Team: BigDataAnalysts

Members: Bhuvan Jayam(IMT2020506), Mukul Gupta(IMT2020083)

**It’s-a-Fraud**

Project Report

horizontal line

# 

# Introduction

Given details about some online transactions. We need to classify whether the given transaction is fraudulent or not. Basically, we are predicting whether an online transaction is fraudulent or not.

Given the training dataset, we have some features that will help us predict the nature of the transaction. First, we will analyze the data and try to fit a model that will predict whether the transaction is fraudulent or not.

## Scenario

Assume a corporation has been providing financial services to a wide range of people in a country. But, at the same time, security methods have flaws. Cybercriminals may attempt to break into the company's interface and commit fraudulent transactions. While there are numerous techniques to avoid fraud and loss from occurring. Let's have a look at a simple solution to the problem of fraudulent transactions by identifying and automating a mode for future transactions.

The collected data set's goal characteristic is **isFraud** over customer transactions.

| **Target Feature** | **Output Values** |
| --- | --- |
| *isFraud* | 0: Not Fraud |
|  | 1: Fraud |

### Importing required libraries

# Importing necessary modules

import numpy as np # linear algebra

import pandas as pd # data processing, CSV file I/O (e.g. pd.read\_csv)

import seaborn as sns # plotting

from sklearn.utils.random import sample\_without\_replacement

from pandas.api.types import is\_numeric\_dtype

# For undersampling and oversampling data

from imblearn.pipeline import Pipeline

from imblearn.over\_sampling import RandomOverSampler, RandomUnderSampler

# For selecting the best parameters for the model

from sklearn.model\_selection import RandomizedSearchCV, GridSearchCV

# For Data Model Development

from sklearn import linear\_model

import xgboost

from sklearn.naive\_bayes import GaussianNB

from sklearn.neural\_network import MLPClassifier

from sklearn.tree import DecisionTreeClassifier

# For Neural Network Model

from keras.models import Sequential

from keras.layers import Dense

from keras import optimizers

from sklearn.preprocessing import StandardScaler

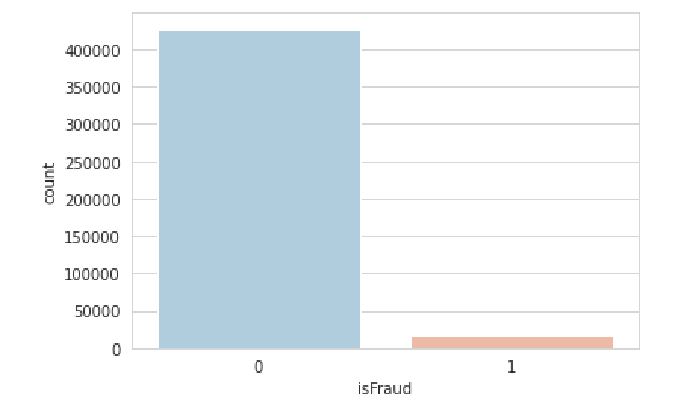
from sklearn.metrics import accuracy\_score

## Features Description of Dataset

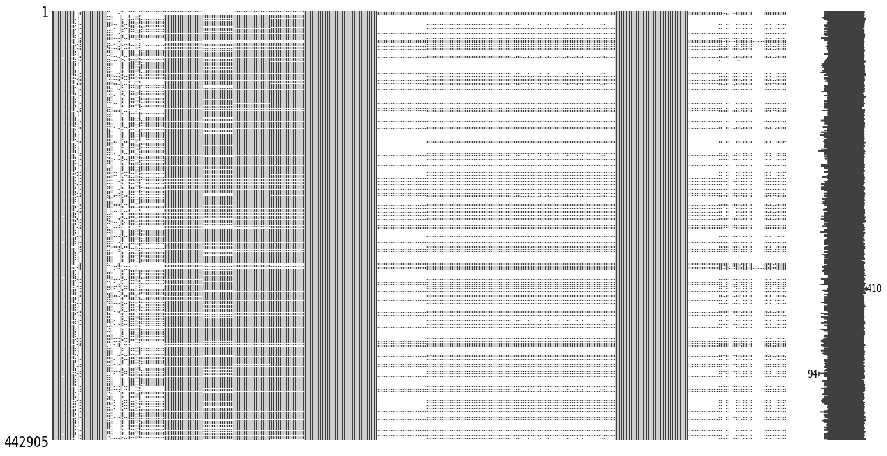
The train dataset contains 442905 records of transactions, each with 434 features.

| **Features** | **Description** |
| --- | --- |
| *TransactionDT* | Time difference between a particular reference date and time |
| *TransactionAMT* | Transaction payment amount in USD |
| *ProductCD* | Product code, product for each transaction |
| *card 1-6* | Payment card information:   * Card type * Card category * Issue bank * Country |
| *addr 1-2* | addr1 - billing region, addr2 - billing country |
| *dist* | Distances between billing and postal addresses, zip codes, IP addresses, phone areas, and so on |
| *{}\_emaildomain* | P : purchaser email domain  R : recipient email domain |
| *C 1-14* | counting, such as how many addresses are discovered to be linked to the payment card, and so on |
| *D 1-15* | time difference, such as the number of days between previous transactions, and so on. |
| *M 1-9* | match, such as names on cards and addresses, etc. |
| *Vxxx* | Vesta created numerous features, such as ranking, counting, and other entity relations. |
| *id 01-11* | numerical features for identity. |

## Analyzing Dataset



As we can see, the number of fraudulent transactions in the train dataset is quite low. Because the data we have is so biased, we would need higher-order models to learn the dataset.



There are also a large number of null values in both the test and training datasets. This is simply a graph displaying null values in the train dataset. There are a total of **115523073** null values in both datasets combined.

Luckily, there are no duplicates in the dataset. So, we can skip the part about deleting duplicates.

In the preprocessing, we need to do four important steps in order to stabilize the data before training a model.

* Handling null values.
* Outlier Removal
* One-hot encoding
* Undersampling and oversampling of data

## -> Data Preprocessing

We will concatenate both the test and train datasets before beginning any of the preprocessing procedures, as the model will be trained on the preprocessed train dataset. To keep the same preprocessing steps, we join both datasets before doing anything.

Y\_train = train.pop('isFraud')

dataframes\_to\_be\_combined = [train,test]

df = pd.concat(dataframes\_to\_be\_combined)

Now, we can proceed with the preprocessing steps on the dataframe ‘df’.

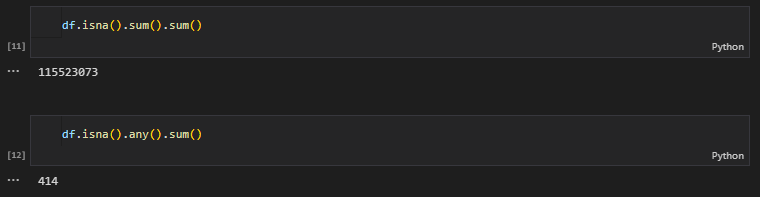
Let's look at all the features that are available in the dataset.



1. **Handling null values**

There are a total of 424 features, including the output here.

Now, let's check for missing values:



There are a total of 414 columns with a total of 115523073 null values in the dataset.

We need to balance the data by filling in the null values for each and every column.

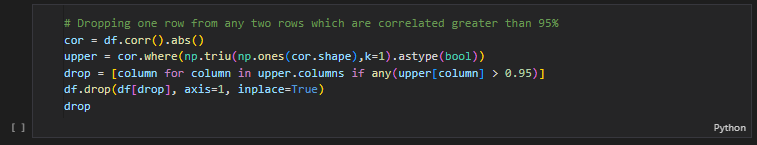


There is a feature called "DeviceInfo" here. As can be seen, it is a feature that contains categorical data. It is very varied and contains almost 50% null values due to the large number of unique values. So, in order to alleviate the imbalance created by this feature and save calculation costs, we will remove it from the dataset.

-> We cannot verify each feature one by one and delete null values in this manner. So we'll go through all of the typical preprocessing procedures and remove any null values.

**1.1) Removing highly correlated features**

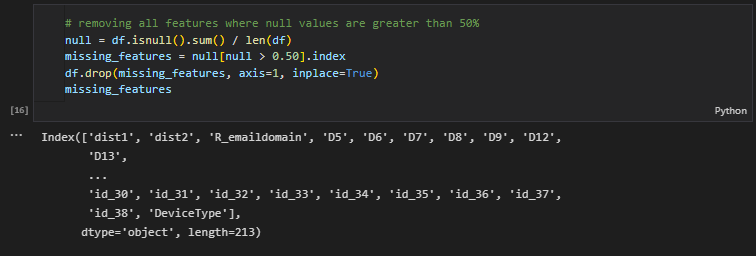
Highly correlated features are those that are slightly linearly dependent on other features. These characteristics contribute very little to output prediction but increase computing expenses. So, we will find the correlation matrix and remove one of the features that is more than 95% correlated to other features.



Here, we will get rid of all the features that are highly correlated with one of the other features, like here.

**1.2) Removing features with more than 50% null values**

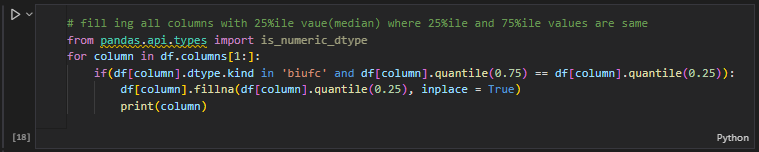
There are numerous features with more than 50% null values. So, all these features don't contribute well to the output prediction due to the low sampling. So we will drop all such features.



df.isnull() -> gives all the columns with at least one null value in them.

**1.3) Replacing null values with 25% when 25% == 75%**

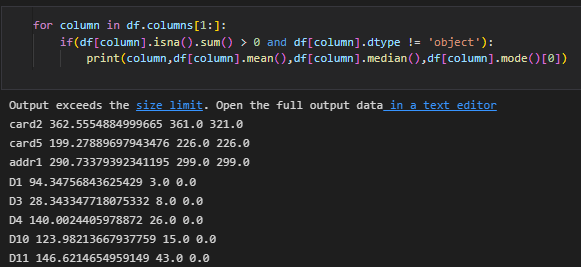
When the data in a column is highly biased or tilted towards a single value, we can replace all the missing data with the median itself. So, to check that, we will check if the 25% value of a feature is equal to the 75% value of that feature, and then we will replace all the null values with the same value itself.

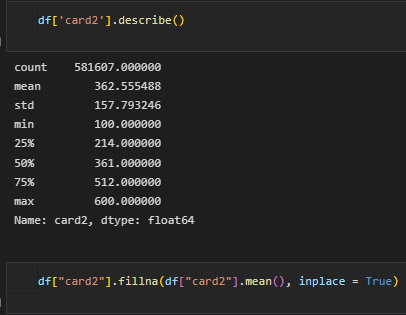
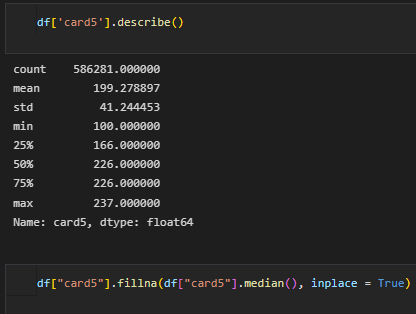


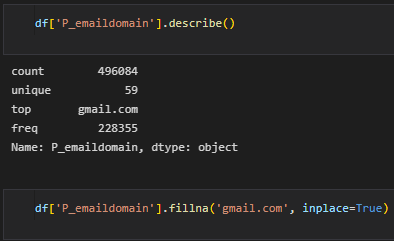
Here, we are not checking for the characteristic data type features, as they wouldn’t have the percentile values due to no encoding at the moment.

**1.4) Replacing null values of few important features by checking its descriptions**

After determining which columns still have null values, we would describe each important feature and select the mean, median, or mode based on its data distribution.

As we can see, the only distinguishing features among the remaining ones are "card2", "card5", "addr1" and "P\_emaildomain" (bottom), and because they contribute sufficiently to output, we will examine them individually.





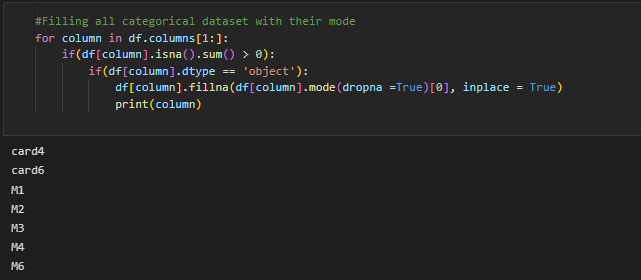
-> The “card2” feature’s null values are replaced by the mean() value, as the data in it is very diversified throughout the percentiles and has a higher standard deviation.

-> The “P\_emaildomain” feature contains “gmail.com” as its value for more than half of its data. So, we can replace all the null values in this feature by its mode() value, which is “gmail.com”.

-> In all the remaining features, nearly half the data contains similar values or the mean() value is very near the standard deviation value, so their null values are replaced by the median value.

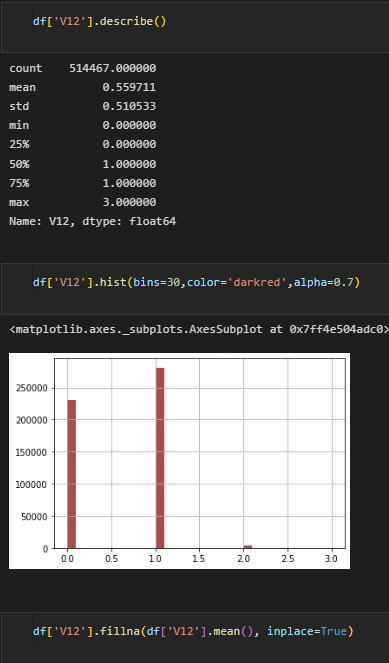
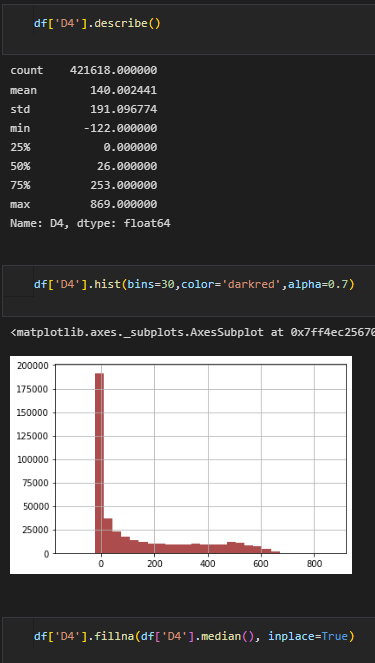
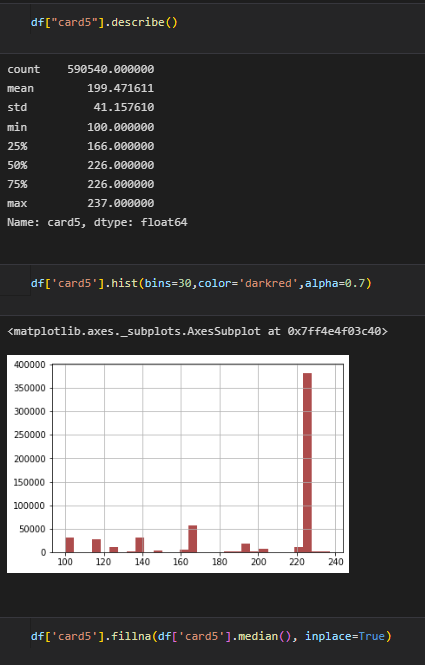
**1.5) Replacing null values of categorical features with their mode()**

As we cannot calculate mean() and median() values for categorical features, the simple method to replace null values would be to replace them with their mode() value.

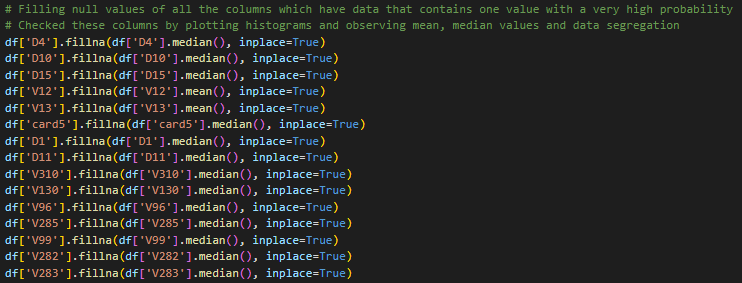


**1.6) Replacing null values of few features by observing their plots**

In all the features that are remaining and have null values, we will check those with higher means and standard deviations and replace null values with means, medians, or modes based on their plots.

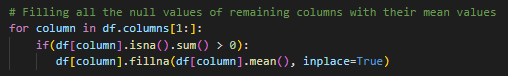


Similarly, by observing all the plots, these are the features for which we replaced null values with means or medians.



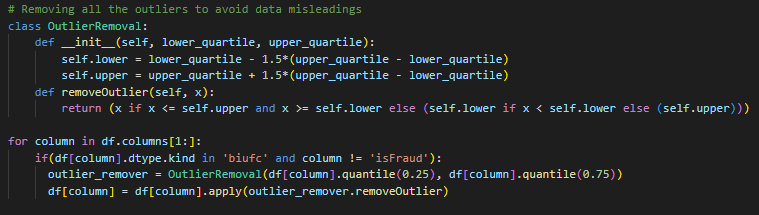
**1.7) Replacing remaining features null values with their mean()**

All the remaining features have mean(), median(), and mode() values between 0 and 1, and also contain only 0 and 1 as data. So, in order to maintain the area curve, we will just replace all the null values with the mean in order to maintain stability.



1. **Outlier Removal**

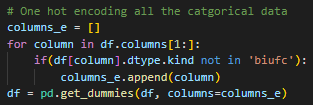
An outlier is any piece of data that is unusually far apart from the other points in the data. In order to stabilize the data, we will remove the outliers. There are different methods to remove outliers. One such method is by comparing the quartile values of the given data.



For all the non-categorical data, we will calculate the upper and lower thresholds based on the 25% and 75% values mentioned above. The data then gets replaced by the values after being compared to thresholds. If the value is less than the lower threshold, it will be replaced by a lower threshold. If the data is greater than the upper threshold, it will be replaced by the upper threshold. Otherwise, it remains the same.

1. **One-Hot Encoding**

One-hot encoding is used to replace all the categorical data with binary values, as the models don’t support categorical data. It replaces any column of categorical data with multiple columns, each for a category, with binary values 0 or 1, with a particular category having only a value of 1.

The get\_dummies() function creates a column for each category in that column.

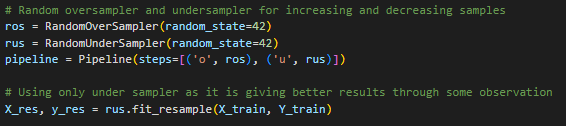
Before performing undersampling or oversampling, we must divide the test and training datasets, as we will need to resample the output Y values.



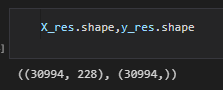
1. **Undersampling and oversampling of data**

Oversampling is utilized to produce outcomes with a more equal proportion of positive results in training when one class of data is the underrepresented minority class in the data sample. If a data class is overrepresented by the majority, undersampling may be used to balance it with the minority class. When the amount of data collected is sufficient, undersampling is used.

There is also a method called pipeline, which is used when we need to both undersample and oversample data to make the data more diversified and a bit easier for the model to train.



After some thorough observation, we came to the conclusion that undersampled data produces better results than oversampled and pipelined data after training. So, we only used undersampling here. And also, the model takes less time to train with undersampled data than the others.



These are the shapes of X\_train and Y\_train after undersampling.

## -> Classification Models

Now that we are done with the preprocessing of the dataset, we can start training the models with the data we have.

The main aspect we need to keep in mind while selecting a model is selecting appropriate parameters for the model. We need to tune the parameters for the best results.

Models can have many hyperparameters, and finding the best combination of parameters can be treated as a search problem. The two best strategies for hyperparameter tuning are:

* GridSearchCV
* RandomizedSearchCV

GridSearchCV is the basic search method, which searches for the best hyperparameters from a grid of hyperparameter values. However, it will go through all of the intermediate hyperparameter choices, increasing the cost.

RandomizedSearchCV only runs through a given number of hyperparameter settings. It moves randomly throughout the grid to discover the optimal collection of hyperparameters. This method eliminates unnecessary computation.

There are different classification models that can be used for this problem. We used some of the best models that we are able to implement and also did hyperparameter tuning and used the best parameters for better results.

Model we implemented:

* Logistic Regression
* Naive Bayes
* Perceptron
* Decision Tree
* Neural Networks
* XGBoost

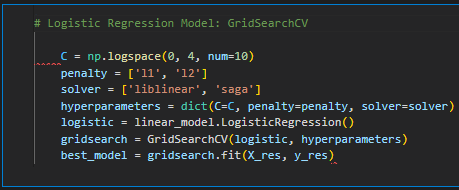
1. **Logistic Regression**

Logistic regression is a classification algorithm that uses supervised learning to predict the likelihood of a target variable. Logistic regression employs logit functions, which aid in determining a link between the dependent variable and the independent variables by forecasting the probabilities or chances of occurrence. Logistic regression does not really have any critical hyperparameters to tune.

Hyperparameters:

* *C*: Inverse of regularization strength
* *Solver*: The algorithm that will be employed in the optimization task
* *penalty*: l1, l2 or elastic net regularization/penalty

We can either use GridSearchCV or RandomizedSearchCV to find the best parameters here. But in general, using RandomizedSearchCV produces better results and is faster than the other method. We used the best hyperparameters here.

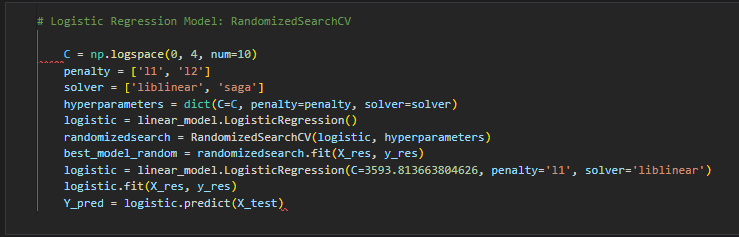


Best params:

*C* = 3593.81366

*Penalty* = “l1”

*Solver* = "liblinear"



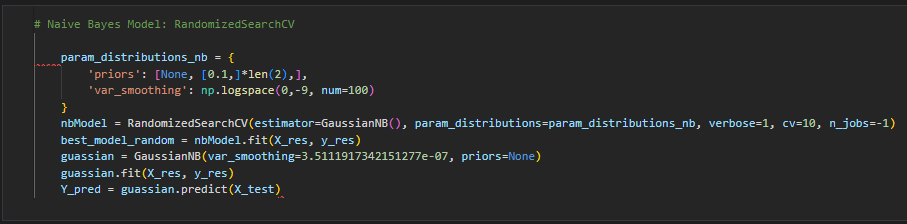
1. **Naive Bayes classifier**

The term "Naive Bayes classifiers" refers to a set of classification methods based on Bayes' Theorem. It is a family of algorithms that all share a common principle, namely, that every pair of features being classified is independent of each other.

There are very few hyperparameters to tune for the Guassian Naive Bayes model.

Hyperparameters:

* *priors*: It accepts arrays of shape (n classes,) indicating target class prior probabilities.
* *var\_smoothing*: accepts a float that specifies the percentage of the biggest variance of all features that is applied to variances for smoothing.



Here, we implemented RandomizedSearchCV as we got better results using the parameters from this tuning method.

Best parameters:

*Var\_smoothing*: 3.5111

*Priors*: None

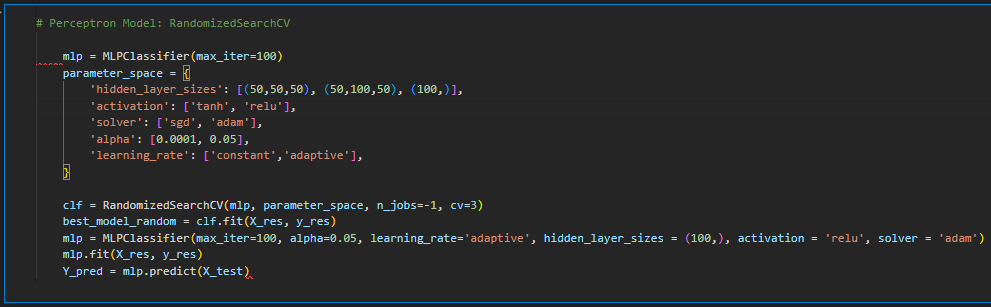
1. **Perceptron Classifier**

The perceptron is a binary classification machine learning algorithm. It can swiftly learn a linear separation in feature space for two-class classification tasks, similar to logistic regression, but unlike logistic regression, it learns using the stochastic gradient descent optimization approach and does not predict probabilities.

There are many hyperparameters with wide ranges to tune as it is a neural network based model. So, it would take a little more time to search for the best parameters for the model.

Hyperparameters:

* *hidden\_layer\_sizes*: the ith element represents the number of neurons in the ith hidden layer
* *activation*: activation function for the hidden layer
* *Solver*: solver for weight optimization
* *Alpha*: strength of the L2 regularization term
* *learning\_rate*: learning rate for updating the weights



Best params:

* *alpha*: 0.05
* *learning\_rate*: “adaptive”
* *hidden\_layer\_sizes*: (100,)
* *activation*: “relu”
* *solver*: “adam”

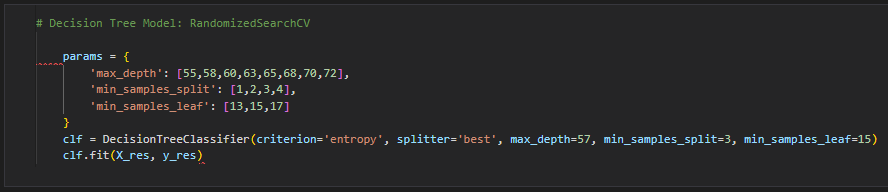
1. **Decision Tree Classifier**

It is a tree-structured classifier in which internal nodes contain dataset attributes, branches represent decision rules, and each leaf node represents the result. A decision tree has two nodes: the decision node and the leaf node. The sequence of features is critical, and not every feature is relevant to every feature. Hence, the decision tree is a good data model to predict. Standardization is not required for this model, and max depth is also limited to avoid overfit.

There are mainly three hyperparameters that change the dynamics of this classifier.

Hyperparameters:

* *max\_depth*: maximum depth of the tree
* *min\_samples\_split*: minimum number of samples required to split an internal node
* *min\_samples\_leaf*: minimum number of samples required to be at a leaf node



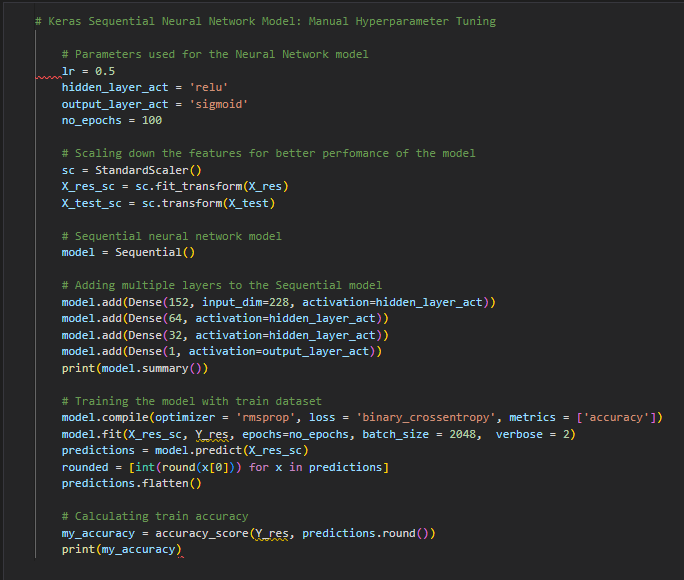
We took the parameter values that are near the default values, which are attained when the model is used without any constraints. We varied them a little bit to attain the best results using this classifier. This is the best model we used before evaluation 1.

Best params:

* *max\_depth*: 57
* *min\_samples\_split*: 3
* *min\_samples\_leaf*: 15

Due to its tree-like structure, it gives some of the best results while classifying, as we can subdivide each class indefinitely in a tree structure. There are some other hyperparameters that can be tuned, like *min\_weight\_fraction\_leaf*, *splitter*, *max\_features*, *random\_state*, etc., for which we may get even better results using this model.

1. **Neural Network**

We implemented neural networks using the Keras API. Its simple foundation simplifies the creation of complicated neural networks. Deep learning is a form of machine learning technique that employs interconnected nodes or neurons in a layered structure that resembles the human brain. We can implement as many layers as we want to increase the accuracy, but the computation may increase due to this.

There are many hyperparameters in neural networks, and manually tuning them is difficult. We utilized the Keras tuner function from its library to tune the parameters. Because it was taking a long time to run, we tested a small set of options and obtained the results. However, if we had computed the tuner function, even if it took time, we might have gotten better results.

We need to standardize the features for this model, as it would give better results while computing the cost function and would not cause overflow.

We used this model for evaluation 3, and the results were satisfactory.

1. **XGBoost Classifier**

The XGBoost classifier is an ML technique for structured and tabular data. XGBoost is a gradient-boosted decision tree solution optimized for speed and performance. It is a combination of many models. It is really fast in computing. The XGBoost model is the most widely used model in all ML competitions around the world right now.

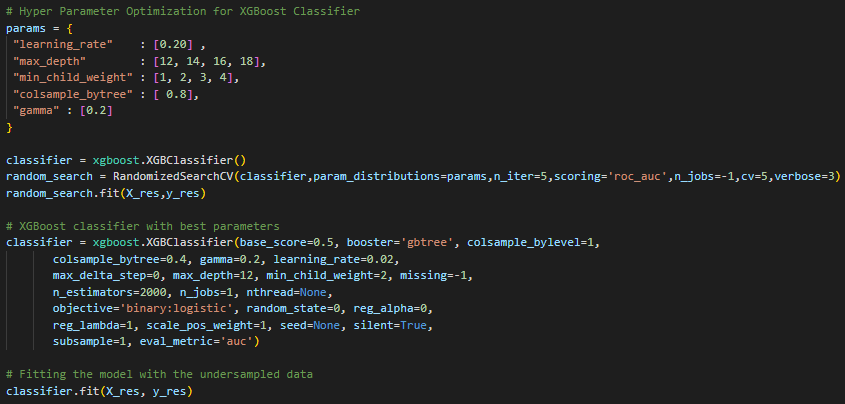
There are three types of parameters that are available in this model. They are:

* General parameters: Direct the overall operation
* Booster parameters: At each stage, direct the specific booster
* Learning Task Parameters: Direct the optimization process

We selected some specific parameters that mainly affect the model's performance to get better results.

Hyperparameters we used:

* *learning\_rate*: learning rate for updating the weights
* *max\_depth*: maximum depth of a tree
* *min\_child\_weight*: minimum sum of weights of all features required in a child
* *colsample\_bytree*: fraction of columns to be randomly samples for each tree
* *gamma:* minimum loss reduction required to make a split



Best params:

* *learning\_rate*: 0.02
* *min\_child\_weight*: 2
* *max\_depth*: 12
* *gamma*: 0.2
* *colsample\_bytree*: 0.4

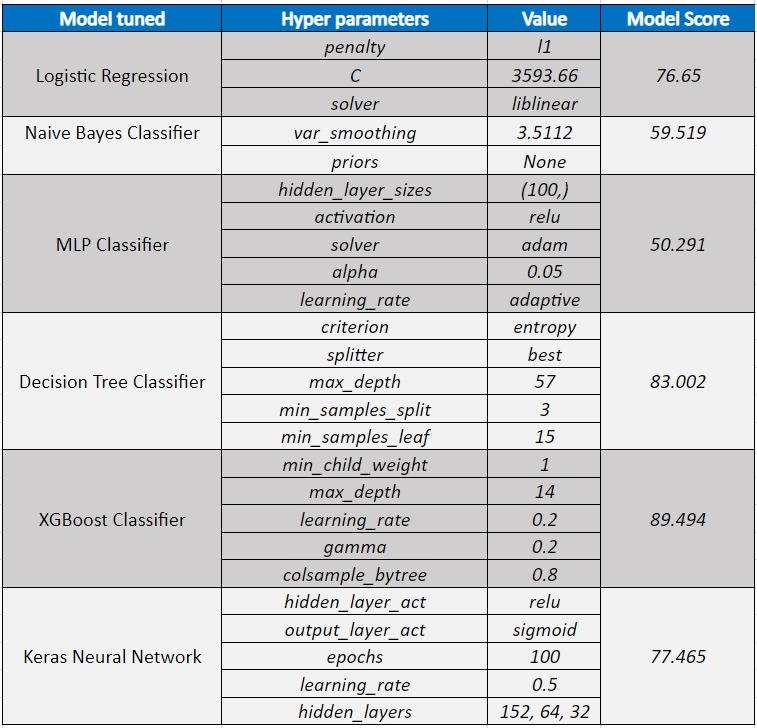
XGBoost's classifier has produced the best results so far. The parameters were occasionally changed due to minor changes in preprocessing. But, overall, XGBoost is faster in computing and gives better results due to the higher-level optimization functions present in it.

This is the best model we implemented before evaluation 2, and it is still our best model with a higher accuracy of nearly 89.5%.

Standardization is not essential for this model.

## -> Final Results

The following table contains the mean roc\_auc scores that we have obtained in the Kaggle competition.



We used all of these different models and tried our hardest to increase the accuracy. We attempted other preparation approaches as well, such as dimensionality reduction, but this way we got better results.