Configuration Manual

MSc Research Project
Data Analytics

Eva Figuerola Ullastres
Student ID: x19209371

School of Computing
National College of Ireland

Supervisor: Majid Latifi

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Eva Figuerola Ullastres
Student ID:	x19209371
Programme:	Data Analytics
Year:	2022
Module:	MSc Research Project
Supervisor:	Majid Latifi
Submission Due Date:	30/05/2022
Project Title:	Configuration Manual
Word Count:	2439
Page Count:	26

Signature:	
Date:	30th May 2022

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Eva Figuerola Ullastres
x19209371

1 Introduction

This configuration manual provides information on the system configuration, software, tools and environment requirements in order to conduct the research project in 'Credit Card Fraud Detection using Ensemble Learning Algorithms'. The purpose of creating this document is to demonstrate how the project's technical work was implemented so that it can be replicated if necessary. The goal of the project was to build an ensemble classifier that can accurately detect credit card fraud. In order to achieve the project's goal, a methodology was proposed and followed.

2 System Configuration

This research project was conducted using a MacBook Pro with the following configuration:

- **Processor:** 2.4 GHz Quad-Core Intel Core i5
- **Hard Disk Storage:** 251 GB
- **Memory:** 8 GB 2133 MHz LPDDR3
- **Operating System:** macOS Monterey Version 12.1

3 Software and Tools

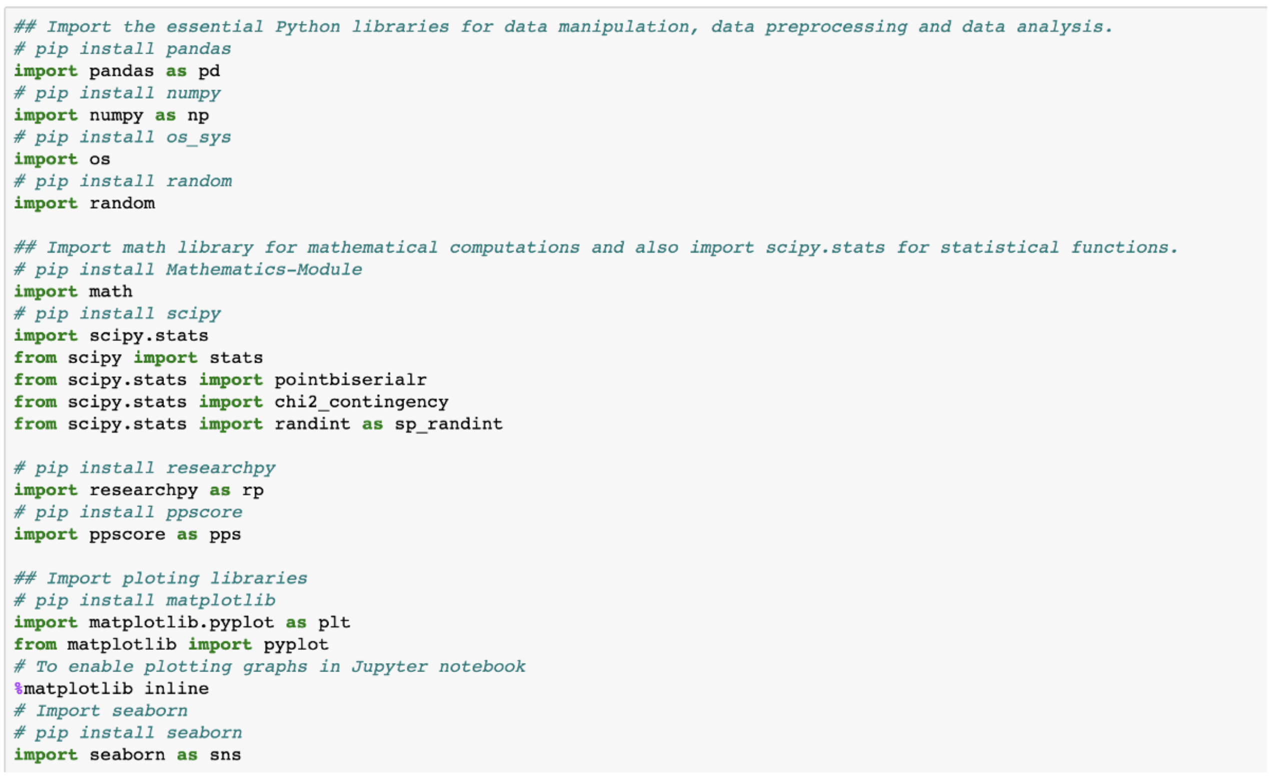
This project was implemented using Python programming language with Jupyter Notebook as Integrated Development Environment (IDE) on the Anaconda platform. The specific versions of the required software/tools are listed below:

- **Programming Language** [1] **:** Python 3.7.4
- **IDE:** Jupyter Notebook 6.0.1
- **Platform** [2]**:** Anaconda 2019.10
- **Tools** [3]**:** Microsoft Excel, Tableau Desktop 2021.3
- **Web Browser:** Google Chrome

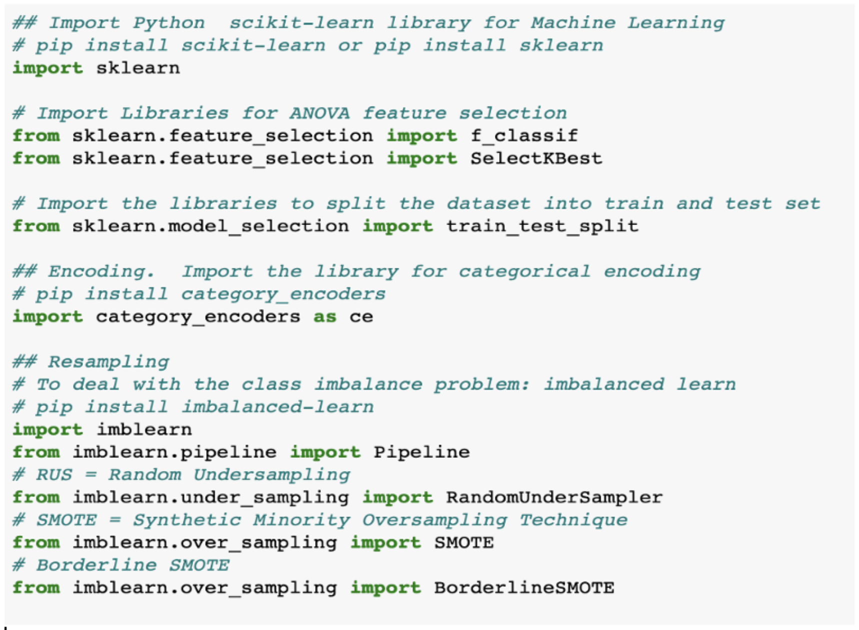
[1] `https://www.python.org/downloads/`
[2] `https://anaconda.org/anaconda/anaconda`
[3] `https://www.tableau.com/products/desktop/download`

4 Python Packages

The Python packages (libraries) required in this project include: Pandas, NumPy, Seaborn, SciPy, Sklearn (also known as Scikit-Learn), Imbalanced-learn and Matplotlib . They were imported as shown in Figure 1 and Figure 2, using the *pip install* command if necessary.

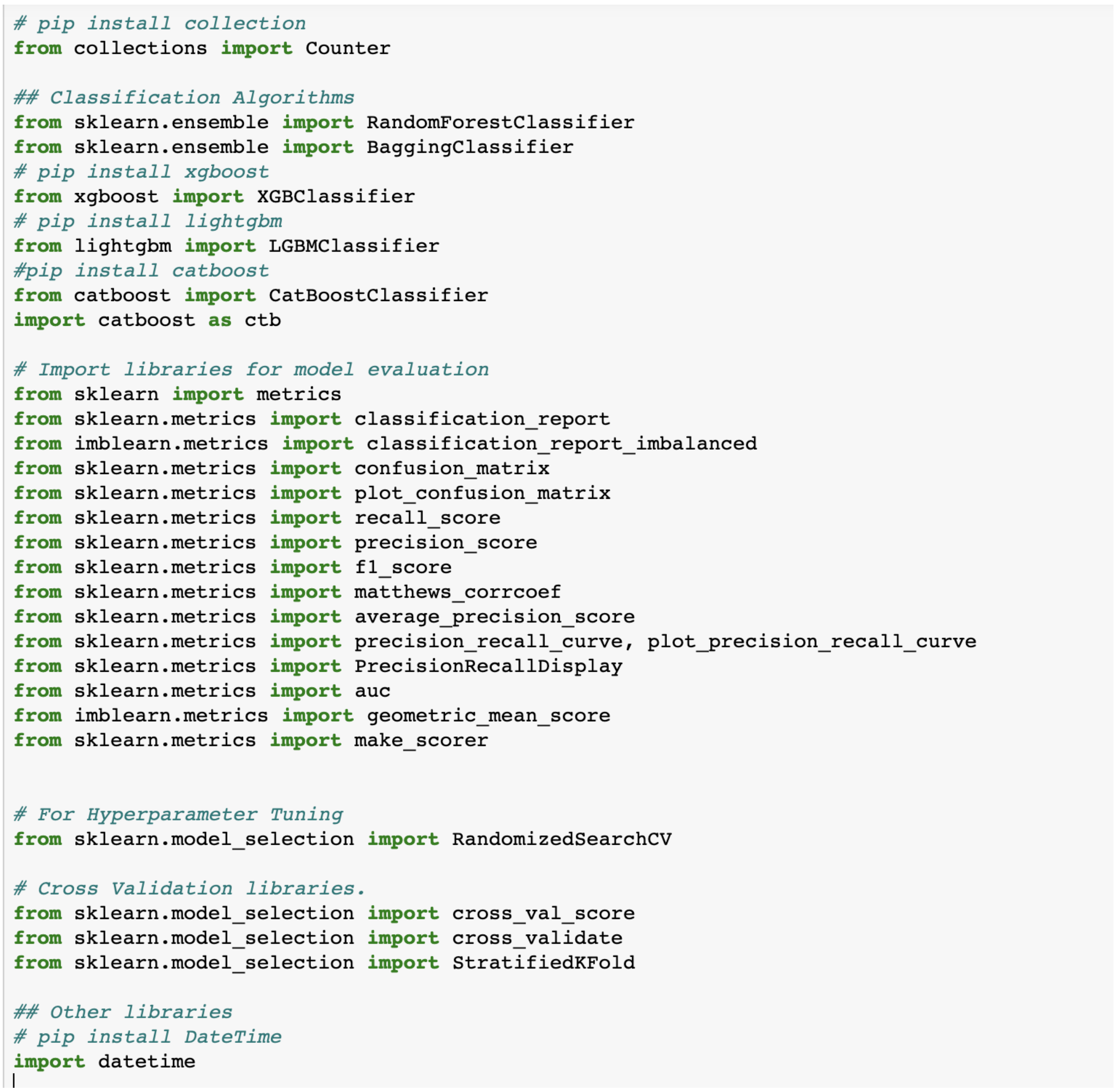


```python
## Import the essential Python libraries for data manipulation, data preprocessing and data analysis.
# pip install pandas
import pandas as pd
# pip install numpy
import numpy as np
# pip install os_sys
import os
# pip install random
import random

## Import math library for mathematical computations and also import scipy.stats for statistical functions.
# pip install Mathematics-Module
import math
# pip install scipy
import scipy.stats
from scipy import stats
from scipy.stats import pointbiserialr
from scipy.stats import chi2_contingency
from scipy.stats import randint as sp_randint

# pip install researchpy
import researchpy as rp
# pip install ppscore
import ppscore as pps

## Import ploting libraries
# pip install matplotlib
import matplotlib.pyplot as plt
from matplotlib import pyplot
# To enable plotting graphs in Jupyter notebook
%matplotlib inline
# Import seaborn
# pip install seaborn
import seaborn as sns
```



```python
## Import Python  scikit-learn library for Machine Learning
# pip install scikit-learn or pip install sklearn
import sklearn

# Import Libraries for ANOVA feature selection
from sklearn.feature_selection import f_classif
from sklearn.feature_selection import SelectKBest

# Import the libraries to split the dataset into train and test set
from sklearn.model_selection import train_test_split

## Encoding.  Import the library for categorical encoding
# pip install category_encoders
import category_encoders as ce

## Resampling
# To deal with the class imbalance problem: imbalanced learn
# pip install imbalanced-learn
import imblearn
from imblearn.pipeline import Pipeline
# RUS = Random Undersampling
from imblearn.under_sampling import RandomUnderSampler
# SMOTE = Synthetic Minority Oversampling Technique
from imblearn.over_sampling import SMOTE
# Borderline SMOTE
from imblearn.over_sampling import BorderlineSMOTE
```

Figure 1: Python Packages



```
# pip install collection
from collections import Counter

## Classification Algorithms
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import BaggingClassifier
# pip install xgboost
from xgboost import XGBClassifier
# pip install lightgbm
from lightgbm import LGBMClassifier
#pip install catboost
from catboost import CatBoostClassifier
import catboost as ctb

# Import libraries for model evaluation
from sklearn import metrics
from sklearn.metrics import classification_report
from imblearn.metrics import classification_report_imbalanced
from sklearn.metrics import confusion_matrix
from sklearn.metrics import plot_confusion_matrix
from sklearn.metrics import recall_score
from sklearn.metrics import precision_score
from sklearn.metrics import f1_score
from sklearn.metrics import matthews_corrcoef
from sklearn.metrics import average_precision_score
from sklearn.metrics import precision_recall_curve, plot_precision_recall_curve
from sklearn.metrics import PrecisionRecallDisplay
from sklearn.metrics import auc
from imblearn.metrics import geometric_mean_score
from sklearn.metrics import make_scorer


# For Hyperparameter Tuning
from sklearn.model_selection import RandomizedSearchCV

# Cross Validation libraries.
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import cross_validate
from sklearn.model_selection import StratifiedKFold

## Other libraries
# pip install DateTime
import datetime
```

Figure 2: Python Packages (continuation)

5 Data Collection

The dataset used in this research was downloaded from the kaggle repository `https://www.kaggle.com/datasets/kartik2112/fraud-detection`. See Figure 3. The dataset contains two files in a CSV format. The files were stored on the computer hard drive.

6 Data Loading

The csv files fraudTrain.csv and fraudTest.csv were converted to a Pandas Dataframe format for analysis, and the two files were then concatenated in order to get a better

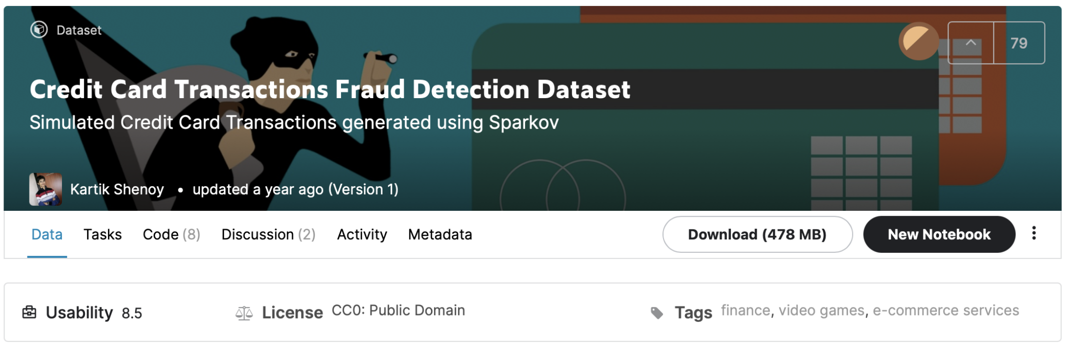
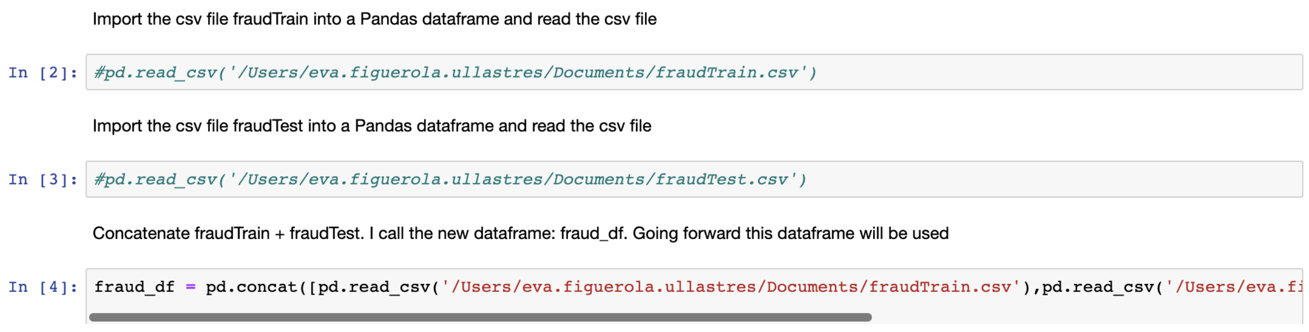

Dataset
Credit Card Transactions Fraud Detection Dataset
Simulated Credit Card Transactions generated using Sparkov
Kartik Shenoy • updated a year ago (Version 1)
79
Data
Tasks
Code (8)
Discussion (2)
Activity
Metadata
Download (478 MB)
New Notebook
Usability 8.5
License CC0: Public Domain
Tags finance, video games, e-commerce services

Figure 3: Credit Card Fraud Dataset

insight from the data. The dataset was loaded in Python Jupyter Notebook as seen in Figure 4.



Import the csv file fraudTrain into a Pandas dataframe and read the csv file

```
In [2]: #pd.read_csv('/Users/eva.figuerola.ullastres/Documents/fraudTrain.csv')
```

Import the csv file fraudTest into a Pandas dataframe and read the csv file

```
In [3]: #pd.read_csv('/Users/eva.figuerola.ullastres/Documents/fraudTest.csv')
```

Concatenate fraudTrain + fraudTest. I call the new dataframe: fraud_df. Going forward this dataframe will be used

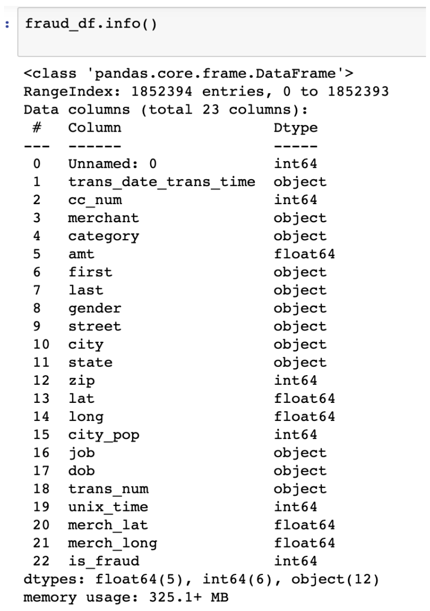
```
In [4]: fraud_df = pd.concat([pd.read_csv('/Users/eva.figuerola.ullastres/Documents/fraudTrain.csv'),pd.read_csv('/Users/eva.fi
```

Figure 4: Data Loading

7 Data Preprocessing

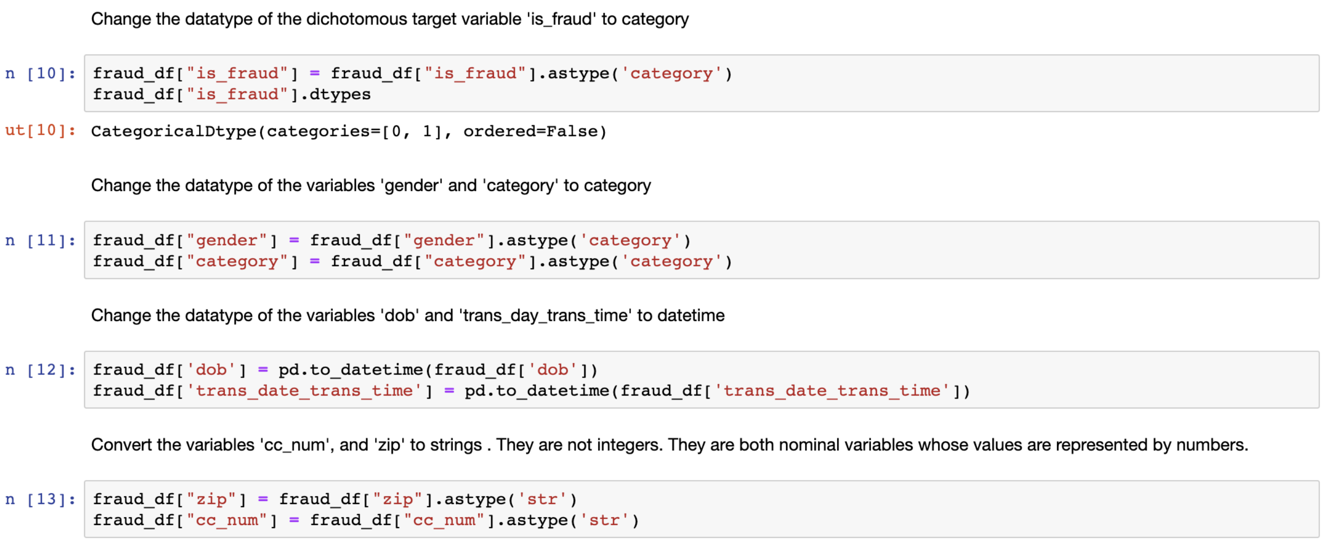
The below data transformation and data cleaning tasks were conducted as part of the data preprocessing step. Pandas and NumPy packages were used in this step:

- Check the structure of the Pandas DataFrame. See Figure 5
- Change the datatype of the target variable `'is_fraud'` to category.
  Change the datatype of the variables *'gender'* and *'category'* to category.
  Change the datatype of the variables *'dob'* and *'trans-date-trans-time'* to datetime.
  Convert the variables *'cc-num'*, and *'zip'* to strings as they are not integers; they are both nominal variables whose values are represented by numbers. See Figure 6
- Remove the column *'Unnamed: 0'* as it is totally irrelevant. Remove variables that have almost unique values. See Figure 7
- Check for missing values and duplicates. See Figure 8
- Check the cardinality (number of unique values) for the categorical variables. See Figure 9



```
: fraud_df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1852394 entries, 0 to 1852393
Data columns (total 23 columns):
 #   Column                 Dtype  
---  ------                 -----  
 0   Unnamed: 0             int64  
 1   trans_date_trans_time  object 
 2   cc_num                 int64  
 3   merchant               object 
 4   category               object 
 5   amt                    float64
 6   first                  object 
 7   last                   object 
 8   gender                 object 
 9   street                 object 
 10  city                   object 
 11  state                  object 
 12  zip                    int64  
 13  lat                    float64
 14  long                   float64
 15  city_pop               int64  
 16  job                    object 
 17  dob                    object 
 18  trans_num              object 
 19  unix_time              int64  
 20  merch_lat              float64
 21  merch_long             float64
 22  is_fraud               int64  
dtypes: float64(5), int64(6), object(12)
memory usage: 325.1+ MB
```

Figure 5: Structure of the Pandas DataFrame



Change the datatype of the dichotomous target variable 'is_fraud' to category

```
n [10]: fraud_df["is_fraud"] = fraud_df["is_fraud"].astype('category')
        fraud_df["is_fraud"].dtypes

ut[10]: CategoricalDtype(categories=[0, 1], ordered=False)
```

Change the datatype of the variables 'gender' and 'category' to category

```
n [11]: fraud_df["gender"] = fraud_df["gender"].astype('category')
        fraud_df["category"] = fraud_df["category"].astype('category')
```

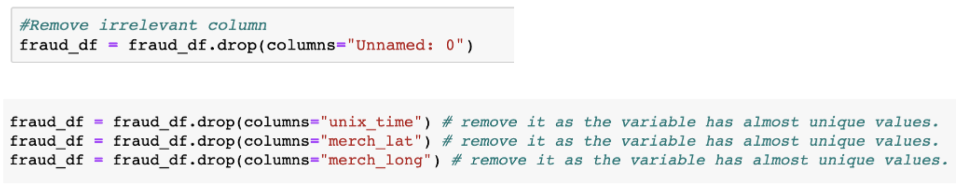
Change the datatype of the variables 'dob' and 'trans_day_trans_time' to datetime

```
n [12]: fraud_df['dob'] = pd.to_datetime(fraud_df['dob'])
        fraud_df['trans_date_trans_time'] = pd.to_datetime(fraud_df['trans_date_trans_time'])
```

Convert the variables 'cc_num', and 'zip' to strings . They are not integers. They are both nominal variables whose values are represented by numbers.

```
n [13]: fraud_df["zip"] = fraud_df["zip"].astype('str')
        fraud_df["cc_num"] = fraud_df["cc_num"].astype('str')
```

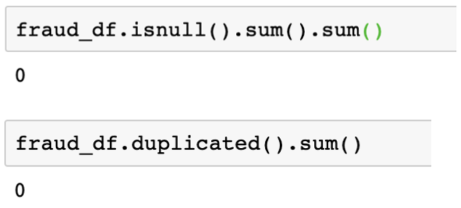
Figure 6: Data Transformation



```
#Remove irrelevant column
fraud_df = fraud_df.drop(columns="Unnamed: 0")
```

```
fraud_df = fraud_df.drop(columns="unix_time") # remove it as the variable has almost unique values.
fraud_df = fraud_df.drop(columns="merch_lat") # remove it as the variable has almost unique values.
fraud_df = fraud_df.drop(columns="merch_long") # remove it as the variable has almost unique values.
```

Figure 7: Data Cleaning



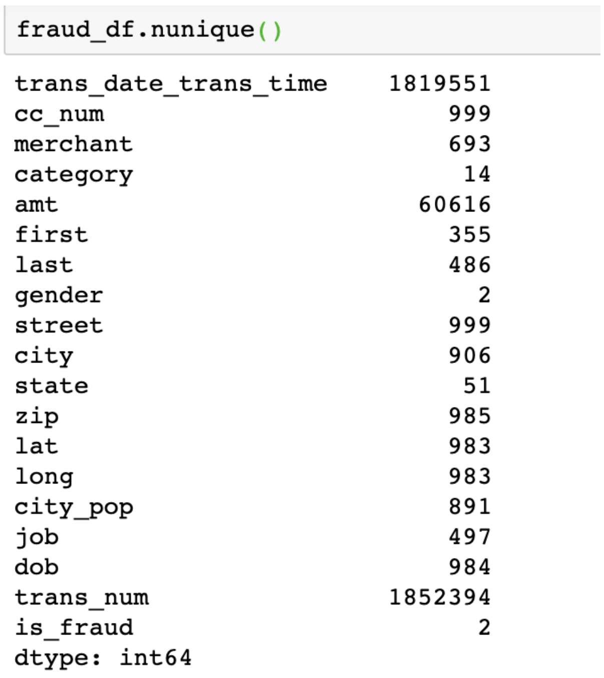
```
fraud_df.isnull().sum().sum()
```

```
0
```

```
fraud_df.duplicated().sum()
```

```
0
```

Figure 8: Checking for Missing values and Duplicates



```
fraud_df.nunique()
```

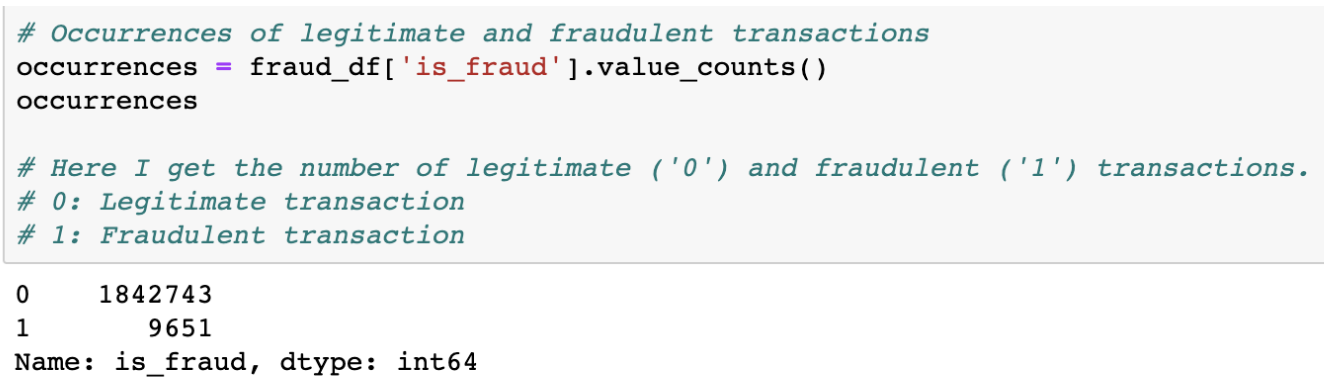
```
trans_date_trans_time    1819551
cc_num                       999
merchant                     693
category                      14
amt                        60616
first                        355
last                         486
gender                         2
street                       999
city                         906
state                         51
zip                          985
lat                          983
long                         983
city_pop                     891
job                          497
dob                          984
trans_num                1852394
is_fraud                       2
dtype: int64
```

Figure 9: Number of Unique values (for the categorical variables this is the cardinality)

8 Exploratory Data Analysis

Exploratory Data Analysis (EDA) was performed in order to get an general understanding of the data and its underlying structure. Pandas, NumPy and Seaborn packages were used in this step. The below tasks were performed as part of the EDA process:

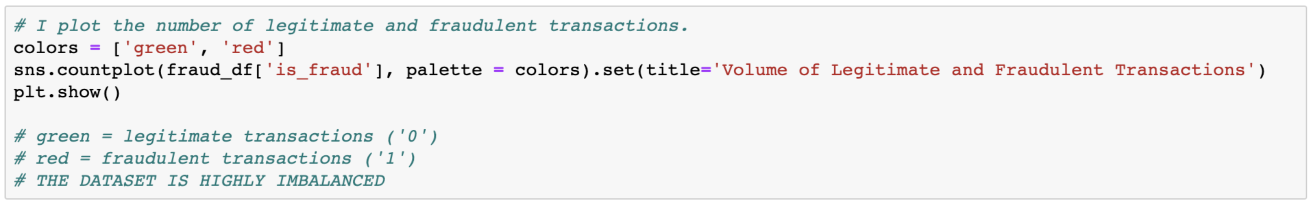
- EDA for the target variable 'is-fraud'. A bar plot was created to see the class distribution of the target variable. As seen in Figure 10 and Figure 11, the dataset is heavily imbalanced.
- Bar plots for the rest of categorical variables. Figure 12 shows the bar plot for the variable 'category'. Same code can be reused for the rest of categorical variables.
- Descriptive statistics (Figure 13) and histograms for the continuous variables.



```
# Occurrences of legitimate and fraudulent transactions
occurrences = fraud_df['is_fraud'].value_counts()
occurrences

# Here I get the number of legitimate ('0') and fraudulent ('1') transactions.
# 0: Legitimate transaction
# 1: Fraudulent transaction
```

```
0    1842743
1       9651
Name: is_fraud, dtype: int64
```

Figure 10: Occurrences of Legitimate and Fraudulent Transactions



```
# I plot the number of legitimate and fraudulent transactions.
colors = ['green', 'red']
sns.countplot(fraud_df['is_fraud'], palette = colors).set(title='Volume of Legitimate and Fraudulent Transactions')
plt.show()

# green = legitimate transactions ('0')
# red = fraudulent transactions ('1')
# THE DATASET IS HIGHLY IMBALANCED
```

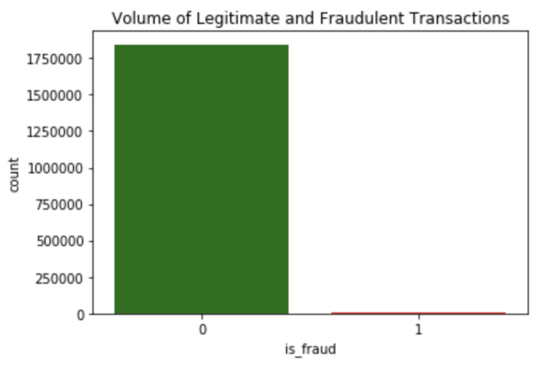


Figure 11: Volume of Legitimate and Fraudulent Transactions

From looking at the descriptive statistics for the ’amount’ variable shown in Figure 13, it is observed that the average amount for a fraudulent transaction is $530.66 in comparison with the average amount for a legitimate transaction which is $67.65. This demonstrates that despite fraudulent transactions being infrequent, not detecting them can result significant financial losses.

```
# I plot the number of legitimate and fraudulent transactions by category
colors = ['green', 'red']
plt.figure(figsize=(20,8))
plt.title('Volume of Legitimate and Fraudulent transactions by Category')
sns.barplot(x="category", y='trans_num' ,hue="is_fraud",palette=colors,
            data=fraud_df.groupby(['category','is_fraud']).agg({'trans_num':'count'}).reset_index())
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f9e504f4290>
```

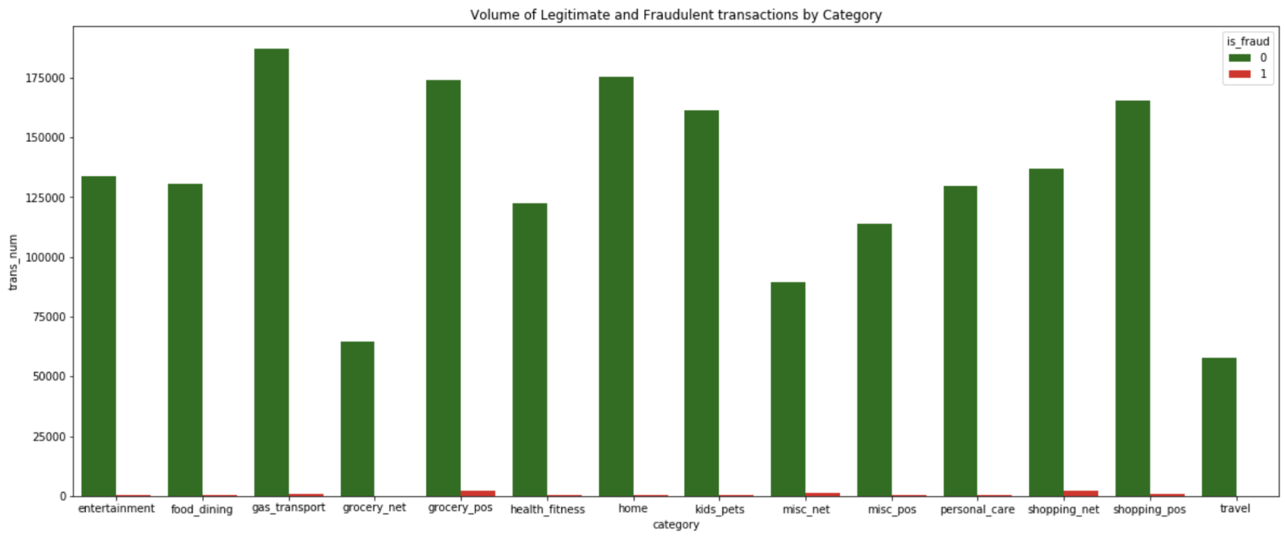


Figure 12: Volume of Legitimate and Fraudulent Transactions by Category

Compute some relevant summary statistics for the variable 'Amount'('amt')

```
max_legitimate_amt=legitimate['amt'].max()
max_fraudulent_amt=fraudulent['amt'].max()
import statistics
mean_legitimate_amt= legitimate['amt'].mean()
mean_fraudulent_amt= fraudulent['amt'].mean()
median_legitimate_amt= legitimate['amt'].median()
median_fraudulent_amt= fraudulent['amt'].median()

print('Maximun Legitimate Transactions Amount: {}'.format(max_legitimate_amt))
print('Maximun Fraudulent Transactions Amount: {}'.format(max_fraudulent_amt))

print('Average Legitimate Transactions Amount: {}'.format(mean_legitimate_amt))
print('Average Fraudulent Transactions Amount: {}'.format(mean_fraudulent_amt))

print('Median of Legitimate Transactions Amount: {}'.format(median_legitimate_amt))
print('Median of Fraudulent Transactions Amount: {}'.format(median_fraudulent_amt))
```

```
Maximun Legitimate Transactions Amount: 28948.9
Maximun Fraudulent Transactions Amount: 1376.04
Average Legitimate Transactions Amount: 67.6512778613193
Average Fraudulent Transactions Amount: 530.6614122888789
Median of Legitimate Transactions Amount: 47.24
Median of Fraudulent Transactions Amount: 390.0
```

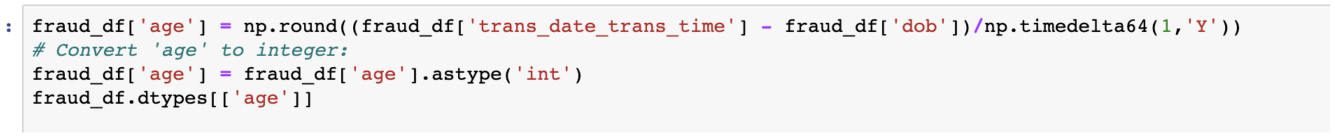
Figure 13: Descriptive Statistics for the Variable Amount

9 Feature Engineering

Feature engineering was performed to derive more informative features. Pandas, NumPy, SciPy and Seaborn packages were used in this step.

9.1 Create the variable 'age' at the time of the transaction

Since the variable 'dob' does not bring much information in its raw format, the variable 'age' was derived, which is more meaningful. The variable 'age' is computed by subtracting the date of birth ('dob') from the transaction date ('trans-date-trans-time) as seen in Figure 14.

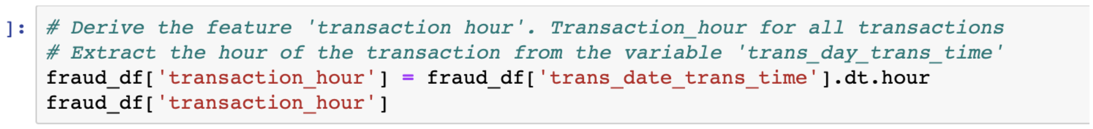


```
: fraud_df['age'] = np.round((fraud_df['trans_date_trans_time'] - fraud_df['dob'])/np.timedelta64(1,'Y'))
  # Convert  'age'  to integer:
  fraud_df['age'] = fraud_df['age'].astype('int')
  fraud_df.dtypes[['age']]
```

Figure 14: Age at the Time of the Transaction

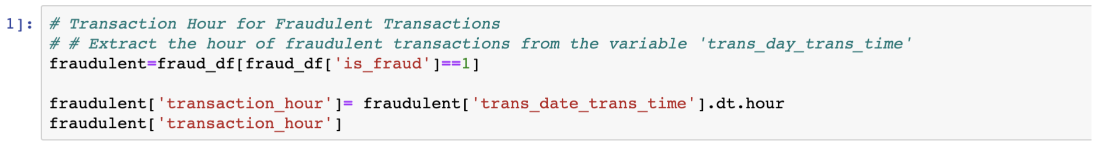
9.2 Create the variable 'transaction-hour'

The time of the transaction may be an important predictor of credit card fraud. However, the variable 'trans-date-trans-time' is not very informative in its raw format; hence the reason to split this variable into three different variables: transaction hour, day of the week and month. Figure 15 shows how the variable 'transaction-hour' is computed from the variable 'trans-date-trans-time'. Figure 16 shows how the variable 'transaction-hour' for fraudulent transactions is computed.



```
]: # Derive the feature 'transaction hour'. Transaction_hour for all transactions
   # Extract the hour of the transaction from the variable 'trans_day_trans_time'
   fraud_df['transaction_hour'] = fraud_df['trans_date_trans_time'].dt.hour
   fraud_df['transaction_hour']
```

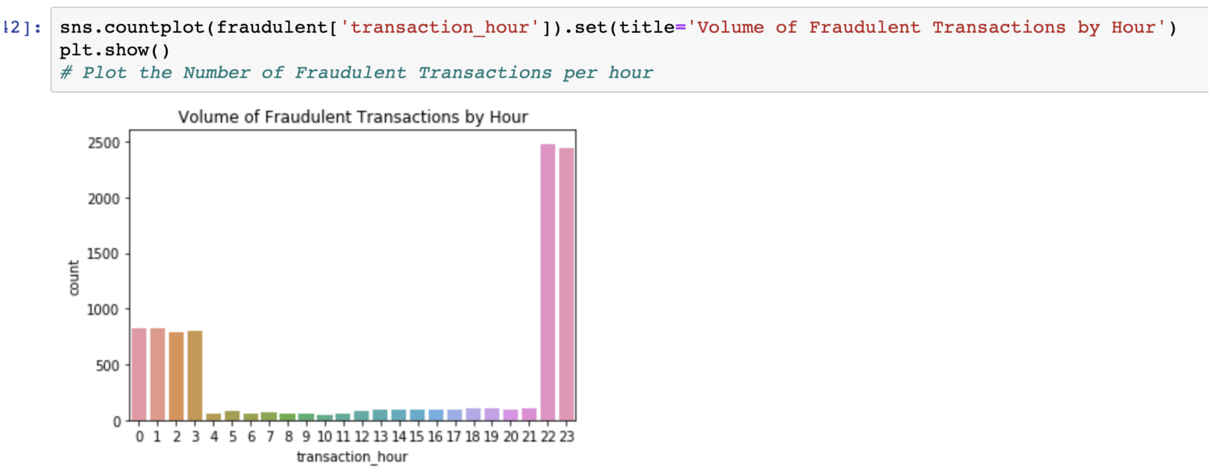
Figure 15: Transaction Hour



```
1]: # Transaction Hour for Fraudulent Transactions
    # # Extract the hour of fraudulent transactions from the variable 'trans_day_trans_time'
    fraudulent=fraud_df[fraud_df['is_fraud']==1]

    fraudulent['transaction_hour']= fraudulent['trans_date_trans_time'].dt.hour
    fraudulent['transaction_hour']
```

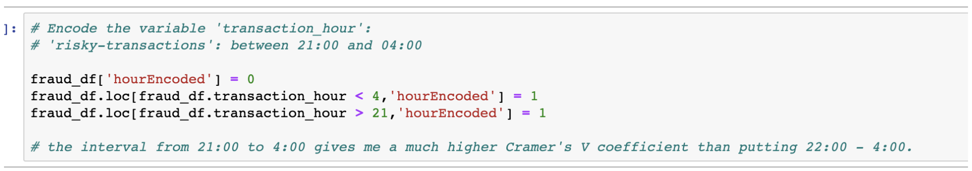
Figure 16: Transaction Hour for Fraudulent Transactions

Figure 17 shows the volume of fraudulent transactions by hour. It was observed that most of the fraudulent transactions occur between 10pm and 12am. The variable 'transaction-hour' seems to have some association with the 'class' of the transaction since the majority of fraudulent transactions occur at night time. The 'transaction-hour' variable was encoded as seen in Figure 18.



```
[42]: sns.countplot(fraudulent['transaction_hour']).set(title='Volume of Fraudulent Transactions by Hour')
plt.show()
# Plot the Number of Fraudulent Transactions per hour
```

Figure 17: Volume of Fraudulent Transactions by Hour

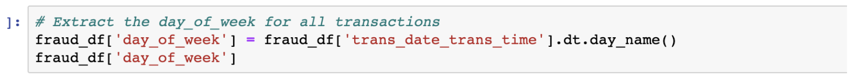


```
# Encode the variable 'transaction_hour':
# 'risky-transactions': between 21:00 and 04:00

fraud_df['hourEncoded'] = 0
fraud_df.loc[fraud_df.transaction_hour < 4,'hourEncoded'] = 1
fraud_df.loc[fraud_df.transaction_hour > 21,'hourEncoded'] = 1

# the interval from 21:00 to 4:00 gives me a much higher Cramer's V coefficient than putting 22:00 - 4:00.
```

Figure 18: Hour Encoded

9.3 Create the variable 'day-of-week'

The variable 'day of the week' is derived from the variable 'trans-date-trans-time' as seen in Figure 19.

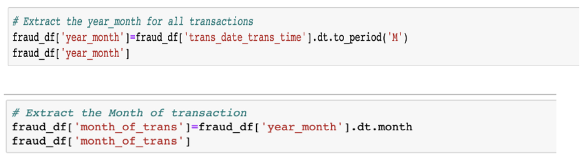


```
# Extract the day_of_week for all transactions
fraud_df['day_of_week'] = fraud_df['trans_date_trans_time'].dt.day_name()
fraud_df['day_of_week']
```

Figure 19: Day of the Week

9.4 Create the variable 'month of transaction'

The variable 'month of transaction' is derived from the variable 'trans-date-trans-time' as seen in Figure 20.



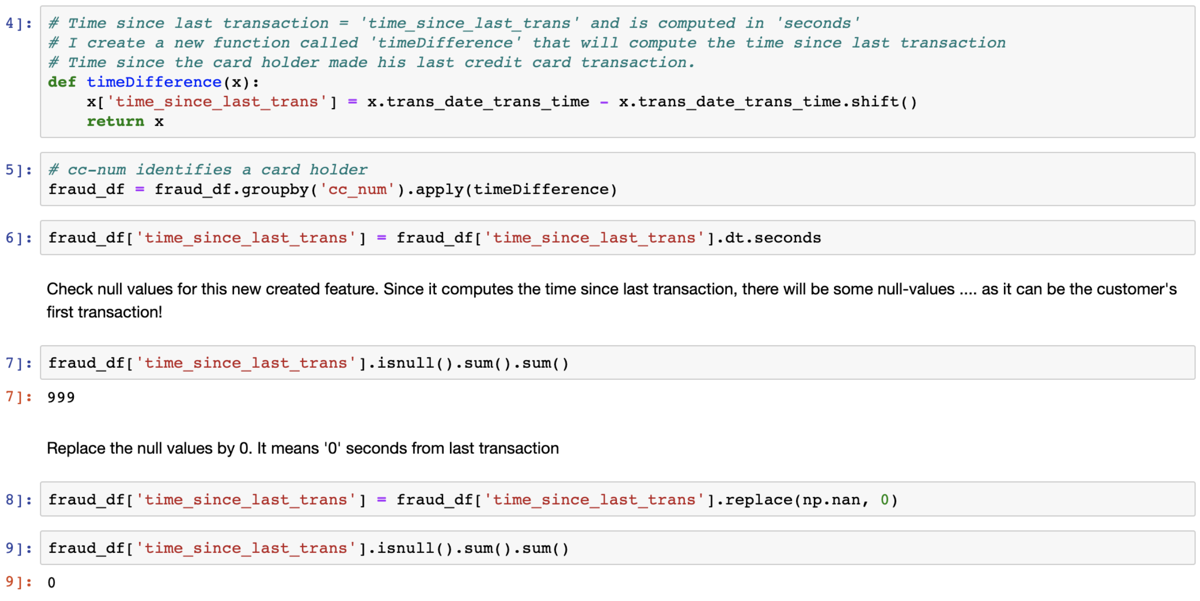
```
# Extract the year_month for all transactions
fraud_df['year_month']=fraud_df['trans_date_trans_time'].dt.to_period('M')
fraud_df['year_month']
```

```
# Extract the Month of transaction
fraud_df['month_of_trans']=fraud_df['year_month'].dt.month
fraud_df['month_of_trans']
```

Figure 20: Month of Transaction

9.5 Create the variable 'time since last transaction'

The variable 'time since last transaction' (in seconds) is derived from the variable 'trans-date-trans-time' as seen in Figure 21.



```
4]: # Time since last transaction = 'time_since_last_trans' and is computed in 'seconds'
    # I create a new function called 'timeDifference' that will compute the time since last transaction
    # Time since the card holder made his last credit card transaction.
    def timeDifference(x):
        x['time_since_last_trans'] = x.trans_date_trans_time - x.trans_date_trans_time.shift()
        return x
```

```
5]: # cc-num identifies a card holder
    fraud_df = fraud_df.groupby('cc_num').apply(timeDifference)
```

```
6]: fraud_df['time_since_last_trans'] = fraud_df['time_since_last_trans'].dt.seconds
```

Check null values for this new created feature. Since it computes the time since last transaction, there will be some null-values .... as it can be the customer's first transaction!

```
7]: fraud_df['time_since_last_trans'].isnull().sum().sum()
7]: 999
```

Replace the null values by 0. It means '0' seconds from last transaction

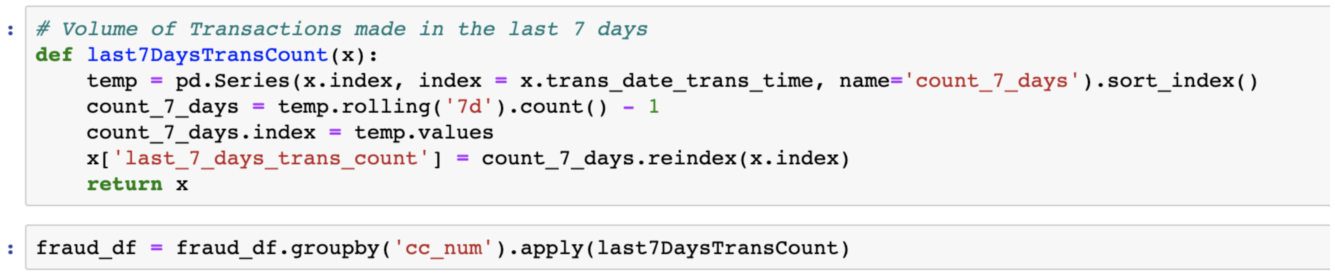
```
8]: fraud_df['time_since_last_trans'] = fraud_df['time_since_last_trans'].replace(np.nan, 0)
```

```
9]: fraud_df['time_since_last_trans'].isnull().sum().sum()
9]: 0
```

Figure 21: Time since last transaction (in seconds)

9.6 Generate Frequencies of Transactions made in the last 1/ 7 / 14 / 30 / 60 days

The frequency of transactions could be an important predictor of credit card fraud, because if the number of transactions made using the same card suddenly increases, it could be a sign of a fraudulent transaction. Figure 22 shows how the number of transactions made over the last 7 days is computed. Same code can be reused to compute frequencies over the last day, 14, 30, 60 days, etc.



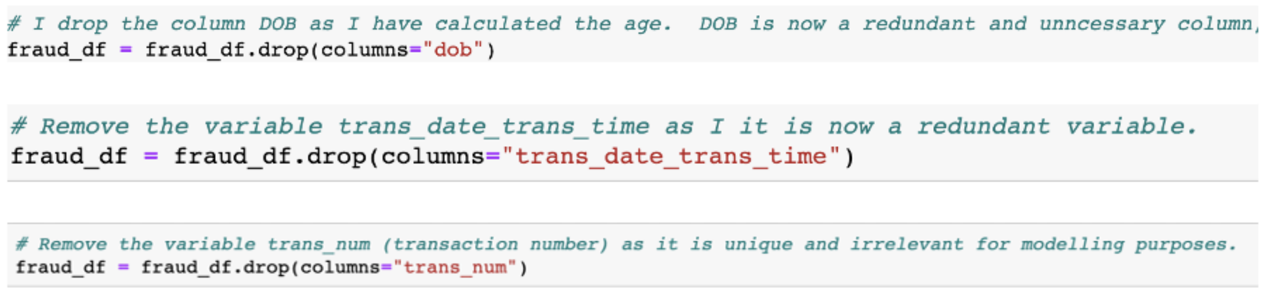
```
: # Volume of Transactions made in the last 7 days
  def last7DaysTransCount(x):
      temp = pd.Series(x.index, index = x.trans_date_trans_time, name='count_7_days').sort_index()
      count_7_days = temp.rolling('7d').count() - 1
      count_7_days.index = temp.values
      x['last_7_days_trans_count'] = count_7_days.reindex(x.index)
      return x
```

```
: fraud_df = fraud_df.groupby('cc_num').apply(last7DaysTransCount)
```

Figure 22: Last 7 days Transaction Count

After conducting feature engineering, the variables 'dob' and 'trans-date-trans-time' were removed since new variables were derived from them and they became redundant.

The variable 'trans-num' was also removed as it is a transaction identifier and is not relevant for modelling. See Figure 23.



```
# I drop the column DOB as I have calculated the age.  DOB is now a redundant and unncessary column,
fraud_df = fraud_df.drop(columns="dob")
```

```
# Remove the variable trans_date_trans_time as I it is now a redundant variable.
fraud_df = fraud_df.drop(columns="trans_date_trans_time")
```

```
# Remove the variable trans_num (transaction number) as it is unique and irrelevant for modelling purposes.
fraud_df = fraud_df.drop(columns="trans_num")
```

Figure 23: Removing redundant and irrelevant variables

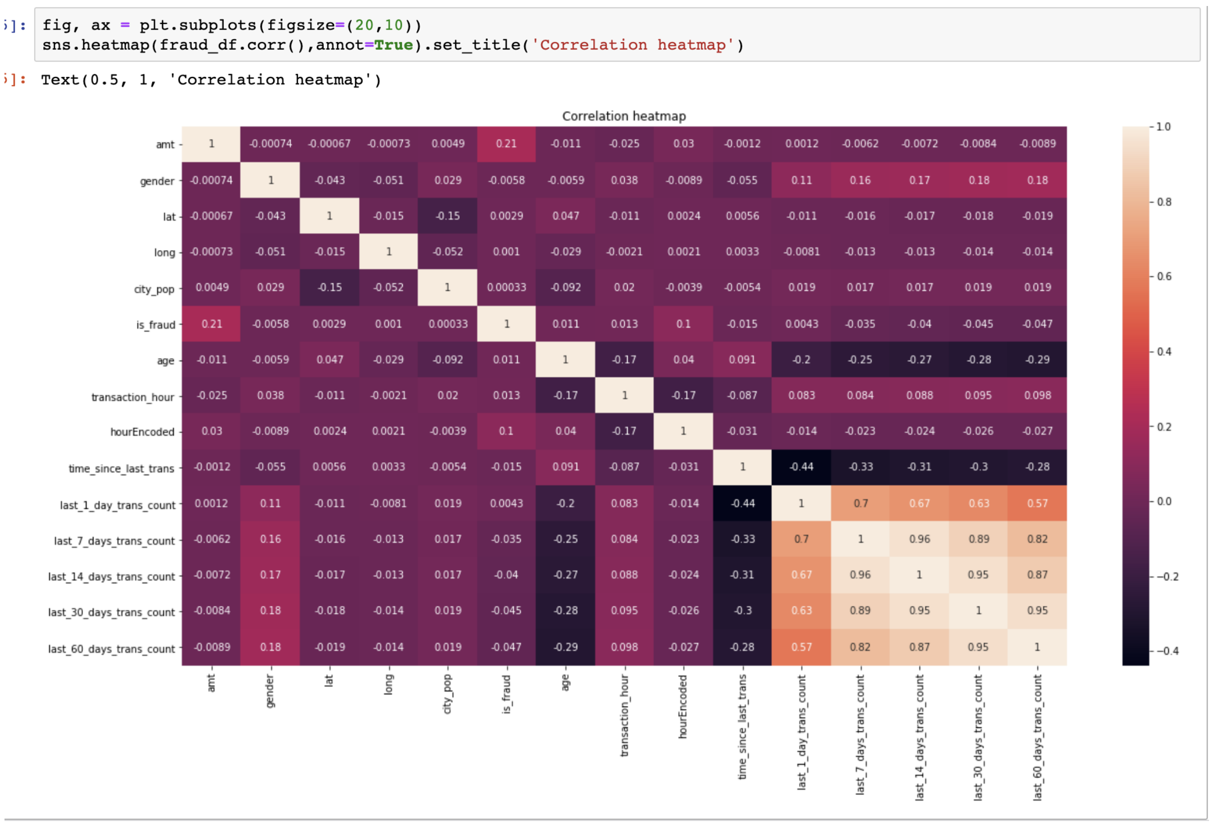
10 Feature Selection

Feature selection was conducted to improve the predictive performance and reduce overfitting and the training time of the classifiers. Since the dataset used in this research contains a mix of continuous and categorical features, different statistical tests were conducted to determine the most relevant features for modelling. The Sklearn and SciPy packages were used in this step.

10.1 Feature Selection for Continuous Variables

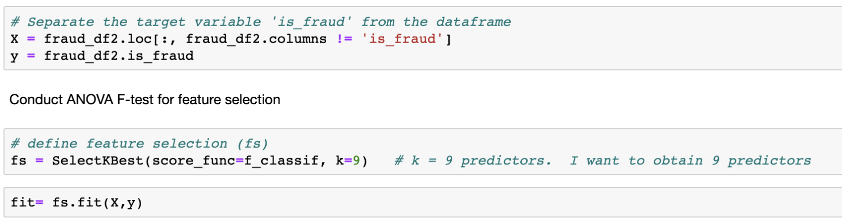
A correlation plot was computed to visualise the correlation between the continuous predictors and the target variable ('class of transaction'). From looking at the correlation plot in Figure 24 it is observed that the variable 'amount' has the strongest relationship with the class of transaction (`'is_fraud'`).

The univariate feature selection method SelectKBest from `sklearn.feature_selection`, which uses ANOVA F-values to compute the feature importance scores, was used to select the most important features. The 8 most informative features to predict the class of transaction were selected. See F-test in Figure 25 and the selected features in Figure 26.



```
]: fig, ax = plt.subplots(figsize=(20,10))
   sns.heatmap(fraud_df.corr(),annot=True).set_title('Correlation heatmap')

]: Text(0.5, 1, 'Correlation heatmap')
```

Figure 24: Correlation Matrix



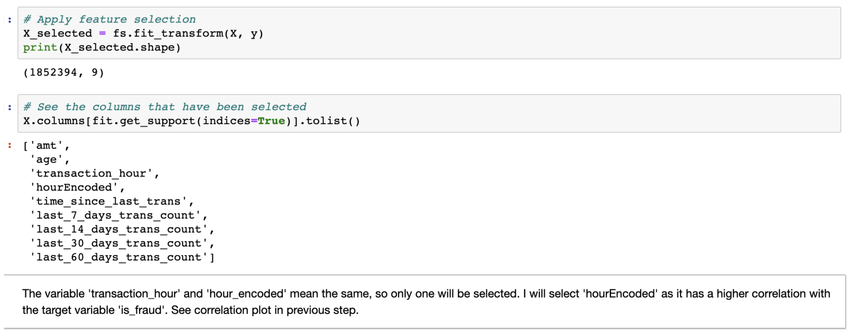
```
# Separate the target variable 'is_fraud' from the dataframe
X = fraud_df2.loc[:, fraud_df2.columns != 'is_fraud']
y = fraud_df2.is_fraud
```

Conduct ANOVA F-test for feature selection

```
# define feature selection (fs)
fs = SelectKBest(score_func=f_classif, k=9)    # k = 9 predictors.  I want to obtain 9 predictors
```

```
fit= fs.fit(X,y)
```

Figure 25: Anova F- Test for Feature Selection



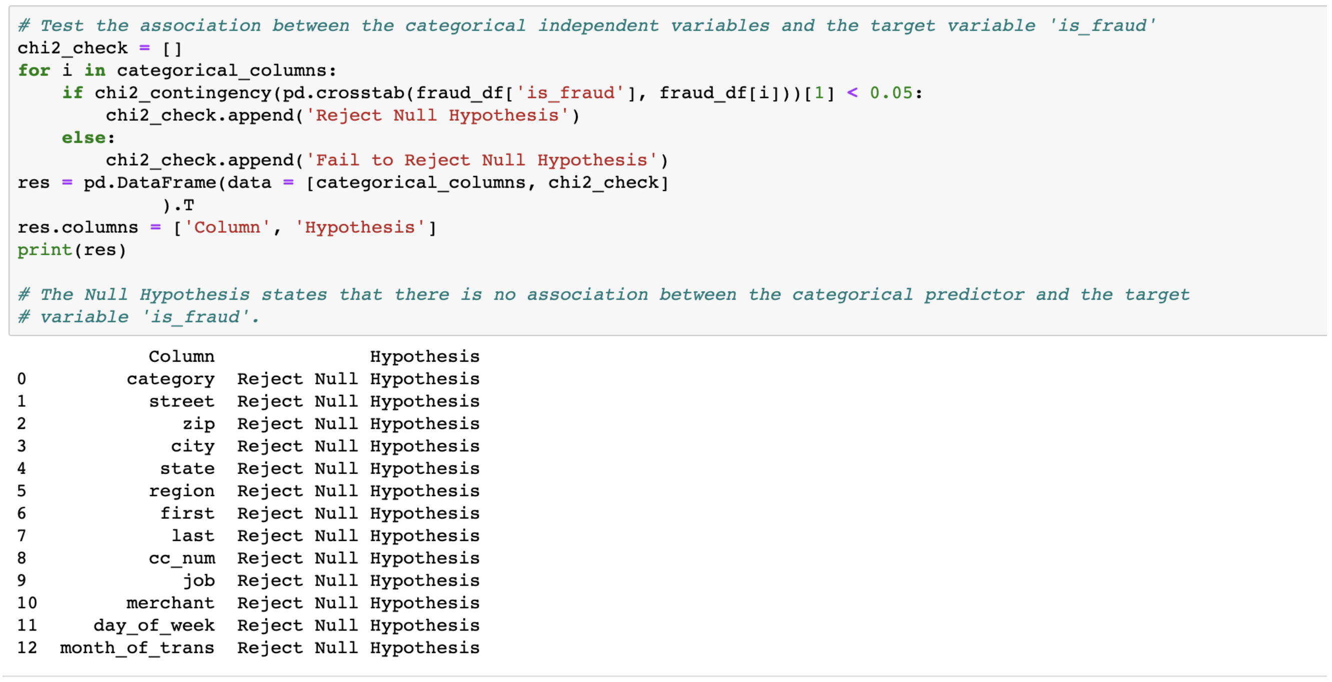
```
: # Apply feature selection
  X_selected = fs.fit_transform(X, y)
  print(X_selected.shape)

  (1852394, 9)

: # See the columns that have been selected
  X.columns[fit.get_support(indices=True)].tolist()

: ['amt',
   'age',
   'transaction_hour',
   'hourEncoded',
   'time_since_last_trans',
   'last_7_days_trans_count',
   'last_14_days_trans_count',
   'last_30_days_trans_count',
   'last_60_days_trans_count']
```

The variable 'transaction_hour' and 'hour_encoded' mean the same, so only one will be selected. I will select 'hourEncoded' as it has a higher correlation with the target variable 'is_fraud'. See correlation plot in previous step.

Figure 26: Selected Features after conducting Anova F-test

10.2 Feature Selection for Categorical Variables

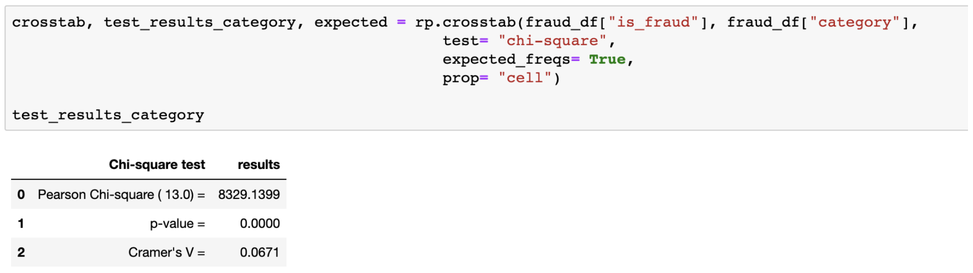
Chi-Square test was conducted to determine whether there is an association between each of the categorical features and the target variable (`'is_fraud'`). The SciPy package was used to perform Chi-Square test. See Figure 27. Cramer's V test was performed to quantify the strength of the association between each of the categorical features and the 'class of transaction'. Figure 28 shows how Cramer's V test was computed using the research.py package.



```
# Test the association between the categorical independent variables and the target variable 'is_fraud'
chi2_check = []
for i in categorical_columns:
    if chi2_contingency(pd.crosstab(fraud_df['is_fraud'], fraud_df[i]))[1] < 0.05:
        chi2_check.append('Reject Null Hypothesis')
    else:
        chi2_check.append('Fail to Reject Null Hypothesis')
res = pd.DataFrame(data = [categorical_columns, chi2_check]
             ).T
res.columns = ['Column', 'Hypothesis']
print(res)

# The Null Hypothesis states that there is no association between the categorical predictor and the target
# variable 'is_fraud'.
```

```
           Column              Hypothesis
0        category  Reject Null Hypothesis
1          street  Reject Null Hypothesis
2             zip  Reject Null Hypothesis
3            city  Reject Null Hypothesis
4           state  Reject Null Hypothesis
5          region  Reject Null Hypothesis
6           first  Reject Null Hypothesis
7            last  Reject Null Hypothesis
8          cc_num  Reject Null Hypothesis
9             job  Reject Null Hypothesis
10       merchant  Reject Null Hypothesis
11    day_of_week  Reject Null Hypothesis
12 month_of_trans  Reject Null Hypothesis
```

Figure 27: Chi-Square Test



```
crosstab, test_results_category, expected = rp.crosstab(fraud_df["is_fraud"], fraud_df["category"],
                                                        test= "chi-square",
                                                        expected_freqs= True,
                                                        prop= "cell")

test_results_category
```

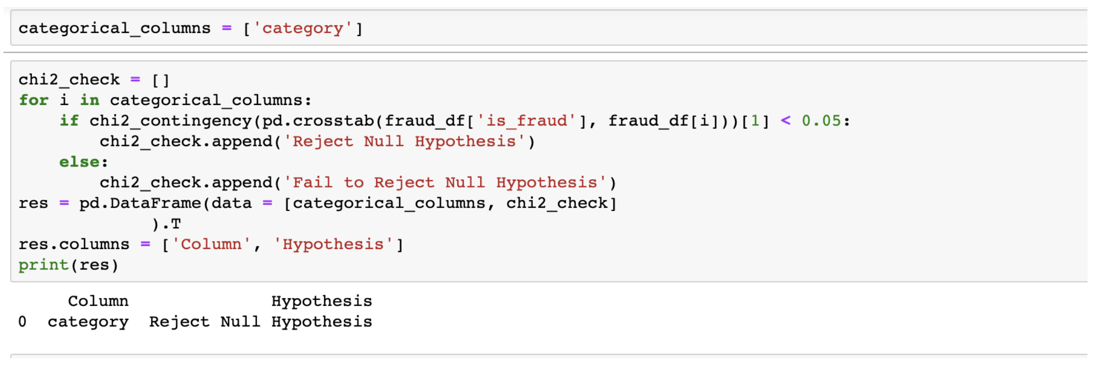
	Chi-square test	results
0	Pearson Chi-square (13.0) =	8329.1399
1	p-value =	0.0000
2	Cramer's V =	0.0671

Figure 28: Cramer's V Test

From looking at Figure 27 it is observed that all the categorical predictors are associated with the 'class' of transaction. However, according Cramer's V test, the association of the categorical predictors with the 'class' of transaction is weak/very weak. But this does not mean that they are not important. Cramer's V coefficient only measures the 'effect' size; however, it needs to be considered that when having a large sample size, small effects can become significant[4].

[4] https://www.datascienceblog.net/post/statistical_test/effect_size/

For feature selection purposes, a Post-Hoc Test was conducted after Chi-Square test in order to decide whether to select a particular categorical predictor for modelling. Since Chi-Square tests the data as a whole, when having multiple classes within a categorical variable, we can not tell which class/es are responsible for the relationship between the categorical predictor and the target variable. This is why it was decided to conduct a Post-Hoc test using Bonferroni Adjustment. Only those predictors that all their categories have a significant relationship with the target variable `'is_fraud'` were selected for modelling. A Bonferroni test was computed for each categorical predictor. Figure 29 and Figure 30 show how the Bonferroni test was computed; same code can be reused for the different variables. Among all the categorical predictors, only the predictors 'category' and 'day of the week' were selected for modelling. The selected features for modelling are shown in Figure 31.

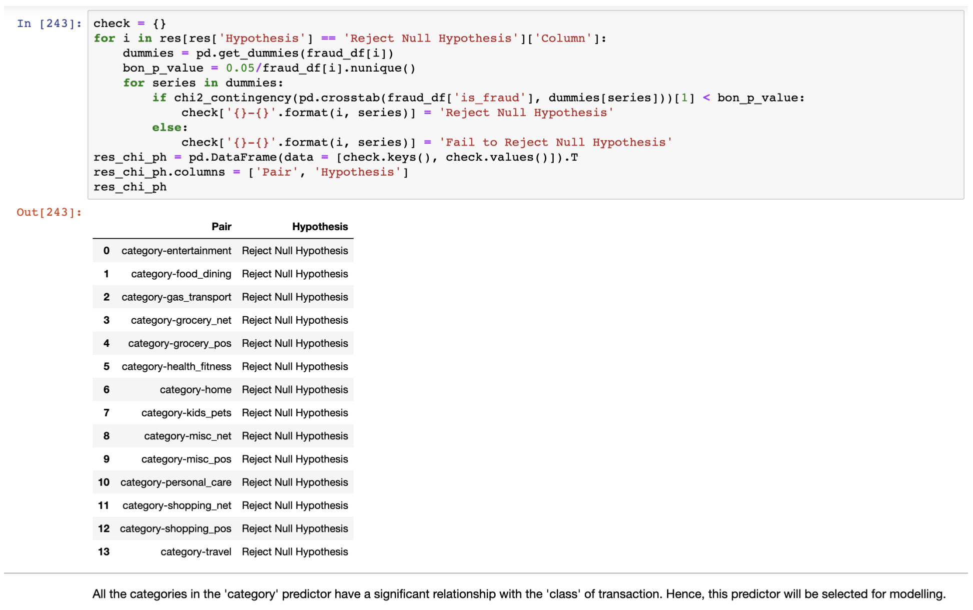


```
categorical_columns = ['category']
```

```
chi2_check = []
for i in categorical_columns:
    if chi2_contingency(pd.crosstab(fraud_df['is_fraud'], fraud_df[i]))[1] < 0.05:
        chi2_check.append('Reject Null Hypothesis')
    else:
        chi2_check.append('Fail to Reject Null Hypothesis')
res = pd.DataFrame(data = [categorical_columns, chi2_check]
             ).T
res.columns = ['Column', 'Hypothesis']
print(res)
```

```
     Column               Hypothesis
0  category  Reject Null Hypothesis
```

Figure 29: Bonferroni Correction Post-Hoc Test



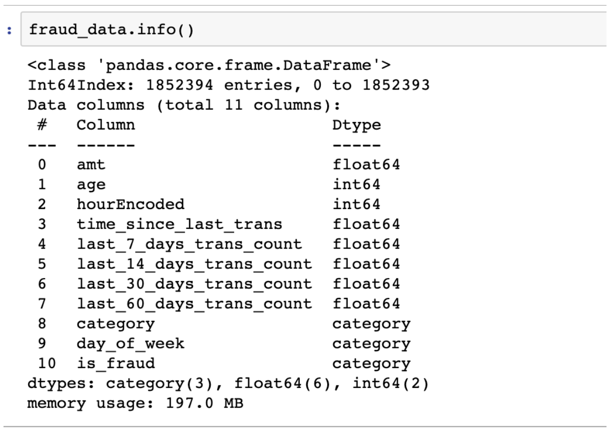
```
In [243]: check = {}
          for i in res[res['Hypothesis'] == 'Reject Null Hypothesis']['Column']:
              dummies = pd.get_dummies(fraud_df[i])
              bon_p_value = 0.05/fraud_df[i].nunique()
              for series in dummies:
                  if chi2_contingency(pd.crosstab(fraud_df['is_fraud'], dummies[series]))[1] < bon_p_value:
                      check['{}-{}'.format(i, series)] = 'Reject Null Hypothesis'
                  else:
                      check['{}-{}'.format(i, series)] = 'Fail to Reject Null Hypothesis'
          res_chi_ph = pd.DataFrame(data = [check.keys(), check.values()]).T
          res_chi_ph.columns = ['Pair', 'Hypothesis']
          res_chi_ph
```

Out[243]:

	Pair	Hypothesis
0	category-entertainment	Reject Null Hypothesis
1	category-food_dining	Reject Null Hypothesis
2	category-gas_transport	Reject Null Hypothesis
3	category-grocery_net	Reject Null Hypothesis
4	category-grocery_pos	Reject Null Hypothesis
5	category-health_fitness	Reject Null Hypothesis
6	category-home	Reject Null Hypothesis
7	category-kids_pets	Reject Null Hypothesis
8	category-misc_net	Reject Null Hypothesis
9	category-misc_pos	Reject Null Hypothesis
10	category-personal_care	Reject Null Hypothesis
11	category-shopping_net	Reject Null Hypothesis
12	category-shopping_pos	Reject Null Hypothesis
13	category-travel	Reject Null Hypothesis

All the categories in the 'category' predictor have a significant relationship with the 'class' of transaction. Hence, this predictor will be selected for modelling.

Figure 30: Bonferroni Correction Post-Hoc Test (continuation)



```
: fraud_data.info()

<class 'pandas.core.frame.DataFrame'>
Int64Index: 1852394 entries, 0 to 1852393
Data columns (total 11 columns):
 #   Column                   Dtype   
---  ------                   -----   
 0   amt                      float64 
 1   age                      int64   
 2   hourEncoded              int64   
 3   time_since_last_trans    float64 
 4   last_7_days_trans_count  float64 
 5   last_14_days_trans_count float64 
 6   last_30_days_trans_count float64 
 7   last_60_days_trans_count float64 
 8   category                 category
 9   day_of_week              category
 10  is_fraud                 category
dtypes: category(3), float64(6), int64(2)
memory usage: 197.0 MB
```

Figure 31: Selected Features for Modelling

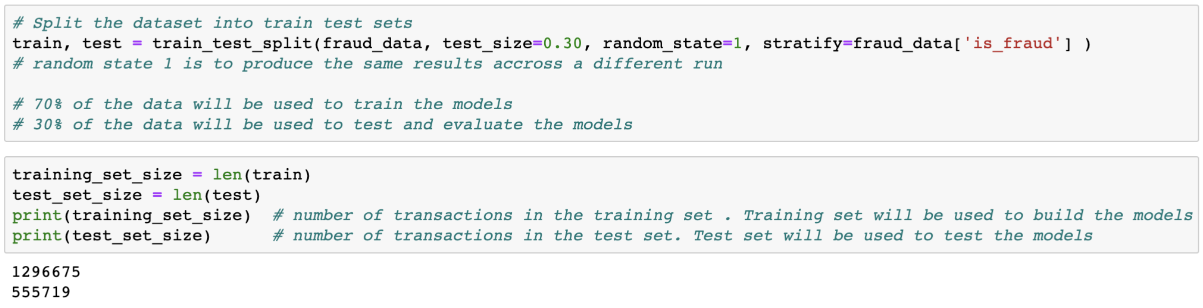
11 Data Preparation for Modelling

In order to prepare the data for modelling , the following tasks were performed using the Sklearn and Imbalanced-learn packages:

- Split the dataset into Train and Test set
- Categorical Encoding
- Balance the Train set

11.1 Split the Dataset into Train and Test Set

The dataset was split in the 70:30 ratio; that is, 70% train and 30% test, using the `train_test_split` function from sklearn. Because the dataset is extremely imbalanced, stratified splitting was applied in order to preserve the class proportions observed in the original dataset. See Figure 32



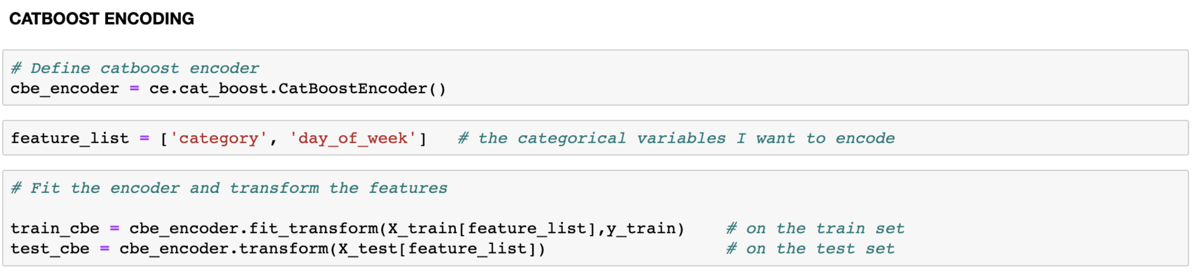
```
# Split the dataset into train test sets
train, test = train_test_split(fraud_data, test_size=0.30, random_state=1, stratify=fraud_data['is_fraud'] )
# random state 1 is to produce the same results accross a different run

# 70% of the data will be used to train the models
# 30% of the data will be used to test and evaluate the models
```

```
training_set_size = len(train)
test_set_size = len(test)
print(training_set_size)  # number of transactions in the training set . Training set will be used to build the models
print(test_set_size)      # number of transactions in the test set. Test set will be used to test the models

1296675
555719
```

Figure 32: Train Test Split

11.2 Categorical Encoding

Since most machine learning algorithms do not take categorical variables as input, categorical variables need to be encoded before creating the models. Also, in order to apply sampling techniques on the training set to balance the class distribution, all the variables need to be converted to numerical. The dataset used for modelling contains two categorical variables : ’category’ and ’day-of-week’. To convert them to numerical, CatBoost encoder was used. The reason why CatBoost encoder was the chosen technique is because it avoids target leakage and hence prevents the risk of overfitting and poor generalisation of the model. CatBoost encoding needs to be performed separately on the train and test set. See Figure 33



CATBOOST ENCODING

```
# Define catboost encoder
cbe_encoder = ce.cat_boost.CatBoostEncoder()
```

```
feature_list = ['category', 'day_of_week']   # the categorical variables I want to encode
```

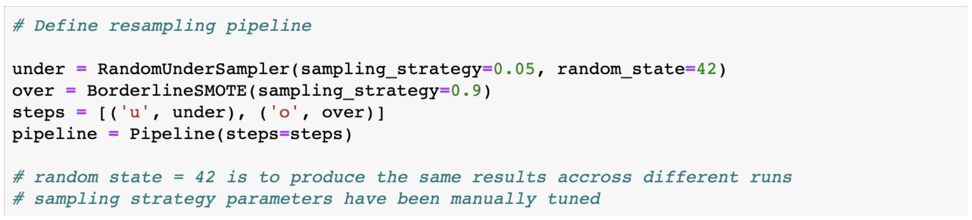
```
# Fit the encoder and transform the features

train_cbe = cbe_encoder.fit_transform(X_train[feature_list],y_train)    # on the train set
test_cbe = cbe_encoder.transform(X_test[feature_list])                  # on the test set
```

Figure 33: CatBoost Encoder

11.3 Dealing with the Class Imbalance : Balance the Train set

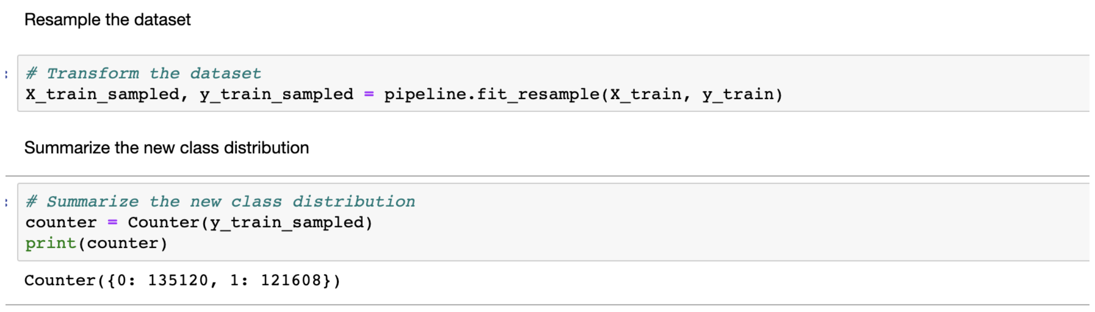
Since the dataset is highly imbalanced, in order to avoid bias towards the majority class in the models, the class imbalance was reduced on the training set before building the classifiers. The imblearn package was used. Only the training data was balanced, as it is used to build the models. The test data does not have to be balanced, as the purpose of the test data is to simulate the results of the model in a ’real-world’ setting, and in real life all the credit card fraud datasets are highly imbalanced.

A hybrid sampling approach of RUS (Random Undersampling) and Borderline-SMOTE was used to reduce the class imbalance on the train set. The large size of the train set (1,296,675 observations) would make training the models computationally expensive. In order to reduce the training time of the models, the train set was first undersampled (RUS) and then oversampled (Borderline SMOTE). The sampling parameters were manually adjusted in a way that the majority class was reduced to 20 times the size of the minority class (sampling strategy= 0.05), and the minority class was 90% of the size of the majority class (sampling strategy = 0.9). See Figure 34, Figure 35 and Figure 36.



```
# Define resampling pipeline

under = RandomUnderSampler(sampling_strategy=0.05, random_state=42)
over = BorderlineSMOTE(sampling_strategy=0.9)
steps = [('u', under), ('o', over)]
pipeline = Pipeline(steps=steps)

# random state = 42 is to produce the same results accross different runs
# sampling strategy parameters have been manually tuned
```

Figure 34: Define the Resampling Pipeline


Resample the dataset

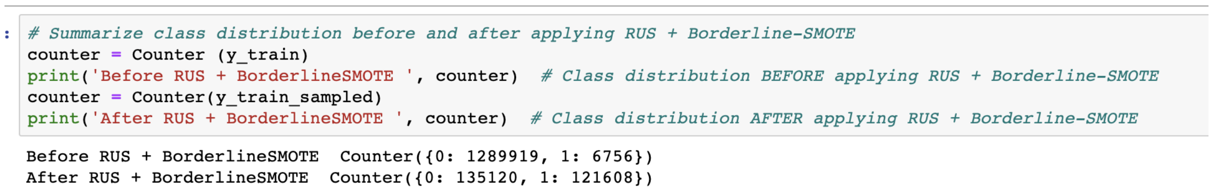
```
# Transform the dataset
X_train_sampled, y_train_sampled = pipeline.fit_resample(X_train, y_train)
```

Summarize the new class distribution

```
# Summarize the new class distribution
counter = Counter(y_train_sampled)
print(counter)
```

```
Counter({0: 135120, 1: 121608})
```

Figure 35: Resample the Train Set

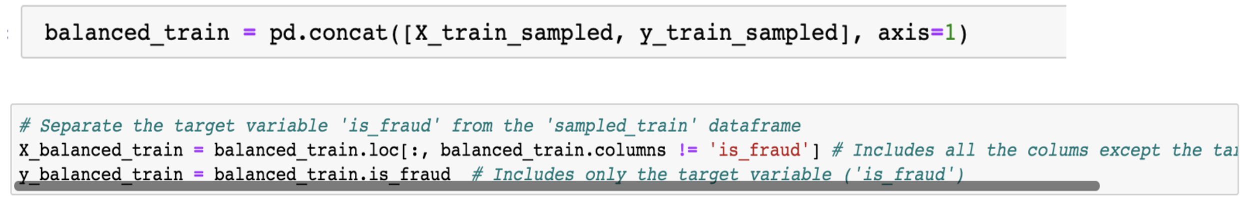

```
# Summarize class distribution before and after applying RUS + Borderline-SMOTE
counter = Counter (y_train)
print('Before RUS + BorderlineSMOTE ', counter)  # Class distribution BEFORE applying RUS + Borderline-SMOTE
counter = Counter(y_train_sampled)
print('After RUS + BorderlineSMOTE ', counter)  # Class distribution AFTER applying RUS + Borderline-SMOTE
```

```
Before RUS + BorderlineSMOTE  Counter({0: 1289919, 1: 6756})
After RUS + BorderlineSMOTE  Counter({0: 135120, 1: 121608})
```

Figure 36: Class Distribution of the Train Set Before and After Resampling

12 Model Implementation

The models (classifiers) were implemented on the train set shown in Figure 37. The Sklearn package was used to train the different models.

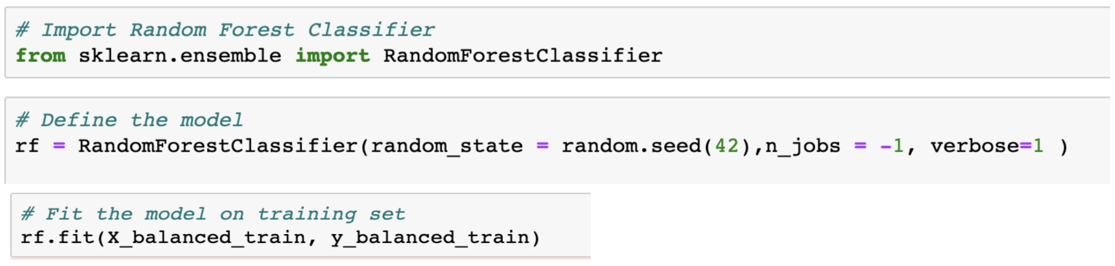

```
balanced_train = pd.concat([X_train_sampled, y_train_sampled], axis=1)
```

```
# Separate the target variable 'is_fraud' from the 'sampled_train' dataframe
X_balanced_train = balanced_train.loc[:, balanced_train.columns != 'is_fraud'] # Includes all the colums except the ta
y_balanced_train = balanced_train.is_fraud  # Includes only the target variable ('is_fraud')
```

Figure 37: Balanced Train Set

12.1 Random Forest

The RF model was first computed with the default hyperparameters as seen in Figure 38. Randomized search 3-fold cross validation (CV) was then computed to get the optimised parameters using the function RandomizedSearchCV() as seen in Figure 39 and Figure 40. It was found that the RF with default hyperparameters achieved a better performance than the RF with optimised hyperparameters. Hence, the RF was built with the default hyperparameters.

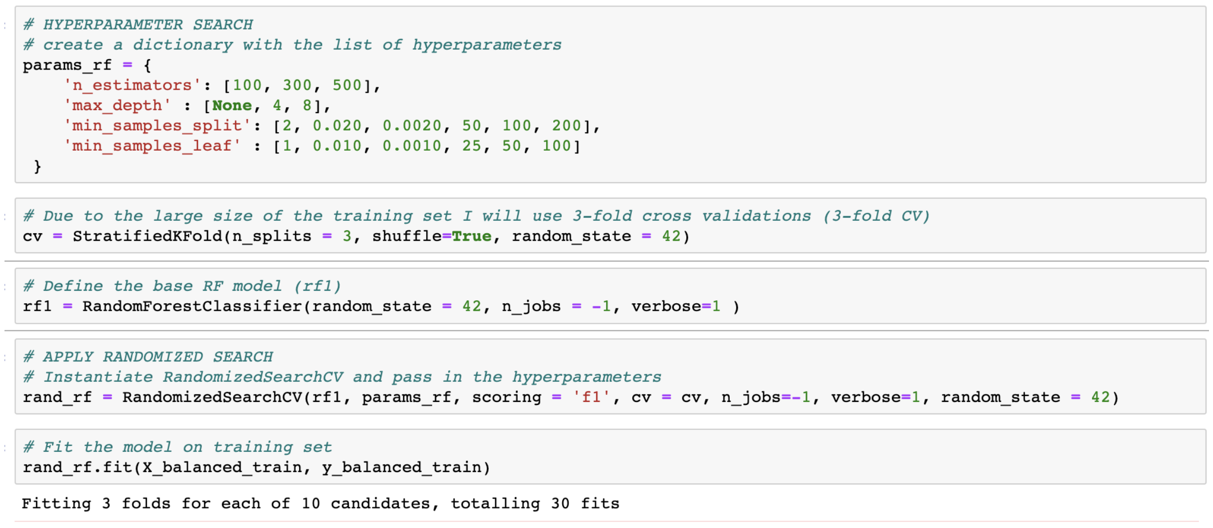


```
# Import Random Forest Classifier
from sklearn.ensemble import RandomForestClassifier
```

```
# Define the model
rf = RandomForestClassifier(random_state = random.seed(42),n_jobs = -1, verbose=1 )
```

```
# Fit the model on training set
rf.fit(X_balanced_train, y_balanced_train)
```

Figure 38: RF Model with Default Hyperparameters



```
# HYPERPARAMETER SEARCH
# create a dictionary with the list of hyperparameters
params_rf = {
    'n_estimators': [100, 300, 500],
    'max_depth' : [None, 4, 8],
    'min_samples_split': [2, 0.020, 0.0020, 50, 100, 200],
    'min_samples_leaf' : [1, 0.010, 0.0010, 25, 50, 100]
 }
```

```
# Due to the large size of the training set I will use 3-fold cross validations (3-fold CV)
cv = StratifiedKFold(n_splits = 3, shuffle=True, random_state = 42)
```

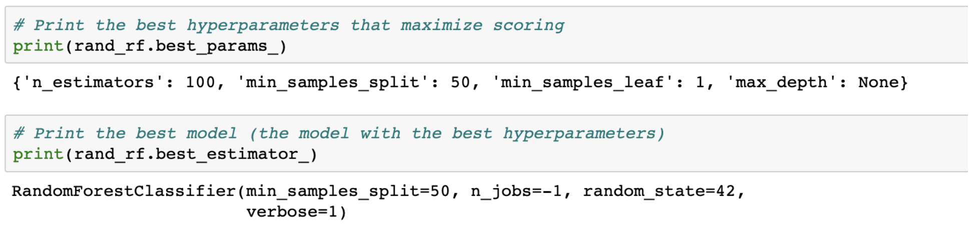
```
# Define the base RF model (rf1)
rf1 = RandomForestClassifier(random_state = 42, n_jobs = -1, verbose=1 )
```

```
# APPLY RANDOMIZED SEARCH
# Instantiate RandomizedSearchCV and pass in the hyperparameters
rand_rf = RandomizedSearchCV(rf1, params_rf, scoring = 'f1', cv = cv, n_jobs=-1, verbose=1, random_state = 42)
```

```
# Fit the model on training set
rand_rf.fit(X_balanced_train, y_balanced_train)
```

```
Fitting 3 folds for each of 10 candidates, totalling 30 fits
```

Figure 39: RF Hyperparameter Optimisation using RandomizedSearchCV



```
# Print the best hyperparameters that maximize scoring
print(rand_rf.best_params_)
```

```
{'n_estimators': 100, 'min_samples_split': 50, 'min_samples_leaf': 1, 'max_depth': None}
```

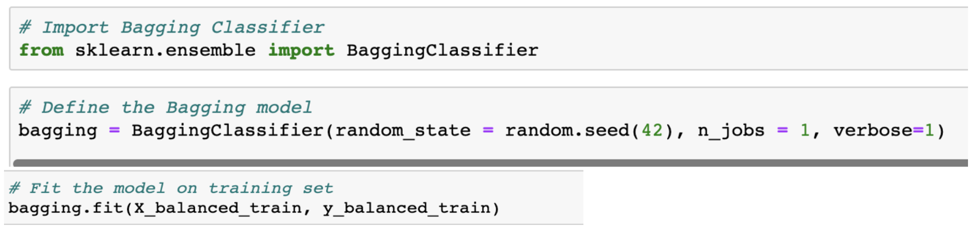
```
# Print the best model (the model with the best hyperparameters)
print(rand_rf.best_estimator_)
```

```
RandomForestClassifier(min_samples_split=50, n_jobs=-1, random_state=42,
                       verbose=1)
```

Figure 40: Optimised Hyperparameters for RF

12.2 Bagging

The Bagging model was first computed with the default hyperparameters as seen in Figure 41. Randomized search 3-fold cross validation (CV) was then computed to get the optimised parameters using RandomizedSearchCV() as seen in Figure 42. The optimised Bagging model outperformed the model with default hyperparameters; hence, the Bagging model was built with the optimised hyperparameters as seen in Figure 43.

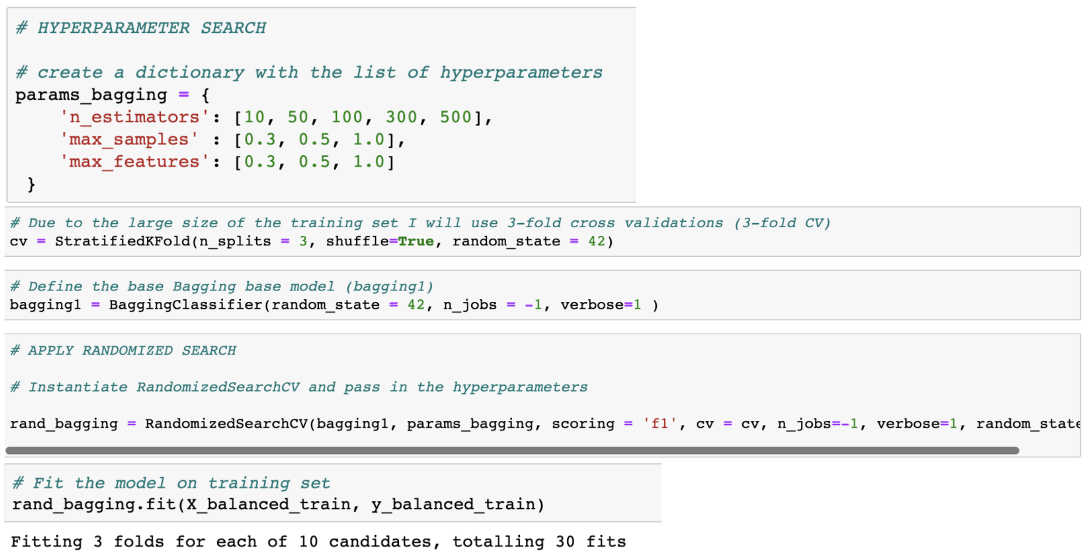


```
# Import Bagging Classifier
from sklearn.ensemble import BaggingClassifier
```

```
# Define the Bagging model
bagging = BaggingClassifier(random_state = random.seed(42), n_jobs = 1, verbose=1)
```

```
# Fit the model on training set
bagging.fit(X_balanced_train, y_balanced_train)
```

Figure 41: Bagging Model with Default Hyperparameters



```
# HYPERPARAMETER SEARCH

# create a dictionary with the list of hyperparameters
params_bagging = {
    'n_estimators': [10, 50, 100, 300, 500],
    'max_samples' : [0.3, 0.5, 1.0],
    'max_features': [0.3, 0.5, 1.0]
 }
```

```
# Due to the large size of the training set I will use 3-fold cross validations (3-fold CV)
cv = StratifiedKFold(n_splits = 3, shuffle=True, random_state = 42)
```

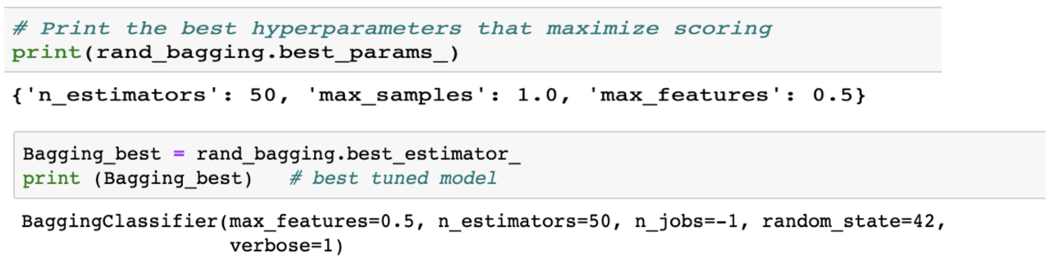
```
# Define the base Bagging base model (bagging1)
bagging1 = BaggingClassifier(random_state = 42, n_jobs = -1, verbose=1 )
```

```
# APPLY RANDOMIZED SEARCH

# Instantiate RandomizedSearchCV and pass in the hyperparameters

rand_bagging = RandomizedSearchCV(bagging1, params_bagging, scoring = 'f1', cv = cv, n_jobs=-1, verbose=1, random_state
```

```
# Fit the model on training set
rand_bagging.fit(X_balanced_train, y_balanced_train)
```

```
Fitting 3 folds for each of 10 candidates, totalling 30 fits
```

Figure 42: Bagging Hyperparameter Optimisation using RandomizedSearchCV



```
# Print the best hyperparameters that maximize scoring
print(rand_bagging.best_params_)
```

```
{'n_estimators': 50, 'max_samples': 1.0, 'max_features': 0.5}
```

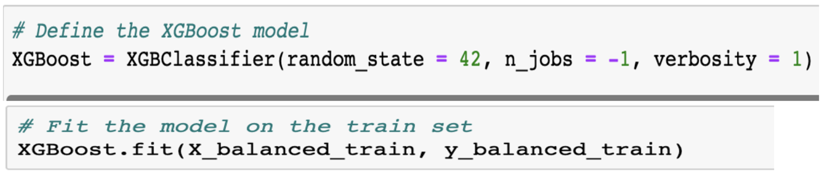
```
Bagging_best = rand_bagging.best_estimator_
print (Bagging_best)    # best tuned model
```

```
BaggingClassifier(max_features=0.5, n_estimators=50, n_jobs=-1, random_state=42,
                  verbose=1)
```

Figure 43: Optimised Hyperparameters for Bagging

12.3 XGBoost

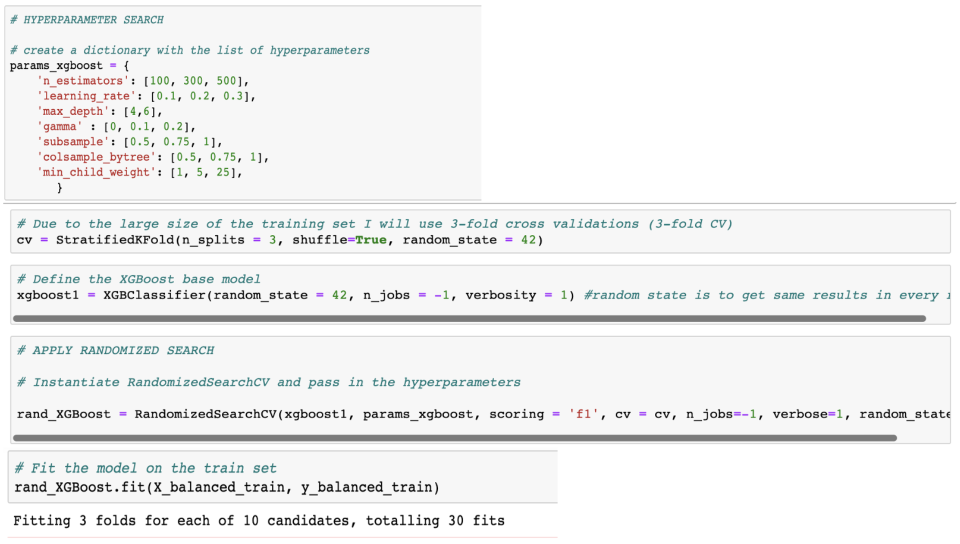
The XGBoost model was first computed with the default hyperparameters as seen in Figure 44. Randomized search 3-fold cross validation (CV) was then computed to get the optimised parameters using RandomizedSearchCV() as seen in Figure 45. The optimised XGBoost model outperformed the model with default hyperparameters; hence, the XGBoost model was built with the optimised hyperparameters as seen in Figure 46.



```
# Define the XGBoost model
XGBoost = XGBClassifier(random_state = 42, n_jobs = -1, verbosity = 1)
```

```
# Fit the model on the train set
XGBoost.fit(X_balanced_train, y_balanced_train)
```

Figure 44: XGBoost Model with Default Hyperparameters



```
# HYPERPARAMETER SEARCH

# create a dictionary with the list of hyperparameters
params_xgboost = {
    'n_estimators': [100, 300, 500],
    'learning_rate': [0.1, 0.2, 0.3],
    'max_depth': [4,6],
    'gamma' : [0, 0.1, 0.2],
    'subsample': [0.5, 0.75, 1],
    'colsample_bytree': [0.5, 0.75, 1],
    'min_child_weight': [1, 5, 25],
      }
```

```
# Due to the large size of the training set I will use 3-fold cross validations (3-fold CV)
cv = StratifiedKFold(n_splits = 3, shuffle=True, random_state = 42)
```

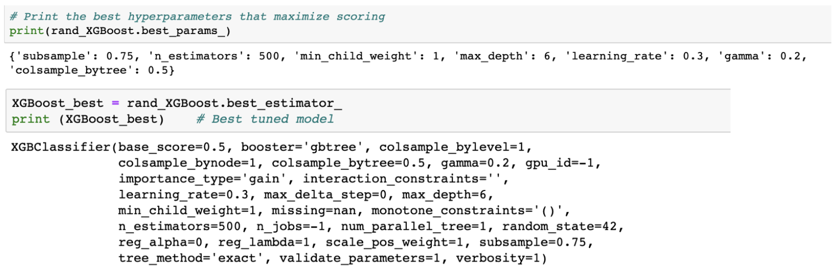
```
# Define the XGBoost base model
xgboost1 = XGBClassifier(random_state = 42, n_jobs = -1, verbosity = 1) #random state is to get same results in every
```

```
# APPLY RANDOMIZED SEARCH

# Instantiate RandomizedSearchCV and pass in the hyperparameters

rand_XGBoost = RandomizedSearchCV(xgboost1, params_xgboost, scoring = 'f1', cv = cv, n_jobs=-1, verbose=1, random_state
```

```
# Fit the model on the train set
rand_XGBoost.fit(X_balanced_train, y_balanced_train)
```

```
Fitting 3 folds for each of 10 candidates, totalling 30 fits
```

Figure 45: XGBoost Hyperparameter Optimisation using RandomizedSearchCV



```
# Print the best hyperparameters that maximize scoring
print(rand_XGBoost.best_params_)
```

```
{'subsample': 0.75, 'n_estimators': 500, 'min_child_weight': 1, 'max_depth': 6, 'learning_rate': 0.3, 'gamma': 0.2,
'colsample_bytree': 0.5}
```

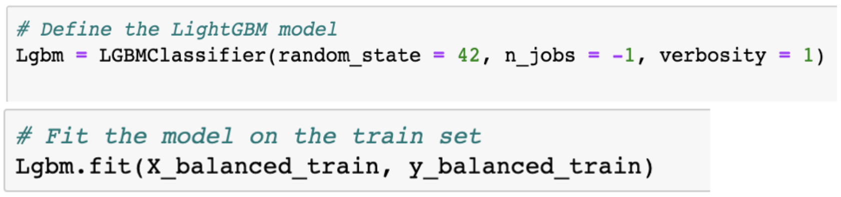
```
XGBoost_best = rand_XGBoost.best_estimator_
print (XGBoost_best)    # Best tuned model
```

```
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=0.5, gamma=0.2, gpu_id=-1,
              importance_type='gain', interaction_constraints='',
              learning_rate=0.3, max_delta_step=0, max_depth=6,
              min_child_weight=1, missing=nan, monotone_constraints='()',
              n_estimators=500, n_jobs=-1, num_parallel_tree=1, random_state=42,
              reg_alpha=0, reg_lambda=1, scale_pos_weight=1, subsample=0.75,
              tree_method='exact', validate_parameters=1, verbosity=1)
```

Figure 46: Optimised Hyperparameters for XGBoost

12.4 LightGBM

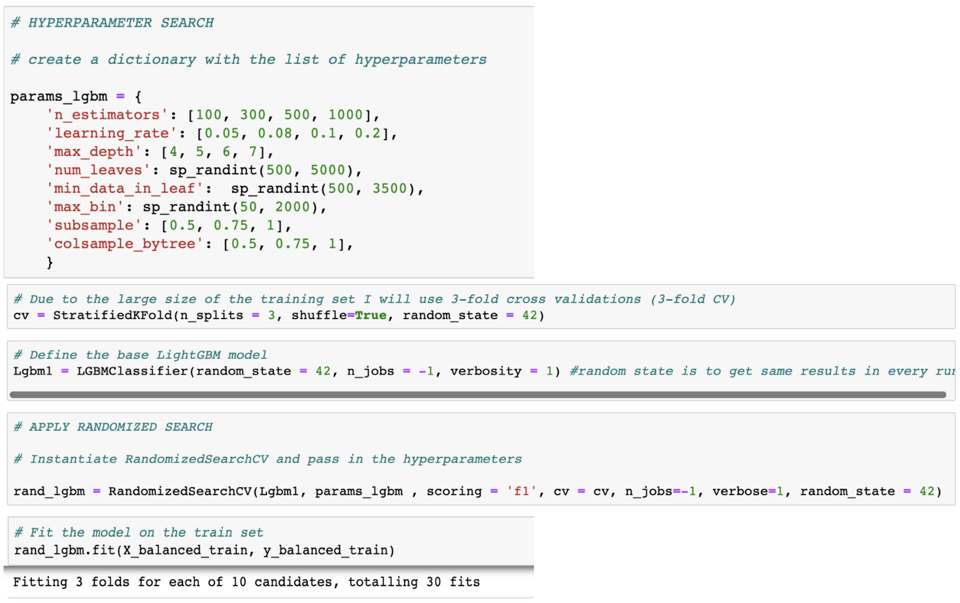
The LightGBM model was first computed with the default hyperparameters as seen in Figure 47. Randomized search 3-fold cross validation (CV) was then computed to get the optimised parameters using RandomizedSearchCV() as seen in Figure 48. The optimised LightGBM model outperformed the model with default hyperparameters; hence, the LightGBM model was built with the optimised hyperparameters as seen in Figure 49.



```
# Define the LightGBM model
Lgbm = LGBMClassifier(random_state = 42, n_jobs = -1, verbosity = 1)
```

```
# Fit the model on the train set
Lgbm.fit(X_balanced_train, y_balanced_train)
```

Figure 47: LightGBM Model with Default Hyperparameters



```
# HYPERPARAMETER SEARCH

# create a dictionary with the list of hyperparameters

params_lgbm = {
    'n_estimators': [100, 300, 500, 1000],
    'learning_rate': [0.05, 0.08, 0.1, 0.2],
    'max_depth': [4, 5, 6, 7],
    'num_leaves': sp_randint(500, 5000),
    'min_data_in_leaf':  sp_randint(500, 3500),
    'max_bin': sp_randint(50, 2000),
    'subsample': [0.5, 0.75, 1],
    'colsample_bytree': [0.5, 0.75, 1],
    }
```

```
# Due to the large size of the training set I will use 3-fold cross validations (3-fold CV)
cv = StratifiedKFold(n_splits = 3, shuffle=True, random_state = 42)
```

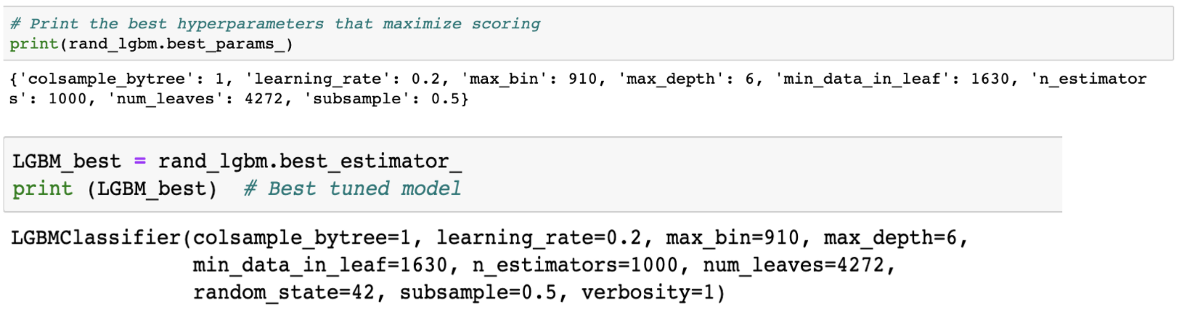
```
# Define the base LightGBM model
Lgbm1 = LGBMClassifier(random_state = 42, n_jobs = -1, verbosity = 1) #random state is to get same results in every ru
```

```
# APPLY RANDOMIZED SEARCH

# Instantiate RandomizedSearchCV and pass in the hyperparameters

rand_lgbm = RandomizedSearchCV(Lgbm1, params_lgbm , scoring = 'f1', cv = cv, n_jobs=-1, verbose=1, random_state = 42)
```

```
# Fit the model on the train set
rand_lgbm.fit(X_balanced_train, y_balanced_train)
```

```
Fitting 3 folds for each of 10 candidates, totalling 30 fits
```

Figure 48: LightGBM Hyperparameter Optimisation using RandomizedSearchCV



```
# Print the best hyperparameters that maximize scoring
print(rand_lgbm.best_params_)
```

```
{'colsample_bytree': 1, 'learning_rate': 0.2, 'max_bin': 910, 'max_depth': 6, 'min_data_in_leaf': 1630, 'n_estimator
s': 1000, 'num_leaves': 4272, 'subsample': 0.5}
```

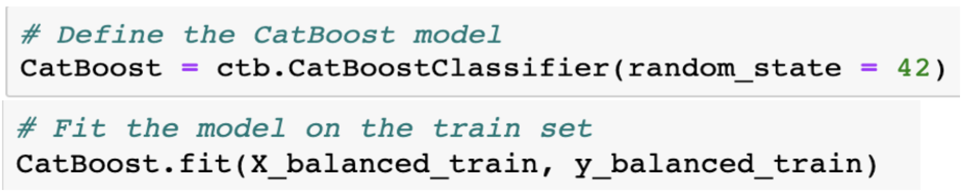
```
LGBM_best = rand_lgbm.best_estimator_
print (LGBM_best)  # Best tuned model
```

```
LGBMClassifier(colsample_bytree=1, learning_rate=0.2, max_bin=910, max_depth=6,
               min_data_in_leaf=1630, n_estimators=1000, num_leaves=4272,
               random_state=42, subsample=0.5, verbosity=1)
```

Figure 49: Optimised Hyperparameters for LightGBM

12.5 CatBoost

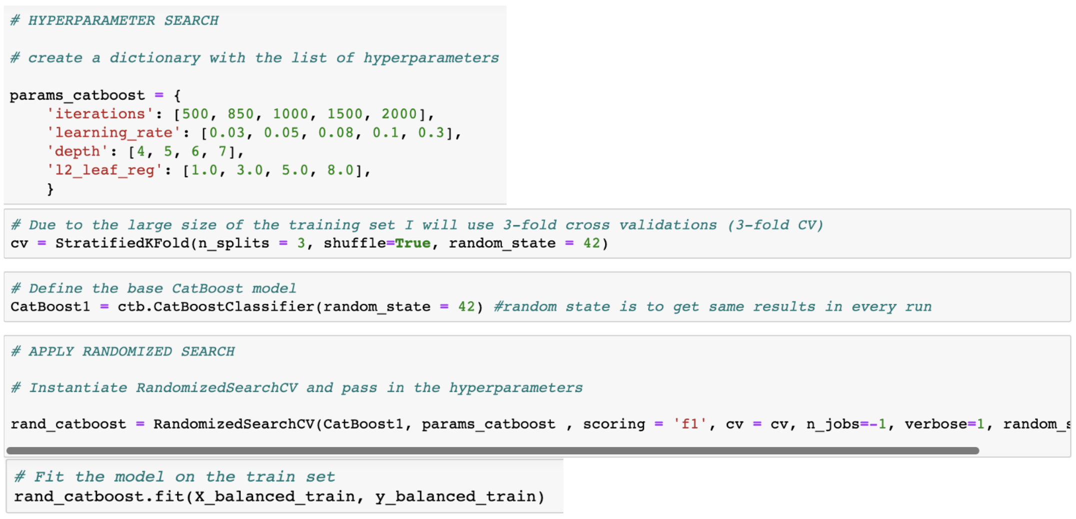
The CatBoost model was first computed with the default hyperparameters as seen in Figure 50. Randomized search 3-fold cross validation (CV) was then computed to get the optimised parameters using RandomizedSearchCV() as seen in Figure 51. The optimised CatBoost model outperformed the model with default hyperparameters; hence, the CatBoost model was built with the optimised hyperparameters as seen in Figure 52.



```
# Define the CatBoost model
CatBoost = ctb.CatBoostClassifier(random_state = 42)
```

```
# Fit the model on the train set
CatBoost.fit(X_balanced_train, y_balanced_train)
```

Figure 50: CatBoost Model with Default Hyperparameters



```
# HYPERPARAMETER SEARCH

# create a dictionary with the list of hyperparameters

params_catboost = {
    'iterations': [500, 850, 1000, 1500, 2000],
    'learning_rate': [0.03, 0.05, 0.08, 0.1, 0.3],
    'depth': [4, 5, 6, 7],
    'l2_leaf_reg': [1.0, 3.0, 5.0, 8.0],
    }
```

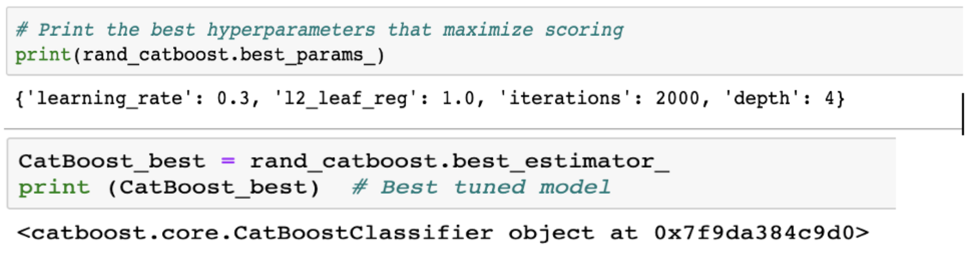
```
# Due to the large size of the training set I will use 3-fold cross validations (3-fold CV)
cv = StratifiedKFold(n_splits = 3, shuffle=True, random_state = 42)
```

```
# Define the base CatBoost model
CatBoost1 = ctb.CatBoostClassifier(random_state = 42) #random state is to get same results in every run
```

```
# APPLY RANDOMIZED SEARCH

# Instantiate RandomizedSearchCV and pass in the hyperparameters

rand_catboost = RandomizedSearchCV(CatBoost1, params_catboost , scoring = 'f1', cv = cv, n_jobs=-1, verbose=1, random_s
```

```
# Fit the model on the train set
rand_catboost.fit(X_balanced_train, y_balanced_train)
```

Figure 51: CatBoost Hyperparameter Optimisation using RandomizedSearchCV



```
# Print the best hyperparameters that maximize scoring
print(rand_catboost.best_params_)
```

```
{'learning_rate': 0.3, 'l2_leaf_reg': 1.0, 'iterations': 2000, 'depth': 4}
```

```
CatBoost_best = rand_catboost.best_estimator_
print (CatBoost_best)  # Best tuned model
```

```
<catboost.core.CatBoostClassifier object at 0x7f9da384c9d0>
```

Figure 52: Optimised Hyperparameters for CatBoost

13 Model Evaluation

The models were evaluated on the test set. Figure 53 shows how the predictions were made by testing the LightGBM model on the test data. Same code can be reused for the different models. The evaluation metrics that were used to evaluate each classifier are : Recall, Precision, F1-Score, Matthews Correlation Coefficient (MCC), G-Mean and Area Under the Precision-Recall Curve (AUC -PR); and they were computed using the Sklearn and Imbalanced-learn packages.

This section provides the code used to evaluate the LightGBM model as a sample code, and same code can be reused for the rest of models. Figure 54 shows the sample

MAKE PREDICTIONS USING THE BEST TUNED MODEL ON THE TEST DATA

```
# Make predictions with 'LGBM_best' using the test data  (predict the labels using the test set)
y_LGBM_best_pred = LGBM_best.predict(X_test)
LGBM_best_predictions = [round(value) for value in y_LGBM_best_pred]
```

Figure 53: Predictions of the LightGBM model on the test data

code for the Classification Report. Figure 55 shows the Confusion Matrix. Figure 56 shows the AUC. Figure 57 shows the Precision-Recall curve.

```
# Evaluate the predictions of the 'LightGBM_best' classifier
print("Classification Report of the Tuned LightGBM classifier: \n", classification_report(y_test, y_LGBM_best_pred))
```

```
Classification Report of the Tuned LightGBM classifier:
               precision    recall  f1-score   support

           0       1.00      1.00      1.00    552824
           1       0.57      0.89      0.70      2895

    accuracy                           1.00    555719
   macro avg       0.79      0.94      0.85    555719
weighted avg       1.00      1.00      1.00    555719
```

Figure 54: Classification Report

```
plot_confusion_matrix(LGBM_best, X_test, y_test)
```

```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x7f9e30a0c0d0>
```

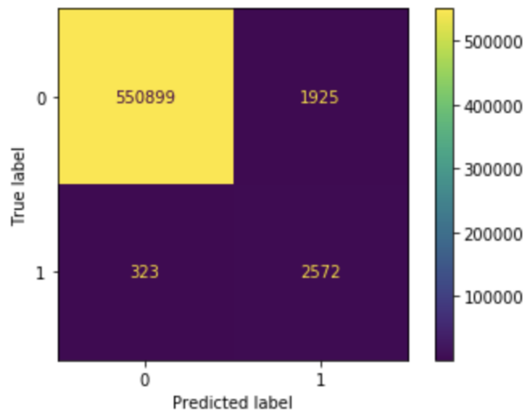
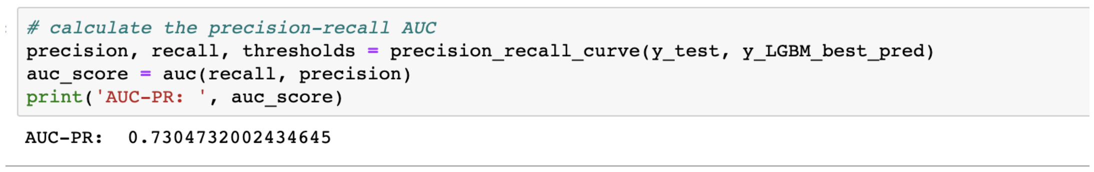


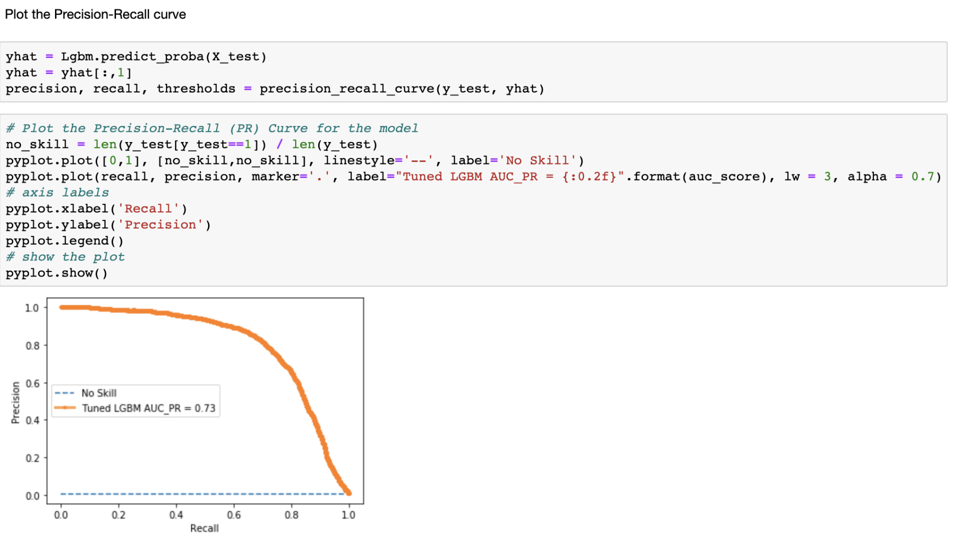
Figure 55: Confusion Matrix



```
# calculate the precision-recall AUC
precision, recall, thresholds = precision_recall_curve(y_test, y_LGBM_best_pred)
auc_score = auc(recall, precision)
print('AUC-PR: ', auc_score)
```

```
AUC-PR:  0.7304732002434645
```

Figure 56: AUC



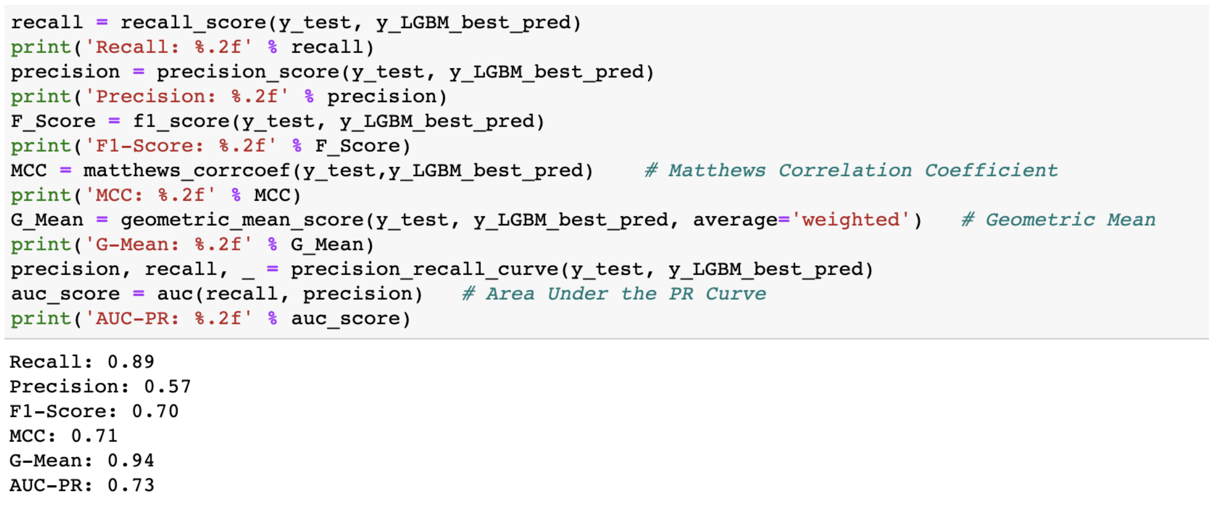
Plot the Precision-Recall curve

```
yhat = Lgbm.predict_proba(X_test)
yhat = yhat[:,1]
precision, recall, thresholds = precision_recall_curve(y_test, yhat)
```

```
# Plot the Precision-Recall (PR) Curve for the model
no_skill = len(y_test[y_test==1]) / len(y_test)
pyplot.plot([0,1], [no_skill,no_skill], linestyle='--', label='No Skill')
pyplot.plot(recall, precision, marker='.', label="Tuned LGBM AUC_PR = {:0.2f}".format(auc_score), lw = 3, alpha = 0.7)
# axis labels
pyplot.xlabel('Recall')
pyplot.ylabel('Precision')
pyplot.legend()
# show the plot
pyplot.show()
```

Figure 57: AUC-PR curve

Figure 58 shows the code used to compute the metrics for evaluating the performance of the classifiers. Figure 59 shows the code used to compute the most important credit card fraud predictors for the classifiers. The sample code provided is for the LightGBM model but same code can be reused for the different models.



```
recall = recall_score(y_test, y_LGBM_best_pred)
print('Recall: %.2f' % recall)
precision = precision_score(y_test, y_LGBM_best_pred)
print('Precision: %.2f' % precision)
F_Score = f1_score(y_test, y_LGBM_best_pred)
print('F1-Score: %.2f' % F_Score)
MCC = matthews_corrcoef(y_test,y_LGBM_best_pred)    # Matthews Correlation Coefficient
print('MCC: %.2f' % MCC)
G_Mean = geometric_mean_score(y_test, y_LGBM_best_pred, average='weighted')   # Geometric Mean
print('G-Mean: %.2f' % G_Mean)
precision, recall, _ = precision_recall_curve(y_test, y_LGBM_best_pred)
auc_score = auc(recall, precision)   # Area Under the PR Curve
print('AUC-PR: %.2f' % auc_score)
```

```
Recall: 0.89
Precision: 0.57
F1-Score: 0.70
MCC: 0.71
G-Mean: 0.94
AUC-PR: 0.73
```

Figure 58: Evaluation Metrics

```
importances_lgbm = LGBM_best.feature_importances_
```

```
# Sort the feature importance in descending order
sorted_indices = np.argsort(importances_lgbm)[::-1]
```

```
plt.title('Tuned LightGBM Feature Importance')
plt.bar(range(X_train_sampled.shape[1]), importances_lgbm[sorted_indices], align='center')
plt.xticks(range(X_train_sampled.shape[1]), X_train_sampled.columns[sorted_indices], rotation=90)
plt.tight_layout()
plt.show()
```

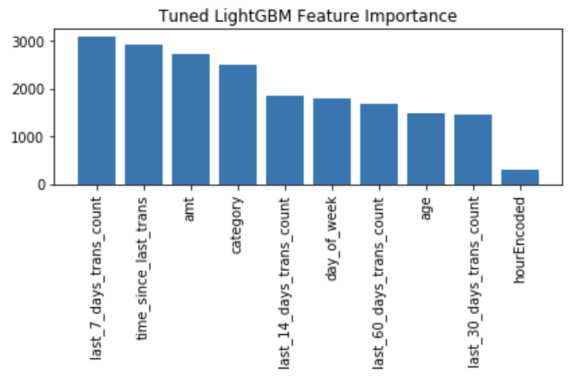


Figure 59: Variable Importance

14 Conclusion

This report outlines the different steps involved in the 'Credit Card Fraud Detection using Ensemble Learning Algorithms' research project. Python was the software used to analyse the data and to create the models. Tableau was also used to create some visualisations; however, this document only focuses on the analysis and modelling aspects of the project. It was found that LightGBM and XGBoost were the best performing models followed by CatBoost; hence, it was concluded that boosting algorithms outperformed bagging algorithms in credit card fraud detection. It was also found that the volume of transactions over the last 7 days, the time since last transaction, the time of the day, the transaction amount and category are the most important variables for predicting credit card fraud.