to run, hence code was commented out the from running again.

The attempt to resample only the majority class and not the minority; did not work as the samples in X and Y remained at 283653, resulting in excessive processing time. In the next cell, an alternative approach was to manually specify the value of the minority class and sample accordingly by mapping it to the majority class. If this method failed, the plan was to proceed with random under sampling.

```

from imblearn.under_sampling import TomekLinks

# Separate the features (X) and the target variable (Y)
X = df_original.drop('Class', axis=1)
Y = df_original['Class']

# Undersample the dataset using Tomek links with specified class ratios
tl = TomekLinks(sampling_strategy='auto', n_jobs = -1)
X_resampled, Y_resampled = tl.fit_resample(X, Y)

# Print the shapes of the resampled datasets
print("Resampled X shape:", X_resampled.shape)
print("Resampled Y shape:", Y_resampled.shape)

```

```

Resampled X shape: (283653, 30)
Resampled Y shape: (283653,)

```

```

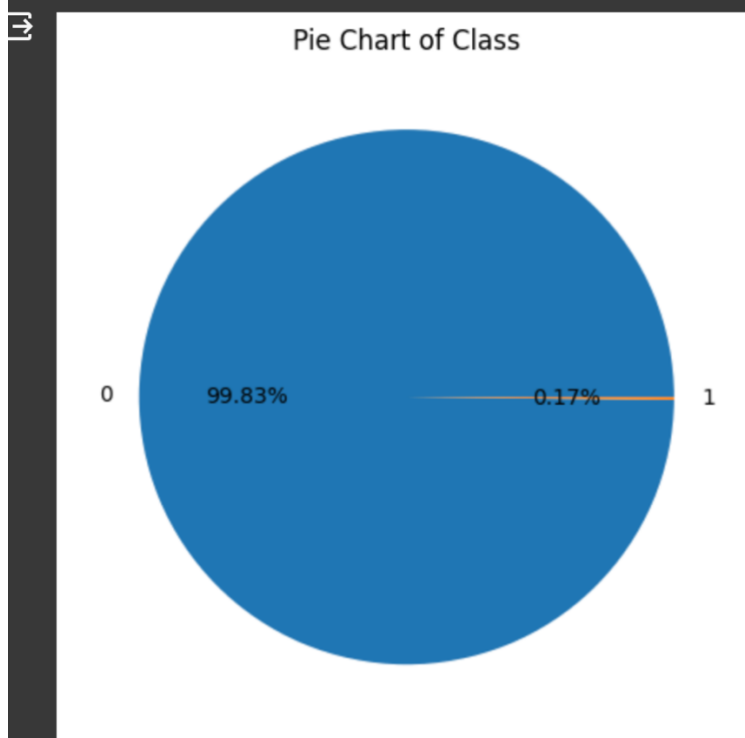
[ ] Y_resampled[Y_resampled == 1].count()

```

```

473

```



```

] percentages

```

```

0    99.833247
1     0.166753
Name: Class, dtype: float64

```

The distribution remained the same. Therefore, another undersampling method had to be tried.

```

from imblearn.under_sampling import RandomUnderSampler

# Assuming your dataset is stored in a pandas DataFrame called 'df_original'
# 'feature' is the column name based on which you want to balance the classes

# Separate the feature column and the target column
X = df_original.drop('Class', axis=1)
y = df_original['Class']

# Create an instance of RandomUnderSampler
rus = RandomUnderSampler(random_state=42)

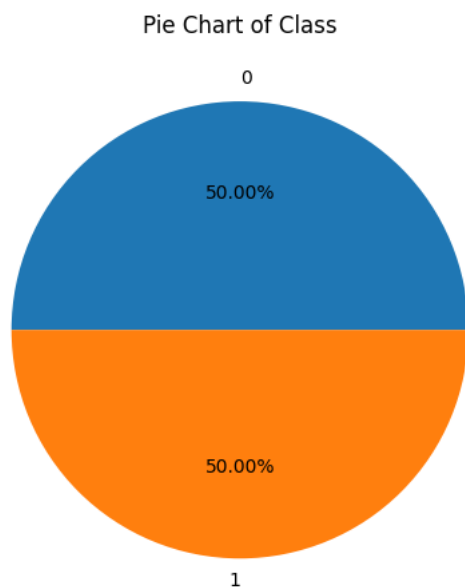
# Perform random under-sampling
X_resampled, y_resampled = rus.fit_resample(X, y)

print(X_resampled.shape)
print(y_resampled.shape )

```

(946, 30)
(946,)

The total number of rows reduced from ~284000 to 946. Below is the distribution of the new sample generated by Using RandomUnderSampler.



We can see there are exactly half Fraud and half legit transactions, totaling to 946. This is extremely favorable with respect to model training, but one problem would be that we would be losing out on most of the data and its hidden patterns. But we can use it here because the data is extremely imbalanced so we need to give more priority to guess the Fraud transactions compared to the Legit ones.

performing scaling using standard Scaler on Amount and time column after Train Test split

```
from sklearn.model_selection import train_test_split

# Split the data into training and test sets
X_train, X_test, Y_train, Y_test = train_test_split(X_resampled, y_resampled, test_size=0.2, random_state=42, shuffle = True)

# Print the shapes of the resulting datasets
print("Train X shape:", X_train.shape)
print("Train Y shape:", Y_train.shape)
print("Test X shape:", X_test.shape)
print("Test Y shape:", Y_test.shape)
```

Train X shape: (756, 30)
Train Y shape: (756,)
Test X shape: (190, 30)
Test Y shape: (190,)

We will be applying train test split function from sklearn to split our resampled dataset to train data and test data.

```
[ ] from sklearn.preprocessing import StandardScaler

# Create a StandardScaler object
scaler = StandardScaler()

# Fit the scaler on the training set and transform the "Amount" feature
X_train["Amount"] = scaler.fit_transform(X_train["Amount"].values.reshape(-1, 1))

# Transform the "Amount" feature in the test set
X_test["Amount"] = scaler.transform(X_test["Amount"].values.reshape(-1, 1))
```

We will now apply standard scaler to 1 Feature only "Amount" and use this scaled data to feed into the model. Fit_transform is applied on the train dataset and the transform method is used to transform the "Amount" feature in the test set. This ensures that the test set is scaled using the mean and standard deviation learned from the training set.

If we use the whole X_resampled dataset for standard scaler without first splitting, this will cause Data Leakage into the model causing bias and overfit because information from test set gets carried over to train set.

```

from sklearn.metrics import f1_score, accuracy_score
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
import xgboost as xgb

# Create a list of models
models = [
    LogisticRegression(),
    SVC(),
    KNeighborsClassifier(),
    GaussianNB(),
    DecisionTreeClassifier(),
    RandomForestClassifier(),
    xgb.XGBClassifier()
]

# Train and test each model
for model in models:
    # Train the model
    model.fit(X_train, Y_train)

    # Test the model
    Y_pred = model.predict(X_test)

    # Calculate the F1 score and accuracy
    f1 = f1_score(Y_test, Y_pred)
    accuracy = accuracy_score(Y_test, Y_pred)

    # Print the model name, F1 score, and accuracy
    print(f"Model: {type(model).__name__}")
    print(f"F1 Score: {f1}")
    print(f"Accuracy: {accuracy}")

```

Running a couple of sample models initially on this train and test set where only “amount” is scaled. The scores are as follows:

Model: LogisticRegression
 F1 Score: 0.9128205128205128
 Accuracy: 0.9105263157894737

Model: SVC
 F1 Score: 0.6020408163265305
 Accuracy: 0.5894736842105263

Model: KNeighborsClassifier
 F1 Score: 0.6850828729281768
 Accuracy: 0.7

Model: GaussianNB
F1 Score: 0.8743169398907104
Accuracy: 0.8789473684210526

Model: DecisionTreeClassifier
F1 Score: 0.9108910891089109
Accuracy: 0.9052631578947369

Model: RandomForestClassifier
F1 Score: 0.9381443298969072
Accuracy: 0.9368421052631579

Model: XGBClassifier
F1 Score: 0.9387755102040817
Accuracy: 0.9368421052631579

As we can see from the above code, XGBoost algorithm gives us the highest accuracy and f1 score. We will now try to standardize the "Time" feature as well and perform model training again to see if there would be any difference between the 2 (2 here refers to with standardizing "time" feature and without standardizing).

```
# Create a StandardScaler object
scaler = StandardScaler()

# Fit the scaler on the training set and transform the "Amount" feature
X_train["Amount"] = scaler.fit_transform(X_train["Amount"].values.reshape(-1, 1))

# Transform the "Amount" feature in the test set
X_test["Amount"] = scaler.transform(X_test["Amount"].values.reshape(-1, 1))

# Fit the scaler on the training set and transform the "Time" feature
X_train["Time"] = scaler.fit_transform(X_train["Time"].values.reshape(-1, 1))

# Transform the "Time" feature in the test set
X_test["Time"] = scaler.transform(X_test["Time"].values.reshape(-1, 1))
```

Both features have now been scaled, we will now retrain all models.

```

from sklearn.metrics import f1_score, accuracy_score
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
import xgboost as xgb

# Create a list of models
models = [
    LogisticRegression(),
    SVC(),
    KNeighborsClassifier(),
    GaussianNB(),
    DecisionTreeClassifier(),
    RandomForestClassifier(),
    xgb.XGBClassifier()
]

# Train and test each model
for model in models:
    # Train the model
    model.fit(X_train, Y_train)

    # Test the model
    Y_pred = model.predict(X_test)

    # Calculate the F1 score and accuracy
    f1 = f1_score(Y_test, Y_pred)
    accuracy = accuracy_score(Y_test, Y_pred)

    # Print the model name, F1 score, and accuracy
    print(f"Model: {type(model).__name__}")
    print(f"F1 Score: {f1}")
    print(f"Accuracy: {accuracy}")

```

Model: LogisticRegression
 F1 Score: 0.9346733668341709
 Accuracy: 0.9315789473684211

Model: SVC
 F1 Score: 0.9333333333333333
 Accuracy: 0.9315789473684211

Model: KNeighborsClassifier
 F1 Score: 0.9435897435897437
 Accuracy: 0.9421052631578948

Model: GaussianNB
 F1 Score: 0.9109947643979056
 Accuracy: 0.9105263157894737

Model: DecisionTreeClassifier
F1 Score: 0.9054726368159205
Accuracy: 0.9

Model: RandomForestClassifier
F1 Score: 0.9489795918367346
Accuracy: 0.9473684210526315

Model: XGBClassifier
F1 Score: 0.9387755102040817
Accuracy: 0.9368421052631579

We can see that RandomForestClassifier now performs the best. But we also notice all classifiers have above 0.9 F1 score now after applying StandardScaler on the "Time" feature and "Amount" Feature together.

We will now be performing GridSearchCV and make a param_grid and perform hyperparameter tuning for each model along with different scaling techniques like-

Min-Max Scaler

Quantile Transformer

StandardScaler

The param_grid object will have different hyperparameters stored for each Machine Learning Model type and their corresponding value. We iterate through all the combination of hyperparameters possible for each model and choose the best model for each model type.

We store the best_model for each model type along with the scaling technique used in results.

```

from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import MinMaxScaler, QuantileTransformer, StandardScaler
from sklearn.metrics import f1_score, accuracy_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
import xgboost as xgb

# Split the data into training and test sets
X_train, X_test, Y_train, Y_test = train_test_split(X_resampled, y_resampled, test_size=0.2, random_state=42, shuffle=True)

# Create a list of scaling techniques
scalers = [
    StandardScaler(),
    MinMaxScaler(),
    QuantileTransformer()
]

# Create a list of models
models = [
    LogisticRegression(),
    SVC(),
    KNeighborsClassifier(),
    DecisionTreeClassifier(),
    RandomForestClassifier(),
    xgb.XGBClassifier()
]

```

Initializing the code.

```

# Create an empty dictionary to store the results
results = {}

# Train and test each model with each scaling technique
for scaler in scalers:
    # Create a separate dictionary for each scaling technique
    scaler_results = {}

    # Fit and transform the "Amount" feature in the training set
    X_train_scaled = X_train.copy()
    X_train_scaled["Amount"] = scaler.fit_transform(X_train_scaled["Amount"].values.reshape(-1, 1))

    # Transform the "Amount" feature in the test set
    X_test_scaled = X_test.copy()
    X_test_scaled["Amount"] = scaler.transform(X_test_scaled["Amount"].values.reshape(-1, 1))

    # Fit and transform the "Time" feature in the training set
    X_train_scaled["Time"] = scaler.fit_transform(X_train_scaled["Time"].values.reshape(-1, 1))

    # Transform the "Time" feature in the test set
    X_test_scaled["Time"] = scaler.transform(X_test_scaled["Time"].values.reshape(-1, 1))

    ID = 1
    for model in models:
        # Define the hyperparameter grid for the model
        param_grid = {}

```

First loop for going through each scaling technique

```

for model in models:
    # Define the hyperparameter grid for the model
    param_grid = {}

    # Define the hyperparameter grid for the model
    if isinstance(model, LogisticRegression):
        param_grid = {'C': [0.1, 1, 10], 'penalty': ['l1', 'l2']}
    elif isinstance(model, SVC):
        param_grid = {'C': [0.1, 1, 10], 'kernel': ['linear', 'rbf'], 'gamma': ['scale', 'auto']}
    elif isinstance(model, KNeighborsClassifier):
        param_grid = {'n_neighbors': [3, 5, 7], 'weights': ['uniform', 'distance']}
    elif isinstance(model, DecisionTreeClassifier):
        param_grid = {'max_depth': [5, 10], 'criterion': ['gini', 'entropy']}
    elif isinstance(model, RandomForestClassifier):
        param_grid = {'n_estimators': [100, 200, 300], 'max_depth': [5, 10]}
    elif isinstance(model, xgb.XGBClassifier):
        param_grid = {'learning_rate': [0.1, 0.01, 0.001], 'max_depth': [3, 6], 'n_estimators': [100, 200, 300]}

    # Perform GridSearchCV
    grid_search = GridSearchCV(estimator=model, param_grid=param_grid, scoring='f1', cv=5)
    grid_search.fit(X_train_scaled, Y_train)
    best_model = grid_search.best_estimator_
    Y_pred = best_model.predict(X_test_scaled)

    # Calculate the F1 score and accuracy
    f1 = f1_score(Y_test, Y_pred)
    accuracy = accuracy_score(Y_test, Y_pred)

    # Store the results in the scaler_results dictionary
    scaler_results[ID] = {
        "f1": f1,
        "accuracy": accuracy,
        "scaler": scaler,
        "best_params": grid_search.best_params_,
        "model_": best_model
    }
}

```

Inner loop for applying each scaling technique on each model and at the same time, performing GridsearchCV to find the best model.

```

    }

    ID += 1

    # Print the model name, F1 score, accuracy, scaling technique, and best params
    print(f"Model: {type(best_model).__name__}")
    print(f"F1 Score: {f1}")
    print(f"Accuracy: {accuracy}")
    print(f"Scaling Technique: {type(scaler).__name__}")
    print(f"Best Params: {grid_search.best_params_}")
    print("\n")

    # Store the scaler_results dictionary in the results dictionary
    results[type(scaler).__name__] = scaler_results

```

Outside inner loop, to store the best model and its attributes.

```
Model: LogisticRegression
F1 Score: 0.9393939393939394
Accuracy: 0.9368421052631579
Scaling Technique: QuantileTransformer
Best Params: {'C': 10, 'penalty': 'l2'}

Model: SVC
F1 Score: 0.9393939393939394
Accuracy: 0.9368421052631579
Scaling Technique: QuantileTransformer
Best Params: {'C': 1, 'gamma': 'scale', 'kernel': 'linear'}

Model: KNeighborsClassifier
F1 Score: 0.9270833333333334
Accuracy: 0.9263157894736842
Scaling Technique: QuantileTransformer
Best Params: {'n_neighbors': 7, 'weights': 'distance'}

Model: DecisionTreeClassifier
F1 Score: 0.9119170984455958
Accuracy: 0.9105263157894737
Scaling Technique: QuantileTransformer
Best Params: {'criterion': 'gini', 'max_depth': 5}

Model: RandomForestClassifier
F1 Score: 0.9387755102040817
Accuracy: 0.9368421052631579
Scaling Technique: QuantileTransformer
Best Params: {'max_depth': 10, 'n_estimators': 100}
```

Example output.

```
results

{ 'StandardScaler': {1: {'f1': 0.9346733668341709,
  'accuracy': 0.9315789473684211,
  'scaler': StandardScaler(),
  'best_params': {'C': 10, 'penalty': 'l2'},
  'model_': LogisticRegression(C=10)},
  2: {'f1': 0.934010152284264,
  'accuracy': 0.9315789473684211,
  'scaler': StandardScaler(),
  'best_params': {'C': 10, 'gamma': 'scale', 'kernel': 'rbf'},
  'model_': SVC(C=10)},
  3: {'f1': 0.9381443298969072,
  'accuracy': 0.9368421052631579,
  'scaler': StandardScaler(),
  'best_params': {'n_neighbors': 7, 'weights': 'distance'},
  'model_': KNeighborsClassifier(n_neighbors=7, weights='distance')},
  4: {'f1': 0.9128205128205128,
  'accuracy': 0.9105263157894737,
  'scaler': StandardScaler(),
  'best_params': {'criterion': 'gini', 'max_depth': 5},
  'model_': DecisionTreeClassifier(max_depth=5)},
  5: {'f1': 0.9387755102040817,
  'accuracy': 0.9368421052631579,
  'scaler': StandardScaler(),
  'best_params': {'max_depth': 10, 'n_estimators': 200},
  'model_': RandomForestClassifier(max_depth=10, n_estimators=200)},
  6: {'f1': 0.9441624365482234,
  'accuracy': 0.9421052631578948,
  'scaler': StandardScaler(),
  'best_params': {'learning_rate': 0.1, 'max_depth': 6, 'n_estimators': 200},
  'model_': XGBClassifier(base_score=None, booster=None, callbacks=None,
    colsample_bylevel=None, colsample_bynode=None,
    colsample_bytree=None, device=None, early_stopping_rounds=None,
    enable_categorical=False, eval_metric=None, feature_types=None,
```

Final results dictionary specifying each scaling technique, model and its corresponding best hyperparameters.

```
highest_f1 = 0
best_model = None
best_scaler = None

for scaler_name, scaler_results in results.items():
    for model_id, model_results in scaler_results.items():
        if model_results['f1'] > highest_f1:
            highest_f1 = model_results['f1']
            best_model = model_results['model_']
            best_scaler = model_results['scaler']

print(f"The model with the highest F1 score {highest_f1} is: {type(best_model).__name__}")
print(f"The scaler used: {type(best_scaler).__name__}")

The model with the highest F1 score 0.9441624365482234 is: XGBClassifier
The scaler used: StandardScaler

import joblib

joblib.dump(best_model, 'best_model.pkl')
joblib.dump(best_scaler, 'best_scaler.pkl')

['best_scaler.pkl']
```

Iterating through the results dictionary to find out the highest F1 score and its corresponding model. This becomes our new best_model object file.

We will now pickle this file using joblib to save the model offline which can be used in other environments notebooks to test the model.

```
[ ] X_train1, X_test1, y_train1, y_test1 = train_test_split(df_original.drop('Class', axis=1), df_original['Class'], test_size=0.25)

from sklearn.pipeline import Pipeline
from sklearn.preprocessing import FunctionTransformer

# Create the winsorization function
def winsorize(df, column_to_winsorize="Amount", limits=[0.01, 0.00]):
    winsorized_data = mstats.winsorize(df[column_to_winsorize], limits=limits)
    df[column_to_winsorize] = winsorized_data
    return df

# Create the preprocessing pipeline
pipeline = Pipeline([
    ('winsorize', FunctionTransformer(winsorize)),
    ('scaler', best_scaler),
    ('classifier', best_model)
])

# Fit the pipeline to the data
pipeline.fit(X_train1, y_train1)

# Predict the labels for new data
y_pred1 = pipeline.predict(X_test1)

# Calculate accuracy and F1 score
accuracy_ = accuracy_score(y_test1, y_pred1)
f1_ = f1_score(y_test1, y_pred1)

print(f"Accuracy: {accuracy_}")
print(f"F1 Score: {f1_}")
```

Accuracy: 0.9996052557378898
F1 Score: 0.8640776699029126

Since we have not used a validation set for this dataset, and we had only used approximately 1k rows to train our model, we will try to test it on the overall original df with 284k rows to see how it performs.

Also note an error here in the pipeline where `best_scaler` object is called in the pipeline, being applied to all the rows which is not required since we only have to apply it on “Time” and “Amount” feature.

The F1 score was therefore reduced.

NOTE: this is a demo pipeline, the final pipeline code is below:

```

import joblib
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import FunctionTransformer
from scipy.stats import mstats

# Load the best model (uploaded onto colab)
best_model = joblib.load('/content/best_model.pkl')

# Create the winsorization function
def winsorize(df, column_to_winsorize="Amount", limits=[0.01, 0.00]):
    winsorized_data = mstats.winsorize(df[column_to_winsorize], limits=limits)
    df[column_to_winsorize] = winsorized_data
    return df

def scale_df(df):
    # Load the StandardScaler object from the saved file (uploaded onto colab)
    scaler = joblib.load('/content/best_model.pkl')
    # Scale the "Time" column
    df["Time"] = scaler.transform(df[["Time"]])
    # Scale the "Amount" column
    df["Amount"] = scaler.transform(df[["Amount"]])
    return df

# Create the preprocessing pipeline
pipeline = Pipeline([
    ('winsorize', FunctionTransformer(winsorize)),
    ('scaler', FunctionTransformer(scale_df)),
    ('classifier', best_model)
])

# Pickle the pipeline.
joblib.dump(pipeline, 'final_pipeline.pkl')

```

This is the final pipeline with the features to scale clearly mentioned in the function `scale_df`.

This file is stored separately in `Preprocessing_steps_capstone.ipynb` or `.py`.