2022

CREDIT CARD FRAUD DETECTION USING MACHINE LEARNING

AIDI CAPSTONE – AIDI 1003-02

FACILITATOR – REZA DIBAJ

TEAM

DEVY RATNASARI (**309)

GOPIKA SHAJI (**568)

OLUWOLE AYODELE (**448)

PRIYANKA SINGH (**824)

SAURAV BISHT (**875)

Contents

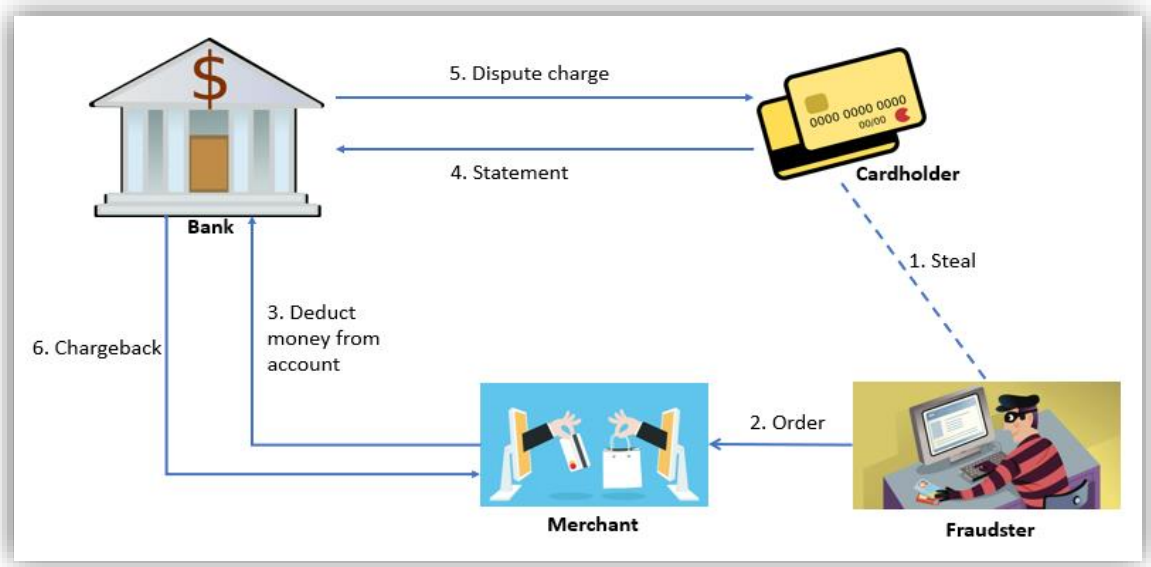
EXECUTIVE SUMMARY

Our objective is to Implement multiple Machine Learning Models on a Dataset that contains information regarding Credit Card Transactions and try to Predict whether a New Transaction is Fraudulent or Not. During this project, we will train these models and tune them to get the best accuracy. We will be evaluating the model using several metrics and will pick the best performing model out of all.

With the new trend of Online Shopping and Online Platforms for Transactions, the number of Credit Card based Transactions increased tremendously. However, there have been a lot of cases where illegal use of Debit/Credit Cards for making Fraudulent Transactions. Credit card companies have been paying a lot of attention to providing the best service for their customers by avoiding such scams occurring in the transactions. Machine Learning Models can work well in detecting such Fraudulent actions when they are trained on a large quantity of historical data.

INTRODUCTION

Credit card fraud is “the unauthorized use of a credit or debit card or similar payment tool to fraudulently obtain money or property.” Credit card fraud detection has become one of the most important aspects in this era of digital payments. Online fraud has widespread business impacts and requires an effective end-to-end strategy to prevent account takeover (ATO), deter new account fraud, and stop suspicious payment transactions. Necessary prevention measures can be taken to stop this abuse and the behaviour of such fraudulent practices can be studied to minimize it and protect against similar occurrences in the future.



Fraud detection involves monitoring the activities of populations of users to estimate, perceive or avoid objectionable behaviour, which consists of fraud, intrusion, and defaulting. In a real-world problem, it is extremely rare to have a balanced dataset to work with, which means that the classification algorithm undermines the importance of the minority class in the dataset in most cases. The minority class is the most significant aspect of the classification process, especially in credit card fraud detection. Due to the unbalanced distribution of the classes in the dataset, the proposed approach highlights the imbalance class issue using various resampling techniques after choosing the best machine learning algorithms.

RATIONALE

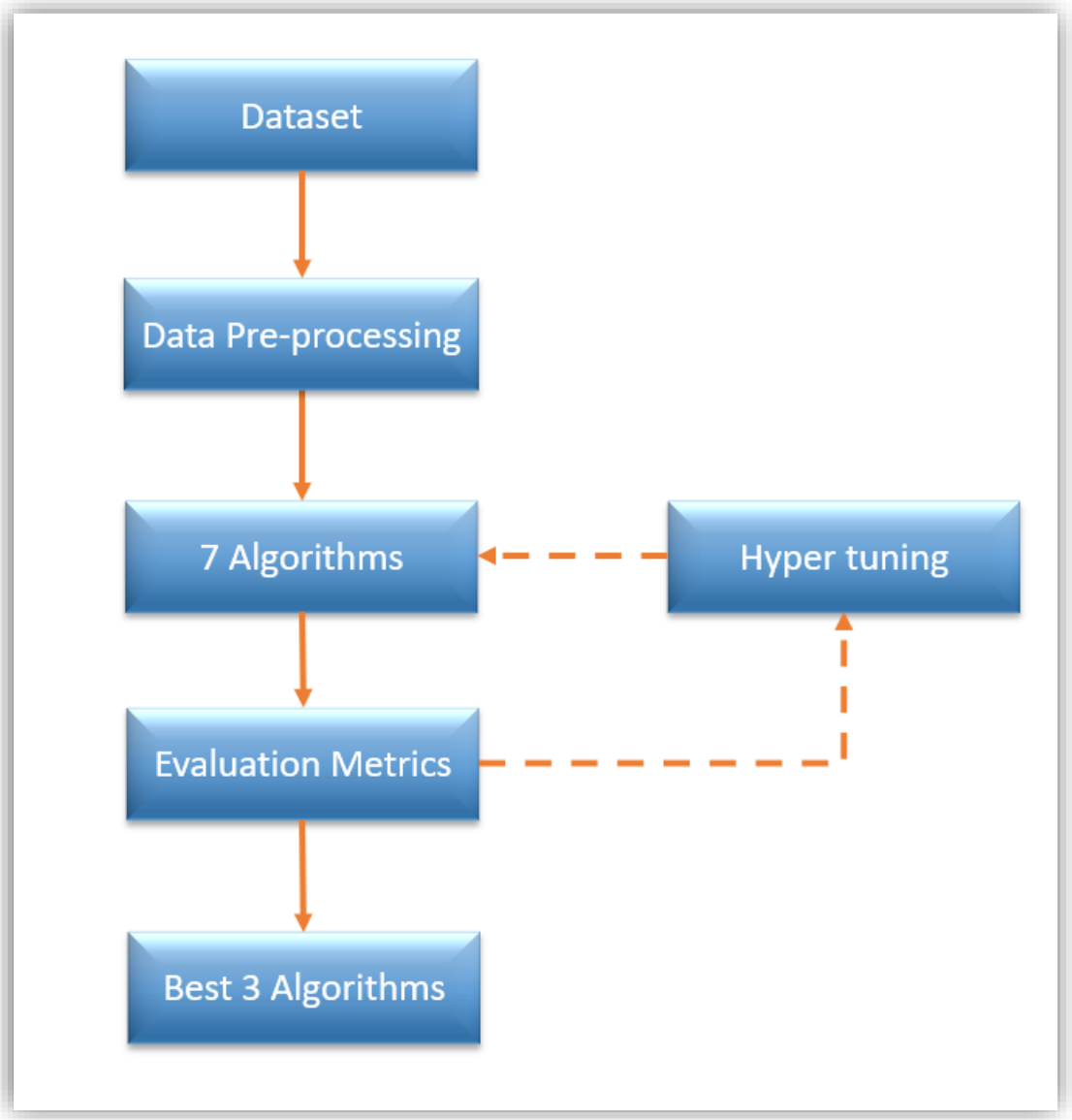
As per TransUnion Fraud Trend Q2 report on Digital Fraud in 2021, the suspected digital fraud attempt rate increased worldwide to 23.82% when comparing the last four months of 2020 to the first four months of 2021. During the first four-month of 2021, suspected fraud increased at an alarming rate of 149.44% with the top fraud type being true identity fraud. The rate of suspected digital fraud attempts against financial services companies increased 217.54% compared to the last four months of 2020 and the first four months of 2021. This shows that the rate of fraud attempts is increasing globally during pandemics with more consumers moving to digital transactions and fraudsters understand this is where the most high-value transactions are taking place. Therefore, financial institutions are implementing fraud prevention solutions.

When it comes to credit card fraud, everyone pays the price. Consumers and the businesses that serve them all suffer from fraudulent activity. And the costs can be staggering. Global financial losses related to payment cards are estimated to reach $34.66 billion in 2022, according to The Nilson Report, a newsletter that tracks the payment industry. Everyone along the payment lifecycle is impacted by a fraudulent transaction—from the consumer who makes purchases in person or online using a credit or debit card to the merchant who finalizes that purchase.

Related to the negative impacts of credit card fraud activities, financial and product losses, it's easy for merchants and users to feel victimized and helpless. Some methods have been used to prevent this from happening, such as Address Verification System, Card Verification Methods, Negative and Positive Lists, Payer Authentication, Lockout Mechanism and Fraudulent Merchants. Many fraud detection techniques help to prevent fraud in the financial industry, such as predictive analytics and data mining, clustering techniques and anomaly detection. However, there are drawbacks to these methods which could be fixed by using the Machine Learning model. The challenges facing credit card fraud detection depend on many factors, such as machine learning algorithms, cross-validation techniques, and resampling techniques. Considering these factors can enhance the performance of the model that can be validated by the evaluation metrics.

PROBLEM STATEMENT

We will use 7 different machine learning algorithms to classify transactions into fraud vs non-fraud. To solve this problem statement, we will be using the following approach:



DATA REQUIREMENT

For this problem, we need a bank transaction dataset containing transaction amount, frequency of transaction, place of transaction, transaction info, location, fraud indicator.

DATASET

The dataset that will be used is from Kaggle.com(https://www.kaggle.com/kartik2112/fraud-detection). This is a simulated credit card transaction dataset containing legitimate and fraud transactions from the duration of 1st Jan 2019 to 31st Dec 2020. It covers credit cards of 1000 customers doing transactions with a pool of 800 merchants.

The dataset has a total of 1852394 observations of which 9651 of them were classified as fraudulent transactions which means the dataset is highly unbalanced. There are 23 columns in the dataset. Also due to privacy issues, we were not able to find the real-world banking dataset.

There were two separate datasets on Kaggle – one for training and another for testing. However, we chose to combine both datasets into one for our problem to complete exploratory data analysis on the entire set and balance percentage before splitting it into train and test datasets.

Below is the snapshot of the code and the datasets:

```
#import the train dataset
dtrain = pd.read_csv('fraudTrain.csv')

#import the test dataset
dtest = pd.read_csv('fraudTest.csv')

#combine 2 datasets into 1
fullset = pd.concat([dtrain, dtest])

#see the structure of data
print(fullset.shape)
print(fullset.info())

#print sample of data(the first 5 rows)
fullset.head()
```

	Unnamed: 0	trans_date_trans_time	cc_num	merchant	category	amt	first	last	gender	street	...	lat	long	city_pop	job	dob	trans_num	unix_time	merch_lat	merch_long	is_fraud
0	0	2019-01-01 00:00:18	2703186189652095	fraud_Rippin, Kub and Mann	misc_net	4.97	Jennifer	Banks	F	561 Perry Cove	...	36.0788	-81.1781	3495	Psychologist, counselling	1988-03-09	0b242abb623afc578575680df30655b9	1325376018	36.011293	-82.048315	0
1	1	2019-01-01 00:00:44	630423337322	fraud_Heller, Gutmann and Zieme	grocery_pos	107.23	Stephanie	Gill	F	43039 Riley Greens Suite 393	...	48.8878	-118.2105	149	Special educational needs teacher	1978-06-21	1f76529f857473494963 61c461b024d99	1325376044	49.159047	-118.186462	0
2	2	2019-01-01 00:00:51	38859492057661	fraud_Lind-Buckridge	entertainment	220.11	Edward	Sanchez	M	594 White Dale Suite 530	...	42.1808	-112.2620	4154	Nature conservation officer	1962-01-19	a1a22d70485983eac12b5b88dad1cf95	1325376051	43.150704	-112.154481	0
3	3	2019-01-01 00:01:16	3534093764340240	fraud_Kutch, Hermiston and Farrell	gas_transport	45.00	Jeremy	White	M	9443 Cynthia Court Apt. 038	...	46.2306	-112.1138	1939	Patent attorney	1967-01-12	6b849c168bdad6f867558c3793159a81	1325376076	47.034331	-112.561071	0
4	4	2019-01-01 00:03:06	375534208663984	fraud_Keeling-Crist	misc_pos	41.96	Tyler	Garcia	M	408 Bradley Rest	...	38.4207	-79.4629	99	Dance movement psychotherapist	1986-03-28	a41d7549acf90789359a9aa5346dcb46	1325376186	38.674999	-78.632459	0

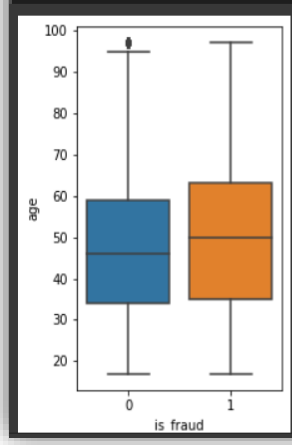
DATASET EXPLORATION AND FEATURE ENGINEERING

Creating a new “Age” variable

In our dataset, we had a column for date of birth (DOB), we used this field to calculate the age of the customer. Age is also a very good criterion to see which age group is more vulnerable to fraudulent activities because fraudsters target vulnerable people for conducting fraud.

```
#creating Age field from DOB to use it for Data Analysis
fullset['dob'] = pd.to_datetime(fullset.dob)
def from_dob_to_age(born):
    today = datetime.date.today()
    return today.year - born.year - ((today.month, today.day) < (born.month, born.day))

fullset['age']=fullset['dob'].apply(lambda x: from_dob_to_age(x))
```

```
#Let’s visualize if there is any relationship between the target variable and Age.
# Visualize relationship between is_fraud and age. We can see that majority of fraud happened between age 37 to 65.
fig= plt.figure(figsize=(10,5) )
fig.add_subplot(1,3,1)
ar_6=sns.boxplot(x=fullset["is_fraud"],y=fullset["age"])
```



From the above box plot, we can see that majority of fraud happened between the ages of 37 to 65.

Checking for Null Values

The next step would be to search the dataset for null values. Missing values in a dataset are usually triggered by data corruption or a failure to properly record the entire dataset. If missing values are not handled properly, it affects the evaluative metrics of various machine learning algorithms. Thus, handling missing values is critical. We can see that there are no missing values.

In case of missing values, we should either drop them or replace them with the mean value for analysis. We prefer replacing it with a mean value rather than dropping the observation because of data loss. However, for this problem, we don't have any missing value.

```
#check if any null value in the dataset
fullset.isna().sum()

#there is no null value in the dataset
```

```
Unnamed: 0                0
trans_date_trans_time     0
cc_num                    0
merchant                  0
category                  0
amt                       0
first                     0
last                      0
gender                    0
street                    0
city                      0
state                     0
zip                       0
lat                       0
long                      0
city_pop                  0
job                       0
dob                       0
trans_num                 0
unix_time                 0
merch_lat                 0
merch_long                0
is_fraud                  0
age                       0
dtype: int64
```

Checking for unique values for each Column

The information in the dataset is given for 999 unique Credit Card Numbers or Cardholders. Also, the information is about transactions completed with 693 unique Merchants.

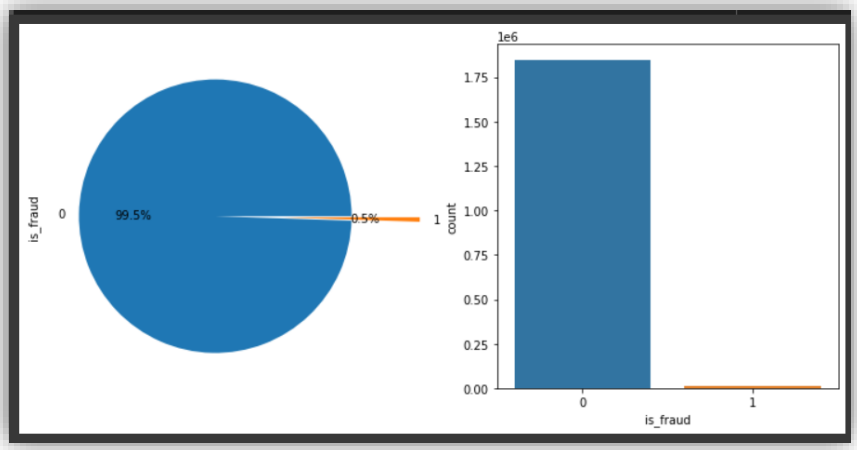
```
[ ] #check for unique values in the dataset.
    for col in fullset:
        uValue = np.unique(fullset[col])
        rValue = len(uValue)
        print('Unique Value {} -- {}'.format(col, rValue))
```

```
Unique Value Unnamed: 0 -- 1296675
Unique Value trans_date_trans_time -- 1819551
Unique Value cc_num -- 999
Unique Value merchant -- 693
Unique Value category -- 14
Unique Value amt -- 60616
Unique Value first -- 355
Unique Value last -- 486
Unique Value gender -- 2
Unique Value street -- 999
Unique Value city -- 906
Unique Value state -- 51
Unique Value zip -- 985
Unique Value lat -- 983
Unique Value long -- 983
Unique Value city_pop -- 891
Unique Value job -- 497
Unique Value dob -- 984
Unique Value trans_num -- 1852394
Unique Value unix_time -- 1819583
Unique Value merch_lat -- 1754157
Unique Value merch_long -- 1809753
Unique Value is_fraud -- 2
Unique Value age -- 81
```

Target Variable Distribution

We can see that our data is highly imbalanced because there are 1.8MM legitimate transactions and only 9651 fraud transactions

```
print(fullset['is_fraud'].value_counts())

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



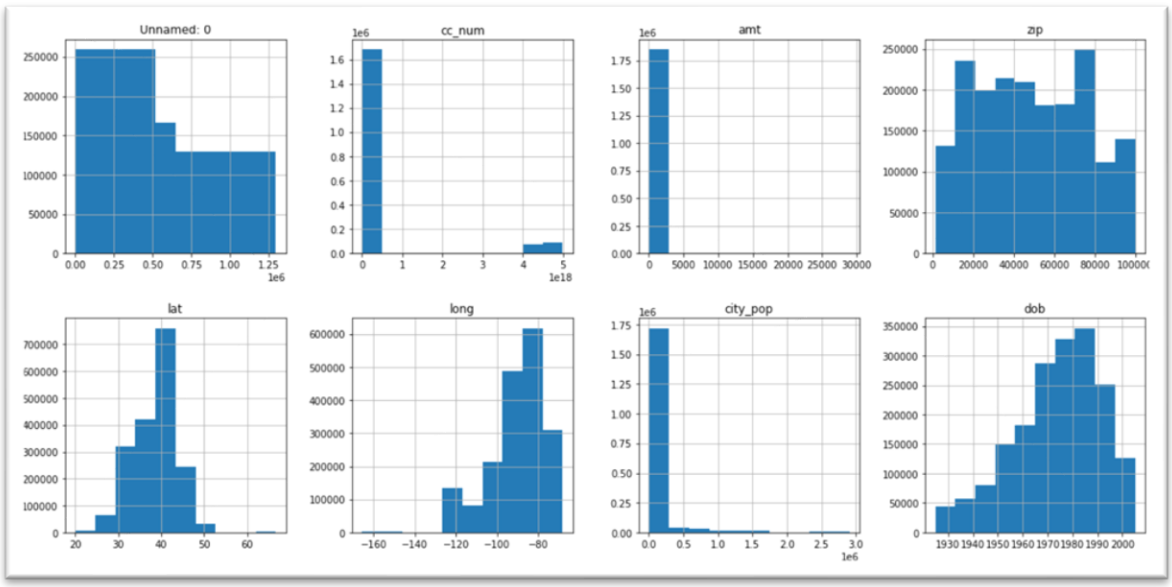

```
[ ]  #checking the percentage comparison for the imbalanced data

     Fraud = len(fullset[fullset['is_fraud']==1])
     NoFraud = len(fullset[fullset['is_fraud']==0])

     print('No Fraud Percentage {}'.format((NoFraud/(NoFraud+Fraud))*100))
     print('Fraud Percentage {}'.format((Fraud/(NoFraud+Fraud))*100))

     No Fraud Percentage 99.47899852839083
     Fraud Percentage 0.5210014716091717
```

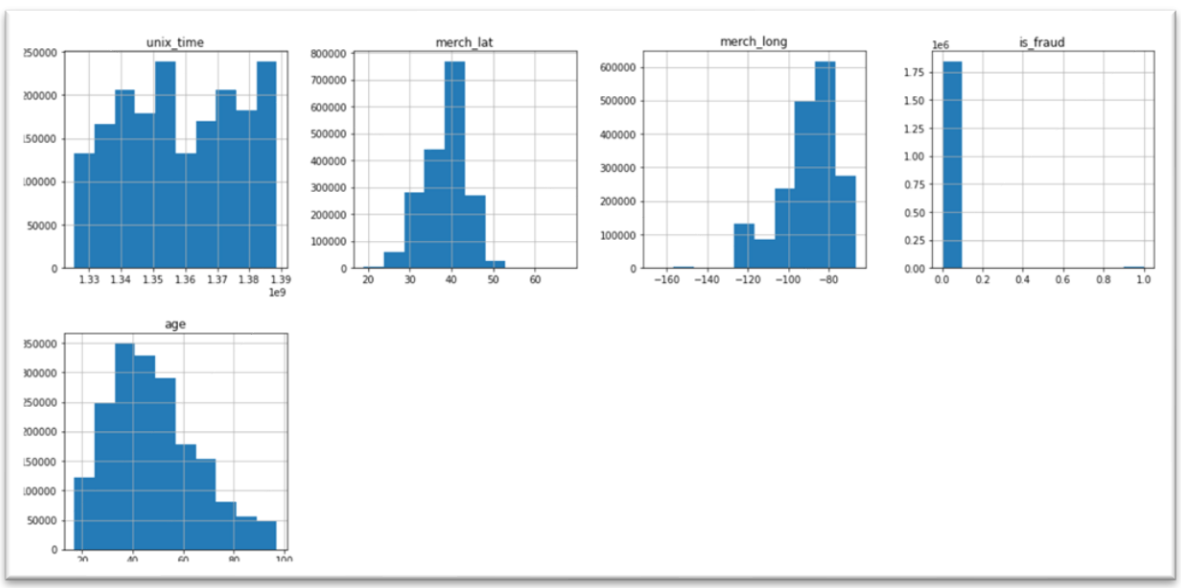
The imbalanced ratio is 99.47: 0.52 and the model trained on this data may yield high overall prediction accuracy since they most likely predict most samples belonging to the majority class. We will perform random under-sampling with Imblearn for this problem. We will convert the string data to int because it won't work with string. After we transform all of the data into an int, we will be checking the correlation between each column using a heatmap from seaborn and feature selection. As the features with high correlation are more linearly dependent, we can drop one of the features.

Feature Variables Data Distribution

By looking at the histogram, we can see that unnamed, zip and unix_time have uniform distribution. Whereas lat, long, dob, age, merch_lat, merch_long have a normal distribution.

```
#viewing features with the help of histogram.
#unnamed, zip and unix_time shows uniform distribution
p = fullset.hist(figsize = (20,20))
```





Addressing Categorical Data – Implementing Label Encoder

In this dataset, various columns do hold categorical values which poses a problem during the classification task. Thus, to solve this problem we have implemented the LABEL ENCODER method to the dataset to convert each value from categorical to a number and stored it in a new dataset.

Label Encoders are advantageous when there is a greater number of features involved which in our case holds.

```
#apply label encoder to the dataset to convert each value from categorical to a number and put in the new dataset
newset = fullset.apply(LabelEncoder().fit_transform)

#print dataset after label encoding
print(newset.head(5))
```

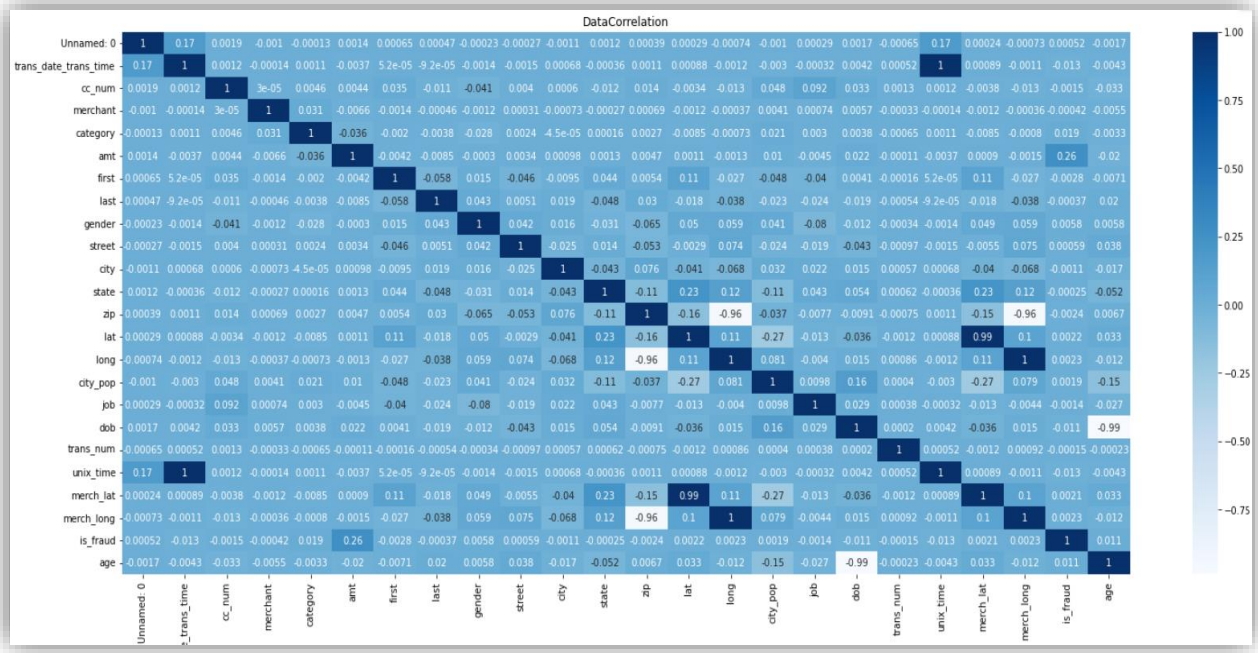
```
   Unnamed: 0  trans_date_trans_time  cc_num  merchant  category    amt  \
0           0                      0     454       514         8    397
1           1                      1      44       241         4  10623
2           2                      2     241       390         0  21911
3           3                      3     519       360         2   4400
4           4                      4     377       297         9   4096

   first  last  gender  street  ...  long  city_pop  job  dob  trans_num  \
0    164    18       0     576  ...   704       462  372  791      80326
1    312   161       0     439  ...    62        43  431  619     227462
2    116   386       1     610  ...    90       491  308  309    1169030
3    165   468       1     945  ...    93       370  330  405     777909
4    339   153       1     422  ...   764        22  116  746    1186866

   unix_time  merch_lat  merch_long  is_fraud  age
0          0     550600     1223201         0   17
1          1    1745263      110910         0   26
2          2    1451077      169563         0   43
3          3    1697797      164676         0   38
4          4     787219     1458121         0   18

[5 rows x 24 columns]
```

Data Correlation Using Heatmap

The heatmap below shows a high correlation of 0.99 between age and dob because age was created using dob so we will be dropping dob from our analysis. Also, there is a high correlation between merch_lat and lat (0.99), merch_long and long (1) and zip (0.96). So, we will be

dropping merch_lat, merch_long and zip taking correlation threshold to be 0.95.

```
'''We will plot correlations between different variables using a heatmap.'''
#correlation between features
fig, ax = plt.subplots(figsize=(25,10))
sns.heatmap(newset.corr(), annot = True, ax=ax, cmap = 'Blues')
plt.title("DataCorrelation")
plt.show()
```



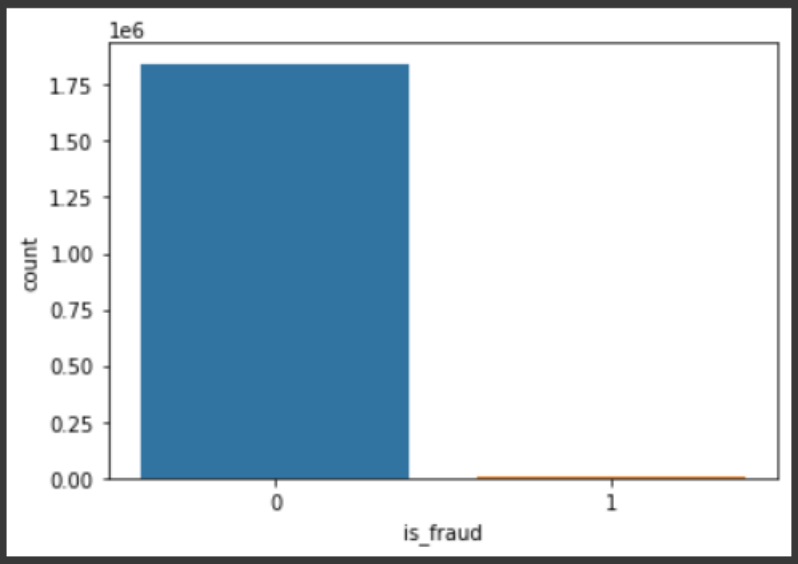
Dropping Other Unfavourable Columns

Apart from highly correlated variables such as “merch_lat”, “merch_long”, “zip”, we will be dropping “Unnamed:0” because it’s only an index number also “first”, “last”, “trans_num” because they were not contributing to the analysis and “dob” as we convert into “age”.

```
#we will be dropping is_fraud because we will use it as our y axis
dropvar = ['is_fraud','Unnamed: 0', 'first', 'last', 'dob', 'trans_num', 'merch_lat', 'merch_long',
'zip']
#create our dependent variables
x = newset.drop(dropvar, axis=1).copy()
print(x.head())
```

Addressing Target Label Imbalance

```
#visual the imbalanced data using bar chart
sns.countplot(x = 'is_fraud', data = fullset)
plt.show()
```



From the bar plot, it can be easily said that the target variable is highly imbalanced. This is a general classification type problem where the distribution of known classes is biased or skewed. This results in poor predictive modelling as predictions become favoured by the majority. Thus, dealing with imbalanced data before modelling takes priority.

There are numerous ways to solve this problem. However, we thought to go forward with Random Under Sampler where the total data will be sampled as the minimum data that we have. In this case, the number will be 9651 same as the total frauds.

The following are the advantages and disadvantages of using this method:

Advantages	Disadvantages
It allows performing analysis of data with a lower risk of making an error.	Large sample size is required for this method to be effective.
It provides ease in creating a sample group out of a larger dataset.	Loss of potentially important data
It involves a certain level of fairness allowing greater accuracy for the model.	It may or may not be an accurate representation of the actual world.

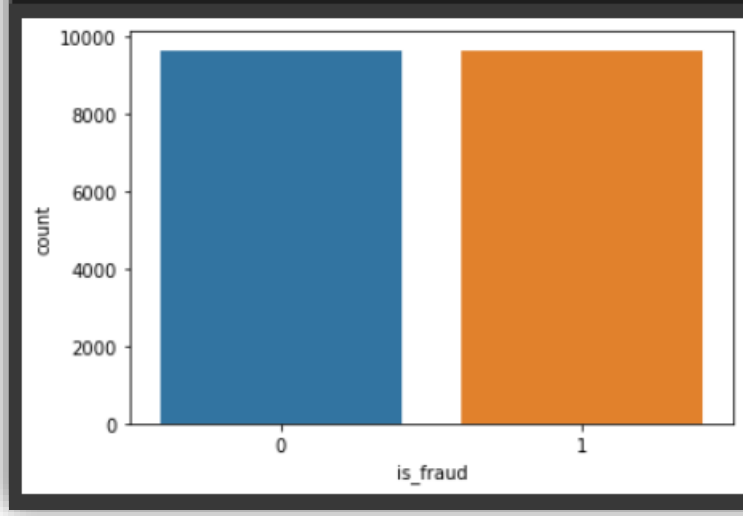
Implementation of Random Under Sampling

The code and bar graph below depicts the results of RandomUndersampler[1] implementation. The problem of imbalance is treated using this method.

```
#import library
import imblearn
from imblearn.under_sampling import RandomUnderSampler
import collections
from collections import Counter

unSampler = RandomUnderSampler(random_state=42, replacement=True)
xund, yund = unSampler.fit_resample(x,y)

print('original dataset shape:', Counter(y))
print('resample dataset shape', Counter(yund))
```

```
original dataset shape: Counter({0: 1842743, 1: 9651})
resample dataset shape Counter({0: 9651, 1: 9651})
```

```
#visual the data using bar chart after RandomUndersampler
underfraud = pd.DataFrame(yund)
sns.countplot(x = 'is_fraud', data = underfraud)
plt.show()
```



[1] *RandomUnderSampler,* imbalanced-learn.org, < https://imbalanced-learn.org/stable/references/generated/imblearn.under_sampling.RandomUnderSampler.html>, viewed 28 February 2022

SCALING AND DATA SPLIT – TEST AND TRAIN

Dataset standardization[2] is a familiar requirement for most machine learning estimators that are implemented in scikit-learn. The estimator might have a bad performance if the features do not look like normal standard distributed data such as Gaussian with zero mean and unit variance.
We used the min-max method[3] to rescale the data where we try to fit all the data points between the range of 0 to 1 so that the data points can become closer to each other. In this method of scaling the data, the minimum value of any feature gets converted into 0 and the maximum value of the feature gets converted into 1. We used this method since not all of our features were following the Gaussian distribution. The formula behind this method is:

```
X_std = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))
X_scaled = X_std * (max - min) + min
```

After standardization, we split the data into train and test using train_test_split[4].

```
#split data to training and test
from sklearn.svm import SVC

#split the data before scaling
xtrain, xtest, ytrain, ytest = train_test_split(xund, yund, test_size=0.3, random_state=42)

#scaling the data
mmscale = MinMaxScaler()
xtrain = mmscale.fit_transform(xtrain)
xtest = mmscale.fit_transform(xtest)
```

[2] *6.3. Preprocessing data*, Scikit-learn.org, <https://scikit-learn.org/stable/modules/preprocessing.html#preprocessing-scaler>, viewed 28 February 2022
[3] *sklearn.preprocessing.MinMaxScaler*, Scikit-learn.org, <https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html>, viewed 28 February 2022
[4] *sklearn.model_selection.train_test_split*, Scikit-learn.org, <https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html>, viewed 28 February 2022

FUNCTION FOR EVALUATION MATRIX

Since we will be using numerous different types of metrics to evaluate our models, we will be creating some functions to visualise and calculate evaluation metrics to save some time while we build our models. Here is the list of functions and how they will be implemented:

1. **cm(algo)**. This function required a model or algorithm has to be passed as a parameter to run. This function will result in confusion matrix visualisation.

```
from sklearn.metrics import ConfusionMatrixDisplay, accuracy_score, classification_report
from sklearn.metrics import log_loss, confusion_matrix, roc_auc_score

#plot the confusion matrix
def cm(algo): #it need the model variable after fitting the data
    disp = ConfusionMatrixDisplay(confusion_matrix=confusion_matrix(ytest, algo.predict(xtest)),
                                  display_labels=algo.classes_)
    disp.plot()
    plt.show()
```

2. **prints(cmatrix, acctest, acctrain, logtest, logtrain, precision1, precision0, recall1, recall0, f1, roctest, roctrain)**. This function required a confusion matrix score, testing accuracy score, training accuracy score, testing log loss, training log loss, precision class 1, precision class 0, recall class 1, recall class 0, F1 score, testing ROC AUC score, training ROC AUC score, as a parameter to execute. The results from this function are the printed-out list of metrics scoring results.

```
#function to print metrics score
def prints(cmatrix, acctest, acctrain, logtest, logtrain, precision1,
                 precision0, recall1, recall0, f1, roctest, roctrain):
    print("Confusion Matrix Accuracy Score = {:.2f}%\n".format(cmatrix))
    print("Accuracy Score: Training -> {:.2f}% Testing -> {:.2f}%\n".format(acctrain, acctest))
    print("Log Loss Training-> {} Testing -> {}\n".format(logtrain, logtest))
    print('Precision class 1: {:.2f}%\nPrecision class 0: {:.2f}%'.format(precision1, precision0))
    print('Recall class 1: {:.2f}%\nRecall class 0: {:.2f}%'.format(recall1, recall0))
    print('F1: {:.2f}%'.format(f1))
    print('ROC AUC Training-> {:.2f}% Testing-> {:.2f}%'.format(roctrain, roctest))
```

3. **insertlist(name, cmatrix, acctest, acctrain, logtest, logtrain, precision1, precision0, recall1, recall0, f1, roctest, roctrain)**. This function required a confusion matrix score, testing accuracy score, training accuracy score, testing log loss, training log loss, precision class 1, precision class 0, recall class 1, recall class 0, F1 score, testing ROC AUC score, training ROC AUC score, as a parameter to run. The purpose of this function is to add all of the metrics scoring into a **score[]** that we will be using at the end of the project for models' evaluation and selection.

```
#function add metrics score to list
def insertlist(name, cmatrix, acctest, acctrain, logtest, logtrain, precision1,
                 precision0, recall1, recall0, f1, roctest, roctrain):
    score.append([name, cmatrix, acctest, acctrain, logtest, logtrain, precision1,
                 precision0, recall1, recall0, f1, roctest, roctrain])
```

4. **auc_plot(algo)**. This function required a model or algorithm as a parameter to run. This will result in a ROC AUC learning curve comparison between training and testing prediction.

```
# Roc Curve Characterics
def auc_plot(algo):
    #create AUC curve
    test_prob = algo.predict_proba(xtest)[::,1]
    train_prob = algo.predict_proba(xtrain)[::,1]
    roctest = roc_auc_score(ytest, test_prob)
    roctrain = roc_auc_score(ytrain, train_prob)

    fpr_test, tpr_test, _ = roc_curve(ytest,  test_prob)
    fpr_train, tpr_train, _ = roc_curve(ytrain,  train_prob)
    plt.title("Area Under Curve")
    plt.plot(fpr_test,tpr_test,label="AUC Test="+str(roctest))
    plt.plot(fpr_train,tpr_train,label="AUC Train="+str(roctrain))
    plt.ylabel('True Positive Rate')
    plt.xlabel('False Positive Rate')
    plt.legend(loc=4)
    plt.grid(True)
    plt.show()
```

5. **scr(algo, name)**. This function required a model or algorithm and string as a parameter to be executed. It will calculate the models' metrics score evaluation such as confusion matrix, accuracy score, log loss, precision, recall, F1 and ROC AUC score.

```
#metrix function
def scr(algo, name): #algo = model, name = string of the model name
    predtest = algo.predict(xtest)
    predtrain = algo.predict(xtrain)

    #confussion matrix percentage
    tn, fp, fn, tp = confusion_matrix(ytest, predtest).ravel()
    tst = ytest.count()
    cmatrix = ((tn + tp)/tst)*100

    #accuracy score
    acctest = (accuracy_score(ytest, predtest))*100
    acctrain = (accuracy_score(ytrain, predtrain))*100

    #log loss
    logtest = log_loss(ytest,predtest)
    logtrain = log_loss(ytrain,predtrain)

    #classification report
    precision1 = (tp / (tp+fp))*100
    precision0 = (tn/(tn+fn))*100
    recall1 = (tp/(tp+fn))*100
    recall0 = (tn/(tn+fp))*100
    f1 = 2*(precision1 * recall1)/(precision1 + recall1)

    #roc auc score
    test_prob = algo.predict_proba(xtest)[::,1]
    train_prob = algo.predict_proba(xtrain)[::,1]
    roctest = (roc_auc_score(ytest, test_prob))*100
    roctrain = (roc_auc_score(ytrain, train_prob))*100

    insertlist(name, cmatrix, acctest, acctrain, logtest, logtrain, precision1,
               precision0, recall1, recall0, f1, roctest, roctrain)
    #print metrics score
    return prints(cmatrix, acctest, acctrain, logtest, logtrain, precision1,
                  precision0, recall1, recall0, f1, roctest, roctrain)
```

ALGORITHMS

There will be 7 machine learning algorithms implemented in the first stage of the proposed approach such as:

- Logistic Regression
- Random Forest
- Decision Tree
- SVM
- KNN
- ANN
- CNN

In the second stage, we tuned the models using hyper-parameters to increase the accuracy of each of the algorithms.

1. Logistic Regression[5]

This is one of the traditional machine learning algorithms that is still used today due to its quick analysis and simple method of processing the features of a class. Logistic Regression is a classification type algorithm that determines the likelihood of the event happening. In our case, the classification type algorithm will work accurately to determine the binary outcome of the data i.e., whether fraud is 0 or 1.

The following are the advantages and disadvantages of Logistic Regression:

Advantages	Disadvantages
It is easier to implement and works efficiently during model training.	For a lesser number of observations than features result in overfitting.
It doesn't make any assumptions about class distributions in feature space.	It assumes linearity between dependent and independent variables

Implementation

```
from sklearn.linear_model import LogisticRegression
logreg = LogisticRegression()

logreg.fit(xtrainscaled, ytrain)
```

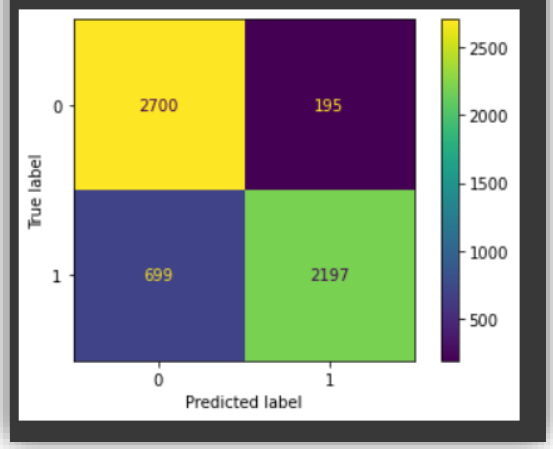
```
LogisticRegression()
```

[5] *sklearn.linear_model.LogisticRegression*, scikit-learn.org, <https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html>, viewed 28 February 2022

Confusion Matrix[6]

A confusion matrix is a summary of prediction results on a classification problem. The number of correct and incorrect predictions are summarized with count values and broken down by each class. This is the key to the confusion matrix.

```
#confusion matrix
cm(logreg)
```



From the confusion matrix, we can see the True Negative is 2700 and the True positive is 2197. We will calculate the for the accuracy score: (2700+2197)/5791 = 0.8456. The accuracy score is 84.56%.

Evaluation

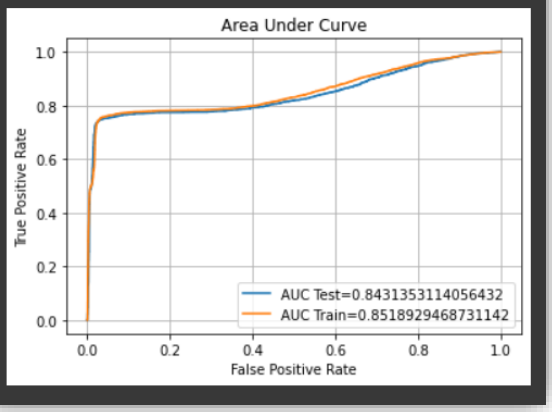
```
scr(logreg, 'Logreg')

Confusion Matrix Accuracy Score = 84.56%

Accuracy Score: Training -> 85.06% Testing -> 84.56%

Log Loss Training-> 5.161286341869521 Testing -> 5.332036266370064

Precision class 1: 91.85%
Precision class 0: 79.44%
Recall class 1: 75.86%
Recall class 0: 93.26%
F1: 83.09%
ROC AUC Training-> 85.19% Testing-> 84.31%
```

From the above, we can context that logistic regression holds 84.56% accuracy for unseen data while, 85.06% accuracy for seen data which can be considered overfitting.

[6] *sklearn.metrics.ConfusionMatrixDisplay*, scikit-learn.org, < https://scikit-learn.org/stable/modules/generated/sklearn.metrics.ConfusionMatrixDisplay.html>, 28 February 2022

Plotting AUC Curve[7] Characteristics

```
#plot learning curve
auc_plot(logreg)
```



The 0.843 value of AUC defines that there is an 84.31% chance that the model will be able to distinguish between positive and negative classes for unseen datasets. The comparison for Train and Test AUC curves indicates little overfitting.

Logistic Regression Tuning using Grid Search CV[8]

To get the best set of hyperparameters we can use Grid Search. GridSearchCV is an exhaustive search over specified parameter values for an estimator, it will pass all combinations of hyperparameters one by one into the model and checks the result. Finally, it gives us the set of hyperparameters that gives the best result after passing in the model.

```
log_param = [
    {
        'C': [0.01, 0.1, 1.0, 10.0, 100.0],
        'penalty': ['l2'],
        'solver': ['newton-cg', 'sag']
    },
]

#tuned the model using gridsearch and kfold
opt_log = GridSearchCV(LogisticRegression(), param_grid = log_param, scoring='accuracy', cv=kfold, verbose = 0)

#fit the tuned model with training dataset
opt_log.fit(xtrain, ytrain)
print(opt_log.best_params_) #best parameter tuning result
```

```
{'C': 0.01, 'penalty': 'l2', 'solver': 'newton-cg'}
```

[7] *sklearn.metrics.roc_curve*, scikit-learn.org, <https://scikit-learn.org/stable/modules/generated/sklearn.metrics.roc_curve.html>, viewed 28 February 2022

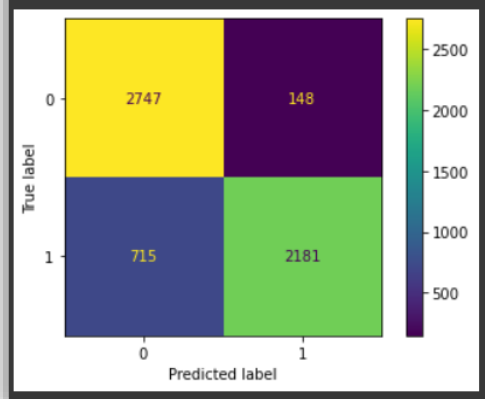
[8] *sklearn.model_selection.GridSearchCV*, scikit-learn.org, <https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html>, viewed 28 February 2022

With the above grid search, we utilize a parameter grid that consists of a log_param dictionary with three parameters. The model is then implemented using the best parameters.

```
opt_log = LogisticRegression(C=0.01, penalty='l2', solver='newton-cg')
opt_log.fit(xtrain, ytrain)
```

```
LogisticRegression(C=0.01, solver='newton-cg')
```

```
cm(opt_log)
```



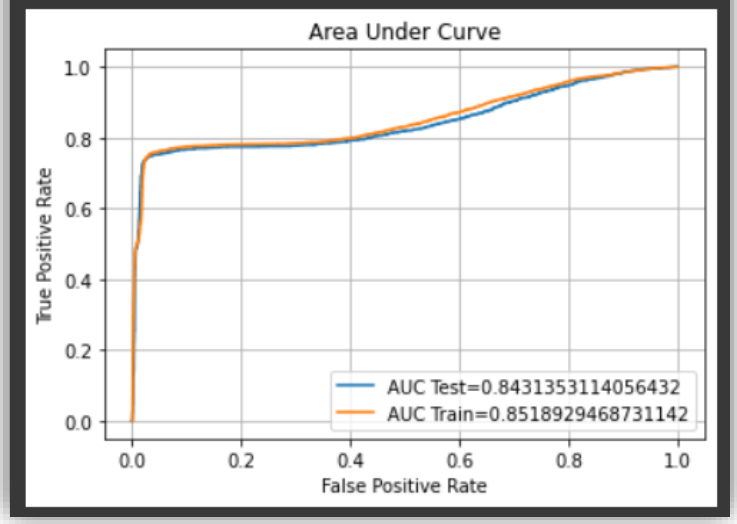
From the confusion matrix, we can see the True Negative is 2747 and the True positive is 2181. We will calculate the for the accuracy score: (2747+2181)/5791 = 0.8510. The accuracy score is 85.10%.

```
scr(opt_log, 'logreg_tuned')
```

```
Confusion Matrix Accuracy Score = 85.10%

Accuracy Score: Training -> 85.56% Testing -> 85.10%

Log Loss Training-> 4.987448139740813 Testing -> 5.1471390725656345

Precision class 1: 93.65%
Precision class 0: 79.35%
Recall class 1: 75.31%
Recall class 0: 94.89%
F1: 83.48%
ROC AUC Training-> 84.70% Testing-> 83.88%
```

Plotting AUC Curve Characteristics

```
#plot learning curve
auc_plot(logreg)
```



The 0.843 value of AUC defines that there is an 84.31% chance that the model will be able to distinguish between positive and negative classes for unseen datasets. The comparison for Train and Test AUC curves indicates little overfitting.

2. Random forest[9]

Random forests, called random decision forests, are an ensemble learning algorithm, used for regression, classification and other tasks that works by training many decision trees.

The following are the advantages and disadvantages:

Advantages	Disadvantages
Random Forest Algorithm generally yields high accuracy.	With the increase in complexity accuracy decreases.
It works well with a large number of Data.	Interpretation of forest is much more difficult than just a single decision tree.

[9] *sklearn.ensemble.RandomForestClassifier*, scikit-learn.org, < https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html >, Viewed 28 February 2022

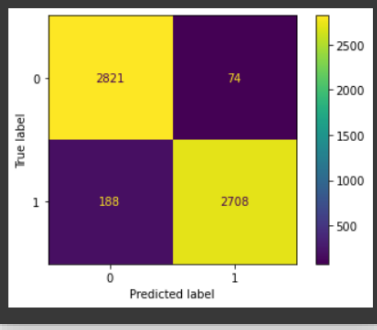
Random Forest Implementation

```
#Random Forest
from sklearn.ensemble import RandomForestClassifier
RF = RandomForestClassifier()
#fit the model with training dataset
RF.fit(xtrain, ytrain)
```

```
RandomForestClassifier()
```

Random Forest Confusion Matrix

```
#calling function to plot confusion matrix
cm(RF)
```



From the confusion matrix, we can see the True Negative is 2821 and the True positive is 2708. We will calculate the accuracy score: (2821+2708)/5791 = 0.9548 The accuracy score is 95.48%.

Random Forest Model Evaluation

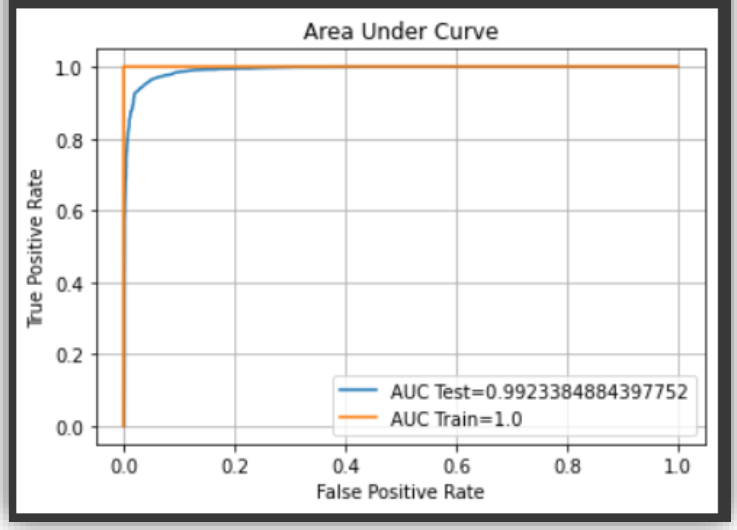
```
#calling function to calculate model score
scr(RF, 'RF')
```

```
Confusion Matrix Accuracy Score = 95.48%

Accuracy Score: Training -> 99.99% Testing -> 95.48%

Log Loss Training-> 0.0025563449333819988 Testing -> 1.562634879239585

Precision class 1: 97.34%
Precision class 0: 93.75%
Recall class 1: 93.51%
Recall class 0: 97.44%
F1: 95.39%
ROC AUC Training-> 100.00% Testing-> 99.23%
```

From the above, we can evaluate that Random Forest Classifier holds 95.20% accuracy for seen data while, 100% accuracy for unseen data. This indicates that the model is over-fitting.

Plotting AUC Curve Characteristics

```
auc_plot(RF)
```



The 0.9923 value of AUC defines that there is a 99.23% chance that the model will be able to distinguish between positive and negative classes for unseen datasets. The comparison for Train and Test AUC curves indicates overfitting.

Random Forest Tuning using RandomizedSearchCV[10] and K-fold[11]

StratifiedKFold cross-validation is implemented on all of the models in our projects, this object is a variation of K-fold that returns stratified folds. The folds are made by preserving the percentage of samples for each class.

In contrast to GridSearchCV, a Randomized search will not try all of the parameters, but rather a fixed number of parameter settings is sampled from the specified distributions. The number of parameter settings that are tried is given by n_iter.

If all parameters are presented as a list, sampling without replacement is performed. If at least one parameter is given as a distribution, sampling with replacement is used. It is highly recommended to use continuous distributions for continuous parameters.

---

[10] *sklearn.model_selection.RandomizedSearchCV*, scikit-learn.org, <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html>, viewed 28 February 2022

[11] *sklearn.model_selection.StratifiedKFold*, scikit-learn.org, < https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html>, viewed 28 February 2022

```python
# Number of trees in random forest
n_estimators = [int(x) for x in np.linspace(start = 200, stop = 2000, num = 10)]
# Number of features to consider at every split
max_features = ['auto', 'sqrt']
# Maximum number of levels in tree
max_depth = [int(x) for x in np.linspace(10, 110, num = 11)]
max_depth.append(None)
# Minimum number of samples required to split a node
min_samples_split = [2, 5, 10]
# Minimum number of samples required at each leaf node
min_samples_leaf = [1, 2, 4]
# Method of selecting samples for training each tree
bootstrap = [True, False]


random_grid = {'n_estimators': n_estimators,
               'max_features': max_features,
               'max_depth': max_depth,
               'min_samples_split': min_samples_split,
               'min_samples_leaf': min_samples_leaf,
               'bootstrap': bootstrap}

#implement parameter tuning using randomized search cv and kfold
opt_RF = RandomizedSearchCV(estimator = RF, param_distributions = random_grid, n_iter = 100, cv = kfold, verbose=2, random_state=42, n_jobs = -1)
#fit the model with training dataset
opt_RF.fit(xtrain, ytrain)
print(opt_RF.best_params_) #best parameter tuning result
```

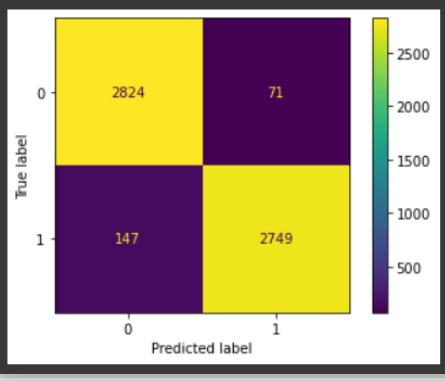
```
Fitting 10 folds for each of 100 candidates, totalling 1000 fits
{'n_estimators': 1000, 'min_samples_split': 2, 'min_samples_leaf': 1, 'max_features': 'auto', 'max_depth': 50, 'bootstrap': False}
```

With the above grid search, we utilize a parameter grid that consists of a random_grid dictionary with three parameters. The model is implemented using the best parameters.

```python
#build the model with hyperparameter tuning
opt_RF = RandomForestClassifier(n_estimators= 1000, min_samples_split= 2, min_samples_leaf= 1,
                                max_features= 'auto', max_depth= 50, bootstrap= False)
opt_RF.fit(xtrain, ytrain)
```

```
RandomForestClassifier(bootstrap=False, max_depth=50, n_estimators=1000)
```

```python
#calling function to plot confusion matrix
cm(opt_RF)
```



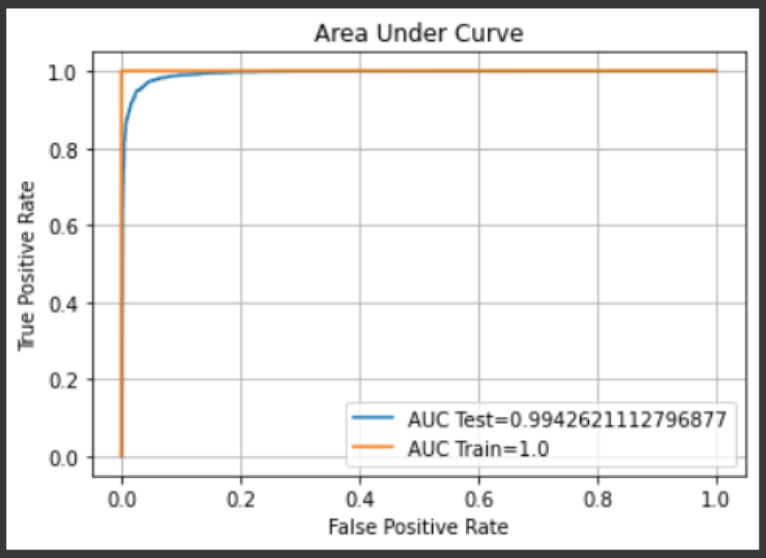
From the confusion matrix, we can see the True Negative is 2824 and the True positive is 2749. We will calculate the accuracy score: (2824+2749)/5791 = 0.9624 The accuracy score is 96.24%.

```
#calling function to show the metrics score
scr(opt_RF, 'RF_tuned')
```

```
Confusion Matrix Accuracy Score = 96.24%

Accuracy Score: Training -> 100.00% Testing -> 96.24%

Log Loss Training-> 9.992007221626413e-16 Testing -> 1.300208949319302

Precision class 1: 97.48%
Precision class 0: 95.05%
Recall class 1: 94.92%
Recall class 0: 97.55%
F1: 96.19%
ROC AUC Training-> 100.00% Testing-> 99.43%
```

Plotting AUC Curve Characteristics

```
#plot learning curve
auc_plot(opt_RF)
```



The accuracy score after hyperparameter tuning is increased by around 1%, however, the overfitting still is a problem as the training accuracy is 100%.

3. Decision Tree[12]

Decision Tree works by transforming data into a tree representation. This is a computational method that aims to classify and predict. It has a tree-like process, including a root node, leaf node, and branch. Each internal node indicates a test based on its attributes, the outcome of the test indicates each branch, and the class label holds each leaf node.

Advantages	Disadvantages
Decision Tree doesn't require scaling and normalization of data.	Small changes in the dataset cause instability in the structure of the tree.
Missing values do not affect the overall accuracy of the model.	The decision becomes inadequate for continuous values.

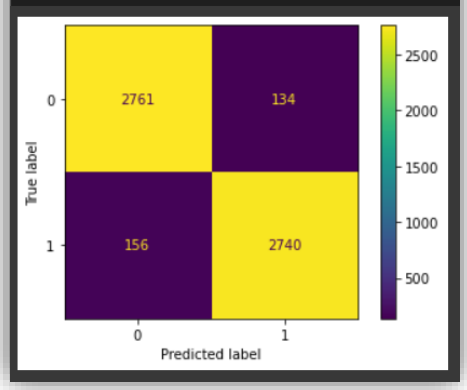
Decision Tree Implementation

```
#import the library
from sklearn.tree import DecisionTreeClassifier
#create the model
decTree = DecisionTreeClassifier()
#fit the model with training data
decTree.fit(xtrain, ytrain)
```

```
DecisionTreeClassifier()
```

Decision Tree Confusion Matrix

```
#confusion matrix
cm(decTree)
```



[12] *1.10. Decision Trees*, scikit-learn.org, https://scikit-learn.org/stable/modules/tree.html#:~:text=Decision%20Trees%20(DTs)%20are%20a,as%20a%20piecewise%20constant%20approximation.>, viewed 28 February 2022

From the confusion matrix, we can see the True Negative is 2761 and the True positive is 2740. We will calculate the for the accuracy score: (2761+2740)/5791 = 0.9499. The accuracy score is 94.99%.

Decision Tree Model Evaluation

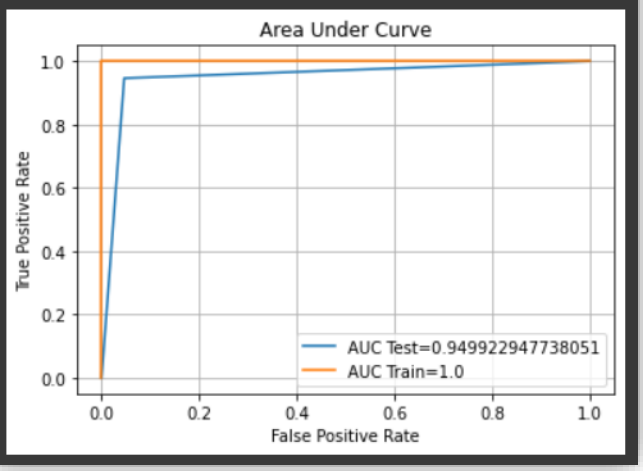
```
#calling metrics score function
scr(decTree, 'decTree')
```

```
Confusion Matrix Accuracy Score = 94.99%

Accuracy Score: Training -> 100.00% Testing -> 94.99%

Log Loss Training-> 9.992007221626413e-16 Testing -> 1.7296412192332498

Precision class 1: 95.34%
Precision class 0: 94.65%
Recall class 1: 94.61%
Recall class 0: 95.37%
F1: 94.97%
ROC AUC Training-> 100.00% Testing-> 94.99%
```

From the above, we can evaluate that the Decision Tree Classifier holds 100% accuracy for seen data while, 94.99% accuracy for unseen data. This indicates that the model is over-fitting.

Plotting AUC Curve Characteristics

```
#plot learning curve
auc_plot(decTree)
```



The 0.9499 value of AUC defines that there is a 94.99% chance that the model will be able to distinguish between positive and negative classes for unseen datasets. The comparison for Train and Test AUC curves indicates overfitting.

Tuning Decision Tree using RandomizedSearchCV

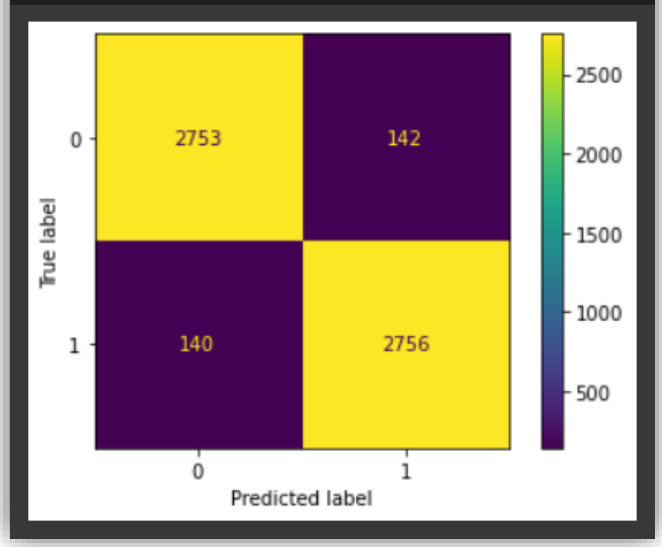
```
#parameter variables
criterion = ['gini', 'entropy']
max_depth = [3,6,9]
min_samples_split = [3,6,9]
min_samples_leaf = [2,4,8]

#put the variables to dict
random_tree = {'criterion': criterion,
               'max_depth': max_depth,
               'min_samples_split': min_samples_split,
               'min_samples_leaf': min_samples_leaf}

#implement hyperparameter tuning to the model
opt_decTree = RandomizedSearchCV(estimator = decTree, param_distributions = random_tree, cv = kfold, verbose=1, n_jobs = -1)
#fit the model with training dataset
opt_decTree.fit(xtrain, ytrain)
print(opt_decTree.best_params_) #best parameter
```

```
Fitting 10 folds for each of 10 candidates, totalling 100 fits
{'min_samples_split': 9, 'min_samples_leaf': 4, 'max_depth': 9, 'criterion': 'entropy'}
```

With the above grid search, we utilize a parameter grid that consists of a random_tree dictionary with three parameters. The model is implemented using the best parameters.

Decision Tree Confusion Matrix After Tuning

```
#function to plot confusion matrix
cm(opt_decTree)
```



From the confusion matrix, we can see the True Negative is 2753 and the True positive is 2756. We will calculate the for the accuracy score: (2761+2740)/5791 = 0.9513. The accuracy score is 95.13% which increased compared to the model without tuning.

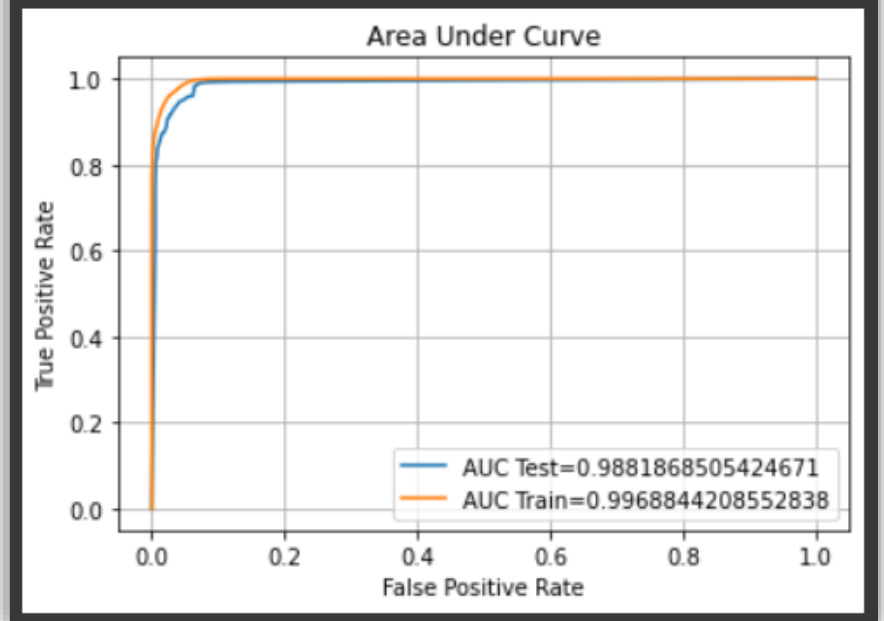
Decision Tree Model Evaluation After Tuning

```
#function to print metrics score
scr(opt_decTree, 'decTree_tuned')
```

```
Confusion Matrix Accuracy Score = 95.13%

Accuracy Score: Training -> 96.94% Testing -> 95.13%

Log Loss Training-> 1.0583452077783897 Testing -> 1.681928593714368

Precision class 1: 95.10%
Precision class 0: 95.16%
Recall class 1: 95.17%
Recall class 0: 95.09%
F1: 95.13%
ROC AUC Training-> 99.69% Testing-> 98.82%
```

Plotting AUC Curve Characteristics

```
#plot learning curve
auc_plot(opt_decTree)
```



The accuracy score after hyperparameter tuning is increased by around 1%, and the overfitting is reduced.

4. Support Vector Machines (SVM)[13]

SVM algorithm uses a decision boundary known as hyperplane for its classification task. This hyperplane acts as a decision line between different classes.

The following are the advantages and disadvantages:

Advantages	Disadvantages
SVM work more efficiently in high dimensional space.	The accuracy of the model decreases relatively when there is noise in the data.
A small change in data does not affect the hyperplane.	SVM tends to underfit when there is a large number of features involved.

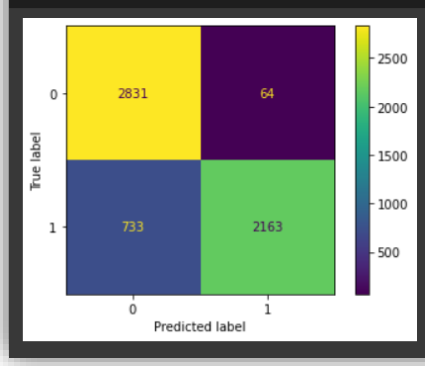
SVM Implementation

```
#build SVM classification
SVM1 = SVC(random_state = 42, probability=True)
SVM1.fit(xtrain, ytrain)
```

```
SVC(probability=True, random_state=42)
```

SVM Confusion Matrix

```
#calling cm function to plot the confusion matrix
cm(SVM1)
```



From the confusion matrix, we can see the True Negative is 2831 and the True positive is 2163. We will calculate the for the accuracy score: (2831+2163)/5791 = 0.8624. The accuracy score is 86.24%.

[13] *1.4. Support Vector Machines*, scikit-learn.org, https://scikit-learn.org/stable/modules/svm.html#:~:text=Support%20vector%20machines%20(SVMs)%20are,than%20the%20number%20of%20samples.>, viewed 28 February 2022

SVM Model Evaluation

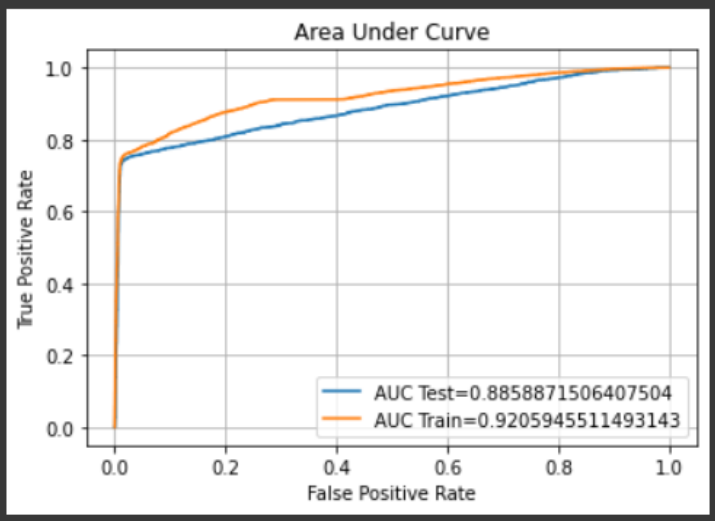
```
#calling function scr to print the metrix score
scr(SVM1, 'SVM1')
```

```
Confusion Matrix Accuracy Score = 86.24%

Accuracy Score: Training -> 86.96% Testing -> 86.24%

Log Loss Training-> 4.504286815182208 Testing -> 4.753489200652625

Precision class 1: 97.13%
Precision class 0: 79.43%
Recall class 1: 74.69%
Recall class 0: 97.79%
F1: 84.44%
ROC AUC Training-> 92.06% Testing-> 88.59%
```

From the figure above, we can evaluate that SVM holds 86.96% accuracy for seen data while 86.24% accuracy for unseen data.

Plotting AUC Curve Characteristics

```
#calling function to plot the ROC_AUC learning curve
auc_plot(SVM1)
```



The 0.8858 value of AUC defines that there is an 88.58% chance that the model will be able to distinguish between positive and negative classes for unseen datasets.

Improving SVM using Cross-Validation (Hyper Parameter Tuning)

```
kfold = StratifiedKFold(n_splits=10, shuffle=True, random_state=5)
param_grid = [
    {
        'C': [0.5, 1, 10, 15, 100],
        'gamma': ['scale', 1, 0.1, 0.01, 0.001, 0.0001],
        'kernel': ['rbf']
    },
]

#tuning the model using gridsearchCV and stratifiedkfold
optimal_param = GridSearchCV(
    SVC(), param_grid, cv=kfold, scoring='accuracy', verbose = 0)
```

The figure above, it shows that utilisation of parameter grid that consists in param_grid dictionary with three parameters. The model is implemented using the best parameters.

```
#fitting the tuned model with training dataset
optimal_param.fit(xtrain, ytrain)

#print the best parameter
print(optimal_param.best_params_)
```

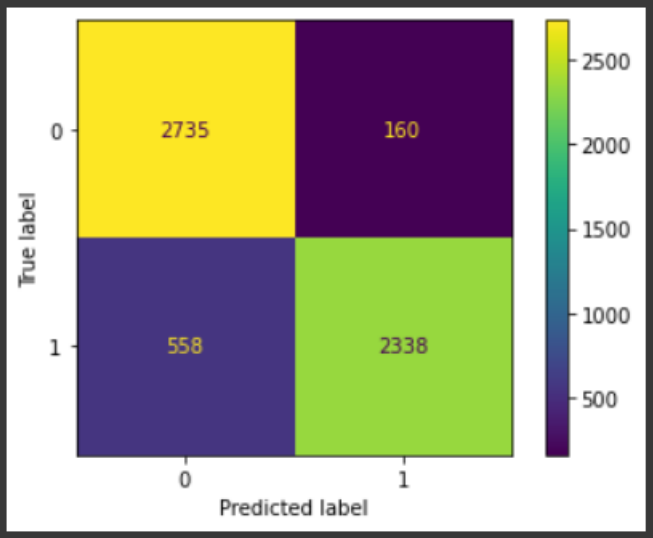
```
{'C': 15, 'gamma': 'scale', 'kernel': 'rbf'}
```

```
#build the model with hyperparameter tuning
optimal_param = SVC(random_state = 42, probability=True, C=15, gamma='scale', kernel='rbf')
optimal_param.fit(xtrain,ytrain)
```

```
SVC(C=15, probability=True, random_state=42)
```

SVM Confusion Matrix

```
#plot confusion matrix for the tuned model
cm(optimal_param)
```



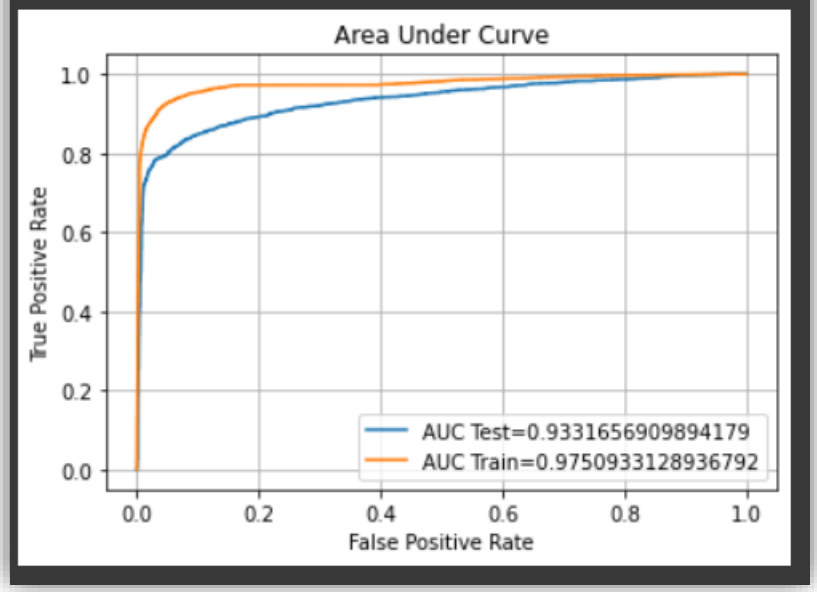
SVM Model Evaluation After Tuning

```
#print metrics score
scr(optimal_param, 'SVM_tuned')
```

```
Confusion Matrix Accuracy Score = 87.60%

Accuracy Score: Training -> 92.44% Testing -> 87.60%

Log Loss Training-> 2.6100345685534956 Testing -> 4.282329370943654

Precision class 1: 93.59%
Precision class 0: 83.05%
Recall class 1: 80.73%
Recall class 0: 94.47%
F1: 86.69%
ROC AUC Training-> 97.51% Testing-> 93.32%
```

Plotting AUC curve Characteristics

```
#plot the learning curve
auc_plot(optimal_param)
```



The 0.9331 value of AUC defines that there is a 93.31% chance that the model will be able to distinguish between positive and negative classes for unseen datasets. The comparison for Train and Test AUC curves indicates overfitting.

5. K-Nearest Neighbours[14]

K-Nearest Neighbours also known as KNN is a supervised machine learning algorithm that uses labelled data to perform classification over a dataset.

Advantages	Disadvantages
KNN specifically does not require training. It implements itself through tagging and making predictions using historical data.	KNN tends to underperform when there is a large number of data points involved.
KNN holds efficiency due to its instance-based learning.	With the increase in the number of new data points, the complexity of the algorithm increases.

[14] *sklearn.neighbors.KNeighborsClassifier*, scikit-learn.org, https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html>, viewed 28 February 2022

Implementation

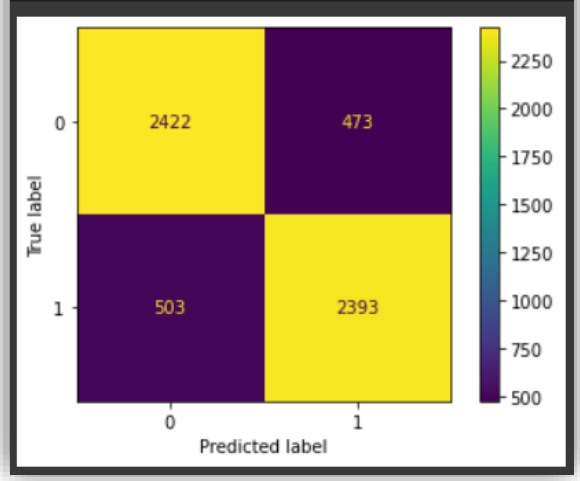
```
#import library for KNN
from sklearn.neighbors import KNeighborsClassifier

#build the model
neigh = KNeighborsClassifier(n_neighbors=3)
#fit the model with training dataset
neigh.fit(xtrain, ytrain)
```

```
KNeighborsClassifier(n_neighbors=3)
```

Confusion Matrix

A confusion matrix is a summary of prediction results on a classification problem. The number of correct and incorrect predictions are summarized with count values and broken down by each class. This is the key to the confusion matrix.

```
#funtion for confusion matrix plot
cm(neigh)
```



From the confusion matrix, we can see the True Negative is 2422 and the True positive is 2393. We will calculate the for the accuracy score: (2422+2393)/5791 = 0.8315. The accuracy score is 83.15%.

Evaluation

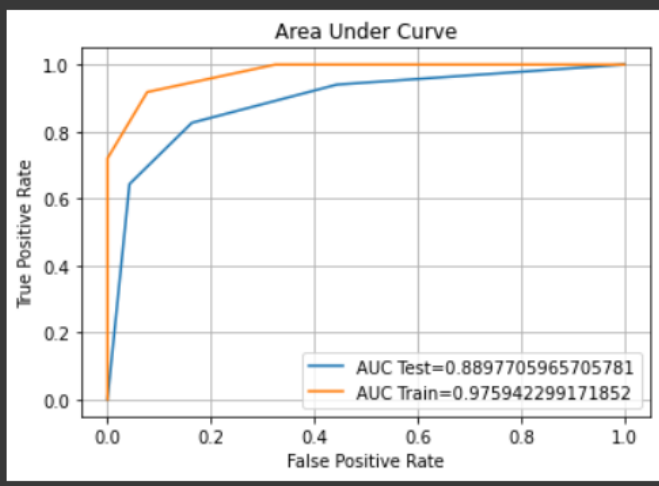
```
#function for metrics calculation
scr(neigh, 'KNN')
```

```
Confusion Matrix Accuracy Score = 83.15%

Accuracy Score: Training -> 92.06% Testing -> 83.15%

Log Loss Training-> 2.7429888285697164 Testing -> 5.821140385255969

Precision class 1: 83.50%
Precision class 0: 82.80%
Recall class 1: 82.63%
Recall class 0: 83.66%
F1: 83.06%
ROC AUC Training-> 97.59% Testing-> 88.98%
```

From the above, we can context that KNN holds 92.06% accuracy for unseen data while, 83.15% accuracy for seen data which can be considered overfitting.

Plotting AUC Curve Characteristics

```
#plot learning curve
auc_plot(neigh)
```



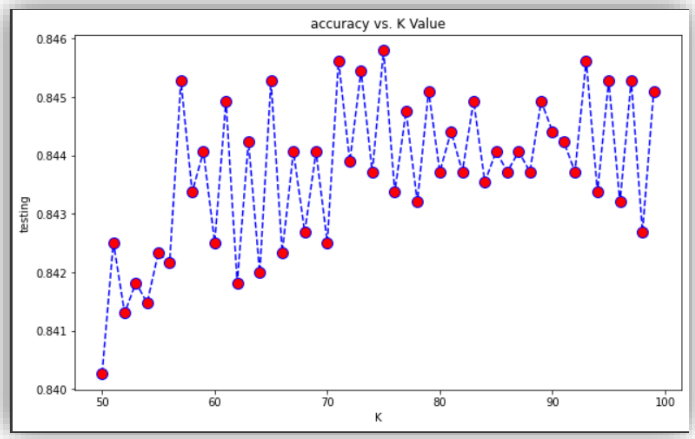
The 0.8897 value of AUC defines that there is an 88.97% chance that the model will be able to distinguish between positive and negative classes for unseen datasets. The comparison for Train and Test AUC curves indicates little overfitting.

KNN Tuning using Grid Search CV

To get the best set of hyperparameters we can use Grid Search. Grid Search passes all combinations of hyperparameters one by one into the model and checks the result. Finally, it gives us the set of hyperparameters that gives the best result after passing in the model.

```
#finding the optimum k number
acc=[]
for i in range(50,100):
    neigh = KNeighborsClassifier(n_neighbors = i).fit(xtrain,ytrain)
    yhat = neigh.predict(xtest)
    acc.append(accuracy_score(ytest, yhat))

#plot to visualise the best k
plt.figure(figsize=(10,6))
plt.plot(range(50,100),acc,color = 'blue',linestyle='dashed',
         marker='o',markerfacecolor='red', markersize=10)
plt.title('accuracy vs. K Value')
plt.xlabel('K')
plt.ylabel('testing')
```

The line chart below shows which neighbour has the best accuracy, which we will use to narrow down the parameter search for tuning.



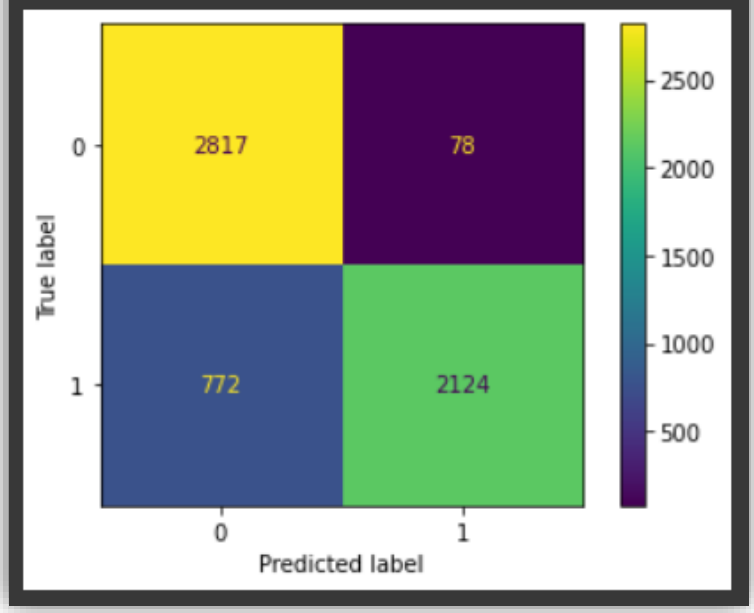

```
#parameter dict
tuning_params = {
    'n_neighbors' : [64,71,75,94], #from the plot above
    "leaf_size":[5,10,20,30],
    "p":[1,2]
}

#implement hyperparameter tuning to the model
opt_knn = GridSearchCV(neigh, param_grid = tuning_params, cv = kfold, verbose = 1, n_jobs = -1)
#fit the model with training dataset
opt_knn.fit(xtrain, ytrain)
print(opt_knn.best_params_) #best parameter
```

```
Fitting 10 folds for each of 32 candidates, totalling 320 fits
{'leaf_size': 5, 'n_neighbors': 94, 'p': 1}
```

With the above grid search, we utilize a parameter grid that consists of a tuning_params dictionary with three parameters. The model is implemented using the best parameters.

KNN Confusion Matrix After Tuning

```
#function for confusion matrix plot
cm(opt_knn)
```



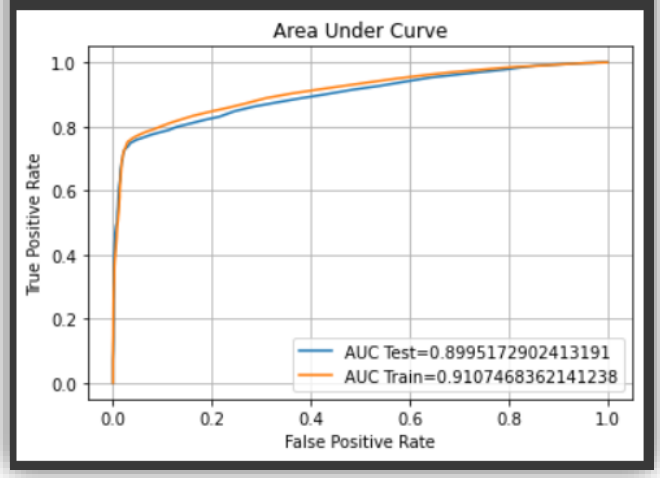
KNN Model Evaluation After Tuning

```
#function for metrics calculation
scr(opt_knn, 'knn_tuned')
```

```
Confusion Matrix Accuracy Score = 85.32%

Accuracy Score: Training -> 85.72% Testing -> 85.32%

Log Loss Training-> 4.931199969929889 Testing -> 5.069594595799281

Precision class 1: 96.46%
Precision class 0: 78.49%
Recall class 1: 73.34%
Recall class 0: 97.31%
F1: 83.33%
ROC AUC Training-> 91.07% Testing-> 89.95%
```

Plotting AUC Curve Characteristics

```
#plot learning curve
auc_plot(opt_knn)
```



The 0.8995 value of AUC defines that there is an 89.95% chance that the model will be able to distinguish between positive and negative classes for unseen datasets. The comparison for Train and Test AUC curves indicates little overfitting.

6. Artificial Neural Network (ANN)[15]

Artificial Neural Network (ANN) uses the processing of the brain as a basis to develop algorithms that can be used to model complex patterns and prediction problems.

Advantages	Disadvantages
ANNs can learn and model non-linear and complex relationships.	Hardware Dependence- Artificial Neural Networks require processors with parallel processing power, by their structure.
ANNs can generalize — After learning from the initial inputs and their relationships, it can infer unseen relationships on unseen data as well, thus making the model generalize and predict on unseen data.	There is no specific rule for determining the structure of artificial neural networks. The appropriate network structure is achieved through experience and trial and error.

[15] L Hardesty, '*Explained: neural networks*', MIT News, < https://news.mit.edu/2017/explained-neural-networks-deep-learning-0414 >, viewed 28 February 2022

Implementation

```
#import library for deep learning
import tensorflow as tf
from tensorflow import keras
from keras.layers import Flatten,Dense,Dropout,BatchNormalization, Conv1D,MaxPool1D
from keras.wrappers.scikit_learn import KerasClassifier
from keras.models import Sequential
from tensorflow.keras.optimizers import Adam
```

```
#build the model
ann=keras.Sequential([keras.layers.Dense(20,input_shape=(15,),activation='relu'),
                      keras.layers.Dense(1,activation='sigmoid'),])
ann.compile(optimizer='adam',loss='binary_crossentropy',metrics=['accuracy', tf.keras.metrics.AUC()])
#train the model
ann.fit(xtrain,ytrain, epochs=30)
```

```
#evaluate testing
testeval1 = ann.evaluate(xtest, ytest)
```

```
181/181 [==============================] - 0s 783us/step - loss: 0.3124 - accuracy: 0.8586 - auc: 0.9378
```

```
#evaluate training
traineval1 = ann.evaluate(xtrain, ytrain)
```

```
423/423 [==============================] - 0s 807us/step - loss: 0.3018 - accuracy: 0.8686 - auc: 0.9403
```

```
ytest[:10]
```

```
10529    1
15361    1
12160    1
2953     0
15870    1
11474    1
1730     0
12313    1
14851    1
15363    1
Name: is_fraud, dtype: int64
```

```
#make prediction
yprediction=ann.predict(xtest)
yprediction[:5]
```

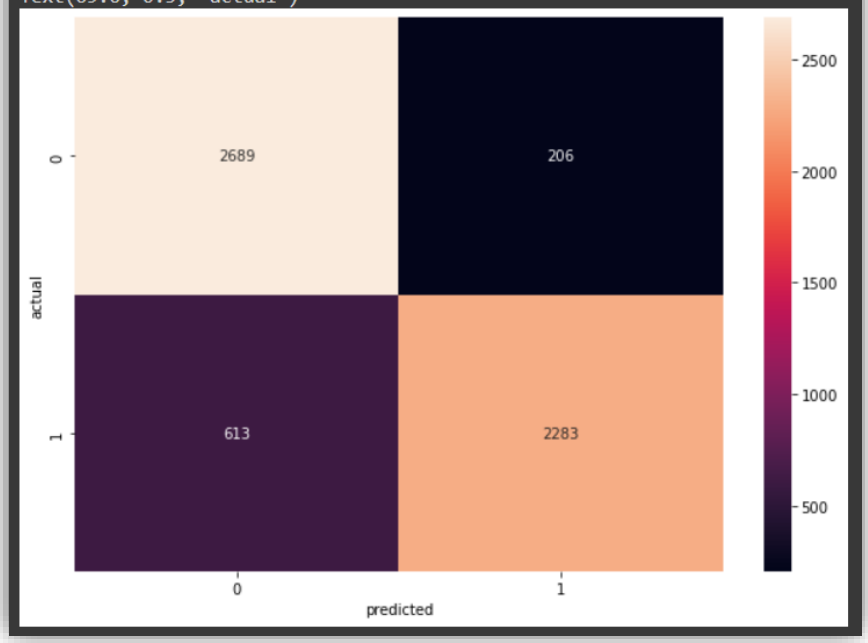
```
array([[0.9962951 ],
       [0.99891055],
       [0.9727628 ],
       [0.11689687],
       [0.5928722 ]], dtype=float32)
```

Confusion Matrix

For deep learning, we will be using the confusion_matrix library from TensorFlow. “The matrix columns represent the prediction labels and the rows represent the real labels. The confusion matrix is always a 2-D array of shapes [n, n], where n is the number of valid labels for a given classification task. Both prediction and labels must be 1-D arrays of the same shape for this function to work.”[16]

```
#plot confusion matrix
cmt =tf.math.confusion_matrix(labels=ytest,predictions=ypred)
plt.figure(figsize=(10,7))
sns.heatmap(cmt,annot=True,fmt='d')
plt.xlabel('predicted')
plt.ylabel('actual')
```

```
Text(69.0, 0.5, 'actual')
```



From the confusion matrix, we can see the True Negative is 2689 and the True positive is 2283. We will calculate the for the accuracy score: (2422+2393)/5791 = 0.8586. The accuracy score is 85.86%.

[16] tf.math.confusion_matrix, tensorflow.org <https://www.tensorflow.org/api_docs/python/tf/math/confusion_matrix>, viewed 28 February 2022

Evaluation

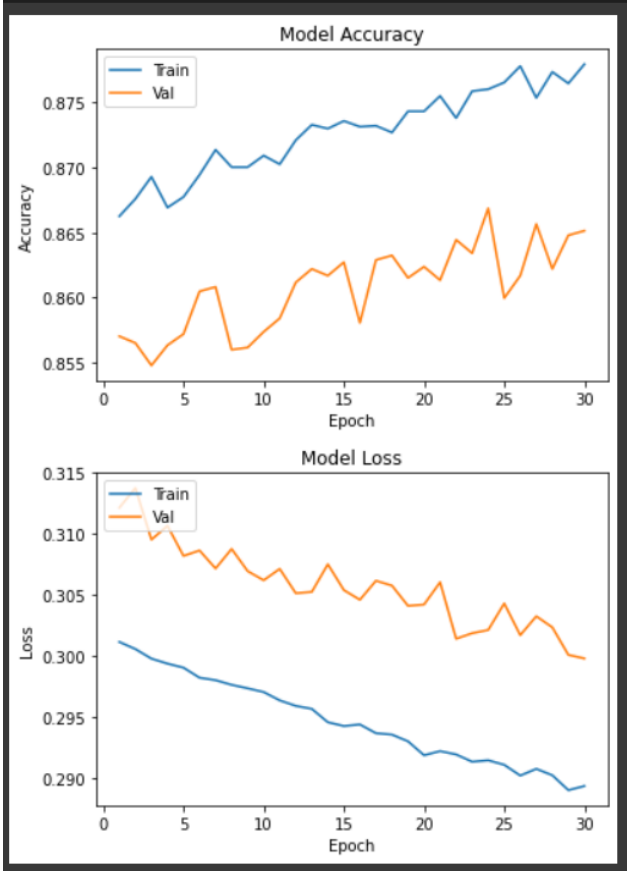
```
#print the metrics score
#pass algo, name, testeval score, traineval score, confusion matrix
calc(ann, 'ann', testeval1, traineval1, cmt)
```

```
Confusion Matrix Accuracy Score = 85.86%

Accuracy Score: Training -> 86.86% Testing -> 85.86%

Log Loss Training-> 0.3017820715904236 Testing -> 0.3123766779899597

Precision class 1: 91.72%
Precision class 0: 81.44%
Recall class 1: 78.83%
Recall class 0: 92.88%
F1: 84.79%
ROC AUC Training-> 94.03% Testing-> 93.78%
```

From the above, we can context that ANN holds 86.86% accuracy for unseen data while, 85.86% accuracy for seen data which can be considered overfitting.

Plotting Learning Curve

```
plot_learningCurve(history,30)
```



The above graph shows that with each epoch, the accuracy of the model is increased and loss is decreased.

ANN Hyperparameter Tuning

We will "explore the space of possible decisions automatically, systematically, in a principled way. The need to search for architecture spaces and find the best-performing ones empirically. That's what the field of automatic hyperparameter optimization is about: it's an entire field of research. The process of optimizing hyperparameters typically looks like this:

1. Choose a set of hyperparameters (automatically).
2. Build the corresponding model.
3. Fit it to your training data, and measure the final performance on the validation data.
4. Choose the next set of hyperparameters to try (automatically).
5. Repeat.
6. Eventually, measure performance on your test data."[17]

Grid Search will be used to find the best parameters for this model. It passes all combinations of hyperparameters one by one into the model and checks the result. Finally, it gives us the set of hyperparameters that gives the best result after passing in the model.

```
optimizer = ['SGD','Adadelta','RMSprop','Adagrad','Adam']
parameter_grid=dict(optimizer=optimizer)
grid= GridSearchCV(estimator=model,param_grid=parameter_grid,n_jobs=-1,cv=kfold)
grid_result=grid.fit(xtrain,ytrain)
```

```
print("Best:%f using %s" % (grid_result.best_score_,grid_result.best_params_))
```

```
Best:0.882465 using {'optimizer': 'Adam'}
```

```
#tuning epoch and batch size
def create_my_model1():
    mymodel=Sequential()
    mymodel.add(Dense(12,input_dim=15,activation='relu'))
    mymodel.add(Dense(1,activation='sigmoid'))
    mymodel.compile(loss='binary_crossentropy',optimizer='Adam',metrics=['accuracy',
                    tf.keras.metrics.AUC()])
    return mymodel
```

[17] F Chollet, '*Deep learning with python*', Getting the most out of your models, Manning, Shelter Island, 2018 p.263

```
def create_my_model(optimizer):
    mymodel=Sequential()
    mymodel.add(Dense(12,input_dim=15,activation='relu'))
    mymodel.add(Dense(1,activation='sigmoid'))
    mymodel.compile(loss='binary_crossentropy',optimizer=optimizer,metrics=['accuracy',
                                                                            tf.keras.metrics.AUC()])
    return mymodel

model=KerasClassifier(build_fn= create_my_model,epochs=50,batch_size=15)
```

```
model=KerasClassifier(build_fn= create_my_model1)
#defining the parameters
batchsize=[10,15,20,40,60,80,100]
epochs=[10,15,30,50]
parameter_grid=dict(batch_size=batchsize,epochs=epochs)
mygrid= GridSearchCV(estimator=model,param_grid=parameter_grid,n_jobs=-1,cv=kfold)
grid_result1=mygrid.fit(xtrain,ytrain)
```

The results from hyperparameter GridSearch.

```
print("Best:%f using %s" % (grid_result1.best_score_,grid_result1.best_params_))
Best:0.878616 using {'batch_size': 10, 'epochs': 50}
```

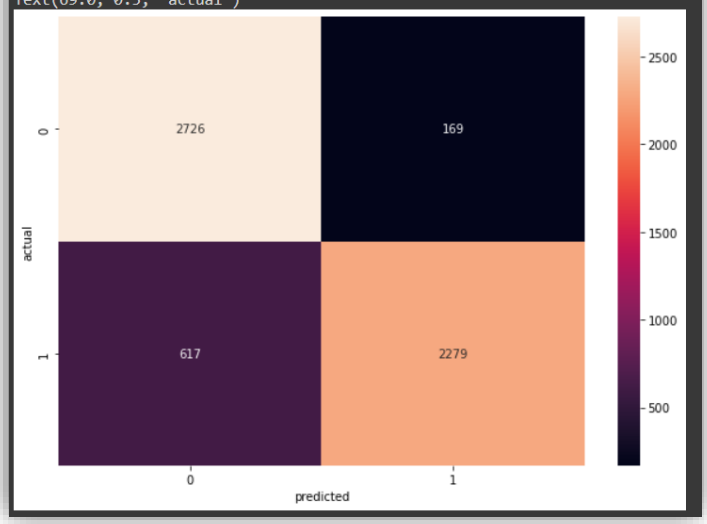
Model implementation using the best parameters.

```
opt_ann=create_my_model1()
#, epochs=50,batch_size=10)
opt_ann.fit(xtrain,ytrain, epochs = 50, batch_size = 10)
```

ANN confusion matrix after Tuning

```
#plot confusion matrix
cm1 =tf.math.confusion_matrix(labels=ytest,predictions=ypred1)
plt.figure(figsize=(10,7))
sns.heatmap(cm1,annot=True,fmt='d')
plt.xlabel('predicted')
plt.ylabel('actual')
```

```
Text(69.0, 0.5, 'actual')
```



We can see from the figure above the TN is 2726 while the TP is 2279, which increases almost 1% accuracy compared to the model before hyperparameter tuning.

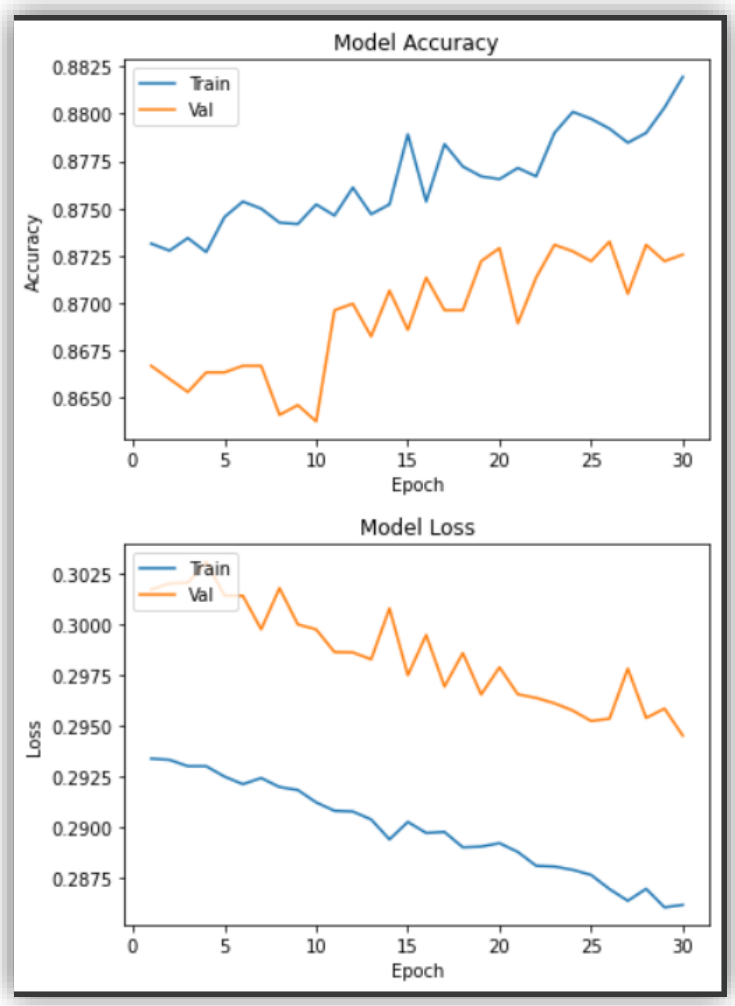
ANN model evaluation scores after hyperparameter tuning

```
calc(opt_ann, 'ann_tuned', testeval, traineval, cm1)
```

```
Confusion Matrix Accuracy Score = 86.43%

Accuracy Score: Training -> 87.25% Testing -> 86.43%

Log Loss Training-> 0.2928681969642639 Testing -> 0.30198410153388977

Precision class 1: 93.10%
Precision class 0: 81.54%
Recall class 1: 78.69%
Recall class 0: 94.16%
F1: 85.29%
ROC AUC Training-> 94.22% Testing-> 94.08%
```

The accuracy score before hyperparameter tuning is 85.86% with overfitting at 1%, meanwhile, it rises to 86.43% accuracy with overfitting reduced to 0.82%. This means the hyperparameter performs quite well and can be improved with deeper research.

Plotting Learning Curve



7. Convolutional Neural Networks (CNN)

CNN was first developed and used around the 1980s. The most that a CNN could do at that time was recognize handwritten digits. It was mostly used in the postal sectors to read zip codes, pin codes, etc. The important thing to remember about any deep learning model is that it requires a large amount of data to train and also requires a lot of computing resources. This was a major drawback for CNNs at that period hence CNNs were only limited to the postal sectors and they failed to enter the world of machine learning. “The convolution layer (CONV) uses filters that perform convolution operations as it is scanning the input II concerning its dimensions. Its hyperparameters include the filter size FF and stride SS. The resulting output OO is called feature map or activation map.”[18]

[18] A Amidi, S Amidi, *Convolutional Neural Networks cheatsheet*, Stanford.Edu, <https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks>, viewed 28 February 2022

Here are the advantages and disadvantages of CNN:

Advantages	Disadvantages
it automatically detects the important features without any human supervision	Hardware Dependence- requires processors with parallel processing power, by their structure.
CNN is also computationally efficient. It uses special convolution and pooling operations and performs parameter sharing. This enables CNN models to run on any device, making them universally attractive.	requires a large Dataset to process and train the neural network.

CNN Implementation

```
#build CNN
epochs=20
cnn=Sequential()
cnn.add(Conv1D(32,2, activation='relu',input_shape=xtrain[0].shape))
cnn.add(BatchNormalization())
cnn.add(Dropout(0.2))

cnn.add(Conv1D(64,2, activation='relu'))
cnn.add(BatchNormalization())
cnn.add(Dropout(0.5))

cnn.add(Flatten())
cnn.add(Dense(64,activation='relu'))
cnn.add(Dropout(0.5))

cnn.add(Dense(1,activation='sigmoid'))
```

```
cnn.summary()

Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv1d (Conv1D)             (None, 14, 32)            96

 batch_normalization (BatchN  (None, 14, 32)           128
 ormalization)

 dropout (Dropout)           (None, 14, 32)            0

 conv1d_1 (Conv1D)           (None, 13, 64)            4160

 batch_normalization_1 (Batc  (None, 13, 64)           256
 hNormalization)

 dropout_1 (Dropout)         (None, 13, 64)            0

 flatten (Flatten)           (None, 832)               0

 dense_6 (Dense)             (None, 64)                53312

 dropout_2 (Dropout)         (None, 64)                0

 dense_7 (Dense)             (None, 1)                 65

=================================================================
Total params: 58,017
Trainable params: 57,825
Non-trainable params: 192
_________________________________________________________________
```

Model evaluation score for testing and training dataset.

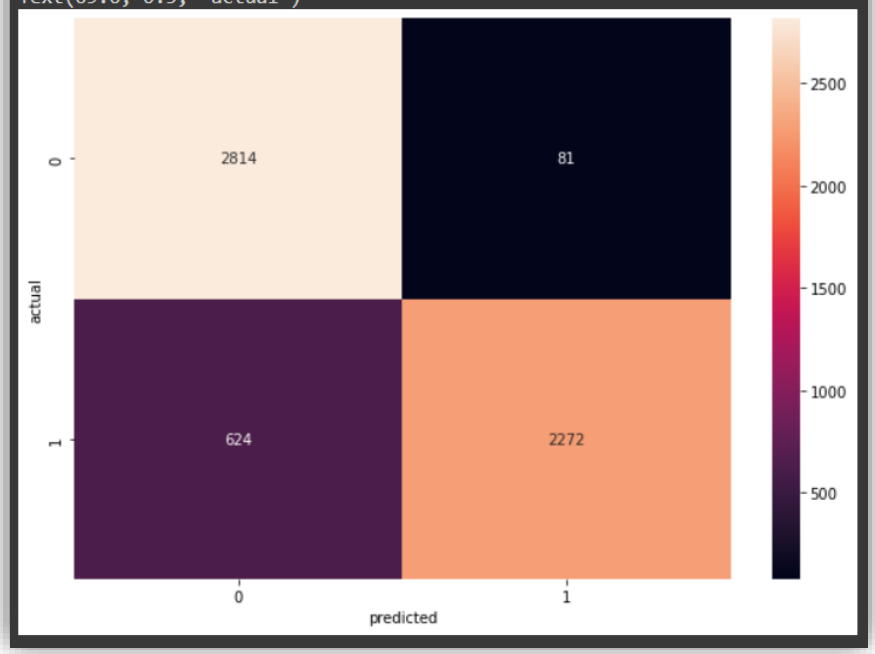
```
cntesteval = cnn.evaluate(xtest, ytest)
cntraineval = cnn.evaluate(xtrain,ytrain)
```

```
181/181 [==============================] - 0s 1ms/step - loss: 0.2906 - accuracy: 0.8783 - auc_3: 0.9527
423/423 [==============================] - 1s 1ms/step - loss: 0.2790 - accuracy: 0.8868 - auc_3: 0.9565
```

Confusion Matrix

```
#plot confusion matrix
cm2 =tf.math.confusion_matrix(labels=ytest,predictions=ypred2)
plt.figure(figsize=(10,7))
sns.heatmap(cm2,annot=True,fmt='d')
plt.xlabel('predicted')
plt.ylabel('actual')
```

```
Text(69.0, 0.5, 'actual')
```



From the confusion matrix, we can see the True Negative is 2814 and the True positive is 2272. We will calculate the for the accuracy score: (2814+2272)/5791 = 0.87.83. The accuracy score is 87.83%.

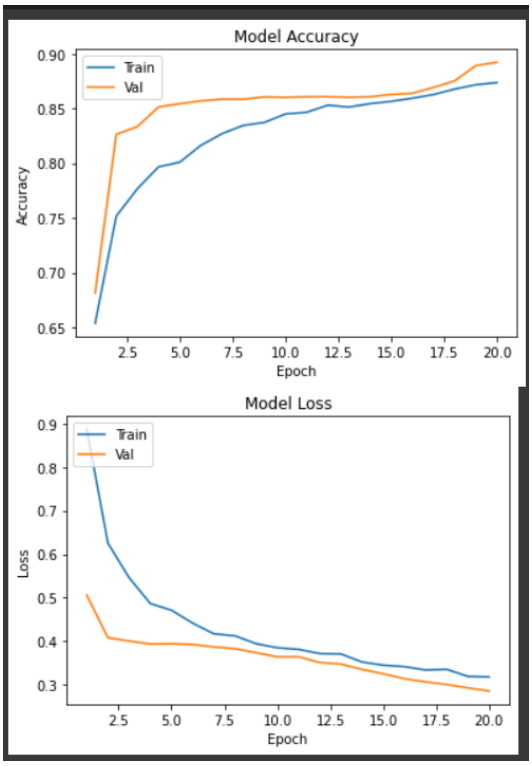
Evaluation

```
calc(cnn, 'cnn', cntesteval, cntraineval, cm2)
```

```
Confusion Matrix Accuracy Score = 87.83%

Accuracy Score: Training -> 88.68% Testing -> 87.83%

Log Loss Training-> 0.27900418639183044 Testing -> 0.290558785200119

Precision class 1: 96.56%
Precision class 0: 81.85%
Recall class 1: 78.45%
Recall class 0: 97.20%
F1: 86.57%
ROC AUC Training-> 95.65% Testing-> 95.27%
```

From the above, we can context that CNN holds 88.68% accuracy for unseen data while, 87.83% accuracy for seen data which can be considered overfitting.

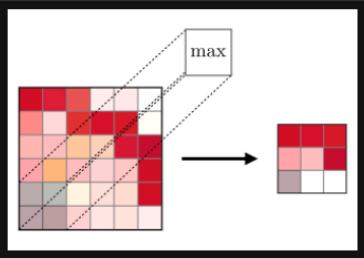
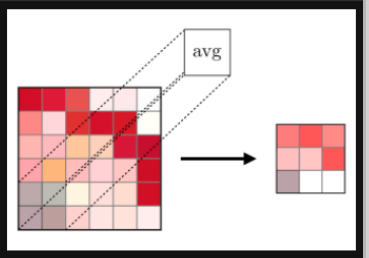
Plotting Learning Curve



The above graph shows that with each epoch, the accuracy of the model is increased and loss is decreased.

CNN Hyperparameter Tuning

To get the increase the accuracy of the CNN model, we added the Maxpool layer. “The pooling layer (POOL) is a downsampling operation, typically applied after a convolution layer, which does some spatial invariance. In particular, max and average pooling are special kinds of pooling where the maximum and average value is taken, respectively.”[19]

Type	Max pooling	Average pooling
Purpose	Each pooling operation selects the maximum value of the current view	Each pooling operation averages the values of the current view
Illustration		
Comments	• Preserves detected features • Most commonly used	• Downsamples feature map • Used in LeNet

```
#Adding MaxPool
epochs=50
cnnmax=Sequential()
cnnmax.add(Conv1D(32,2, activation='relu',input_shape=xtrain[0].shape))
cnnmax.add(BatchNormalization())
cnnmax.add(MaxPool1D(2))
cnnmax.add(Dropout(0.2))

cnnmax.add(Conv1D(64,2, activation='relu'))
cnnmax.add(BatchNormalization())
cnnmax.add(MaxPool1D(2))
cnnmax.add(Dropout(0.5))

cnnmax.add(Flatten())
cnnmax.add(Dense(64,activation='relu'))
cnnmax.add(Dropout(0.5))

cnnmax.add(Dense(1,activation='sigmoid'))
```

```
cnnmax.compile(optimizer=Adam(learning_rate=0.0001),loss='binary_crossentropy',metrics=['accuracy',
                                                                                        tf.keras.metrics.AUC()])
```

[19] A Amidi and S Amidi, *Convolutional Neural Networks cheatsheet*, Stanford.Edu, <https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks>, viewed 28 February 2022

Model evaluation after hyperparameter tuning.

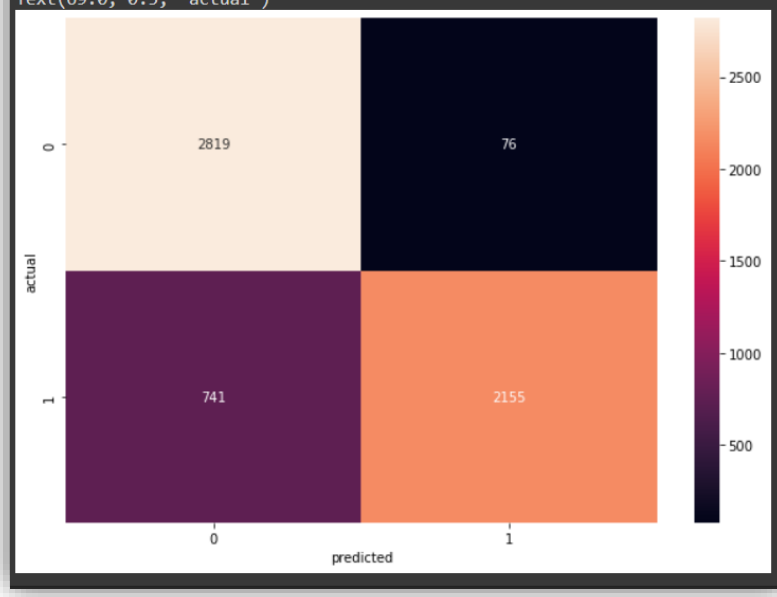
```
maxtesteval = cnnmax.evaluate(xtest, ytest)
maxtraineval = cnnmax.evaluate(xtrain,ytrain)
```

```
181/181 [==============================] - 0s 1ms/step - loss: 0.3516 - accuracy: 0.8589 - auc_4: 0.9112
423/423 [==============================] - 1s 1ms/step - loss: 0.3442 - accuracy: 0.8613 - auc_4: 0.9201
```

CNN confusion matrix after tuning

```
#plot confusion matrix
cm3 =tf.math.confusion_matrix(labels=ytest,predictions=ypred3)
plt.figure(figsize=(10,7))
sns.heatmap(cm3,annot=True,fmt='d')
plt.xlabel('predicted')
plt.ylabel('actual')
```

```
Text(69.0, 0.5, 'actual')
```



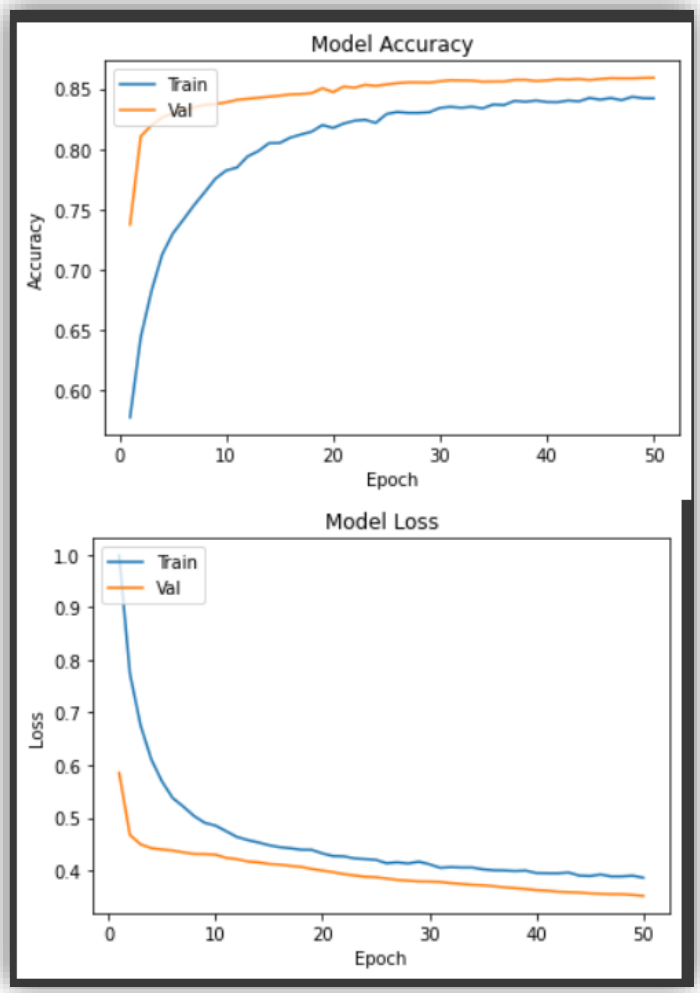
From the confusion matrix, we can see that the TN is 2819 and TP is 2155 and the accuracy score is 85.89%. It is slightly lower than the CNN before tuning almost 2%, however, the overfitting is reduced from 0.81% to 0.23%.

CNN model evaluation after tuning

```
calc(cnn, 'cnn_tuned', maxtesteval, maxtraineval, cm3)
```

```
Confusion Matrix Accuracy Score = 85.89%

Accuracy Score: Training -> 86.13% Testing -> 85.89%

Log Loss Training-> 0.34419187903404236 Testing -> 0.35158464312553406

Precision class 1: 96.59%
Precision class 0: 79.19%
Recall class 1: 74.41%
Recall class 0: 97.37%
F1: 84.06%
ROC AUC Training-> 92.01% Testing-> 91.12%
```

Plotting Learning Curve



ALGORITHM SELECTION

In the next steps, we will be evaluating the best model based on their evaluation scores.

algo	c_matrix	acc_test	acc_train	loss_test	loss_train	prec1	prec0	recall1	recall0	F1	roctest	roctrain
SVM1	86.237265	86.237265	86.958774	4.753489	4.504287e+00	97.126179	79.433221	74.689227	97.789292	84.442709	88.588715	92.059455
SVM_tuned	87.601451	87.601451	92.443194	4.282329	2.610035e+00	93.594876	83.054965	80.732044	94.473230	86.688914	93.316569	97.509331
Logreg	84.562252	84.562252	85.056621	5.332036	5.161286e+00	91.847826	79.435128	75.863260	93.264249	83.093797	84.313531	85.189295
logreg_tuned	85.097565	85.097565	85.559914	5.147139	4.987448e+00	93.645341	79.347198	75.310773	94.887737	83.483254	83.880607	84.703730
RF	95.475738	95.475738	99.992599	1.562635	2.556345e-03	97.340043	93.752077	93.508287	97.443869	95.385699	99.233849	100.000000
RF_tuned	96.235538	96.235538	100.000000	1.300209	9.992007e-16	97.482270	95.052171	94.924033	97.547496	96.186144	99.426211	100.000000
decTree	94.992229	94.992229	100.000000	1.729641	9.992007e-16	95.337509	94.652040	94.613260	95.371330	94.974003	94.992295	100.000000
decTree_tuned	95.130375	95.130375	96.935830	1.681929	1.058345e+00	95.100069	95.160733	95.165746	95.094991	95.132896	98.818685	99.688442
KNN	83.146261	83.146261	92.058323	5.821140	2.742989e+00	83.496162	82.803419	82.631215	83.661485	83.061437	88.977060	97.594230
knn_tuned	85.322051	85.322051	85.722744	5.069595	4.931200e+00	96.457766	78.489830	73.342541	97.305699	83.326795	89.951729	91.074684
ann	85.857365	85.857368	86.855155	0.312377	3.017821e-01	91.723584	81.435494	78.832873	92.884283	84.791086	93.776286	94.028568
ann_tuned	86.427215	86.427212	87.247425	0.301984	2.928682e-01	93.096405	81.543524	78.694751	94.162349	85.291916	94.079715	94.221348
cnn	87.825937	87.825936	88.683295	0.290559	2.790042e-01	96.557586	81.849913	78.453039	97.202073	86.568870	95.269221	95.649558
cnn_tuned	85.891901	85.891902	86.129820	0.351585	3.441919e-01	96.593456	79.185393	74.412983	97.374784	84.064755	91.116285	92.011243

Algorithms	Confusion Matrix	Test Accuracy	Train Accuracy	Test Loss	Train Loss	Precision 1	Precision 0	Recall 1	Recall 0	F1 Score	ROC Test	ROC Train
SVM1	86.24	86.24	86.96	4.75	4.50	97.13	79.43	74.69	97.79	84.44	88.59	92.06
SVM_tuned	87.60	87.60	92.44	4.28	2.61	93.59	83.05	80.73	94.47	86.69	93.32	97.51
Logreg	84.56	84.56	85.06	5.33	5.16	91.85	79.44	75.86	93.26	83.09	84.31	85.19
logreg_tuned	85.10	85.10	85.56	5.15	4.99	93.65	79.35	75.31	94.89	83.48	83.88	84.70
RF	95.48	95.48	99.99	1.56	0.00	97.34	93.75	93.51	97.44	95.39	99.23	100.00
RF_tuned	96.24	96.24	100.00	1.30	0.00	97.48	95.05	94.92	97.55	96.19	99.43	100.00
decTree	94.99	94.99	100.00	1.73	0.00	95.34	94.65	94.61	95.37	94.97	94.99	100.00
decTree_tuned	95.13	95.13	96.94	1.68	1.06	95.10	95.16	95.17	95.09	95.13	98.82	99.69
KNN	83.15	83.15	92.06	5.82	2.74	83.50	82.80	82.63	83.66	83.06	88.98	97.59
knn_tuned	85.32	85.32	85.72	5.07	4.93	96.46	78.49	73.34	97.31	83.33	89.95	91.07
ann	85.86	85.86	86.86	0.31	0.30	91.72	81.44	78.83	92.88	84.79	93.78	94.03
ann_tuned	86.43	86.43	87.25	0.30	0.29	93.10	81.54	78.69	94.16	85.29	94.08	94.22
cnn	87.83	87.83	88.68	0.29	0.28	96.56	81.85	78.45	97.20	86.57	95.27	95.65
cnn_tuned	85.89	85.89	86.13	0.35	0.34	96.59	79.19	74.41	97.37	84.06	91.12	92.01

From the table above we can see the model with the highest accuracy score is Random Forest after hyperparameter tuning equal to 96.23%, yet the overfitting percentage is 3.76%. This means that the model is not that optimal in performing against unseen data[20], however, we can consider it is improved compared to Random Forest before hyperparameter tuning which the overfitting is 4.51%. The second highest accuracy score is going to Decision Tree after hyperparameter tuning with 95.13% and the overfitting is 1.80% which we can conclude that the model is still not in the best performance. Although the overfitting is reduced compared to the model before hyperparameter tuning which is 5%.

[20] *Overfitting*, ibm.com, <https://www.ibm.com/cloud/learn/overfitting>, viewed 28 February 2022

If we sort the accuracy score from the highest to the lowest, it shows in the table below.

algo	c_matrix	acc_test	acc_train	Overfit	loss_test	loss_train	prec1	prec0	recall1	recall0	F1	roctest	roctrain
RF_tuned	96.24	96.24	100.00	3.76	1.30	0.00	97.48	95.05	94.92	97.55	96.19	99.43	100.00
RF	95.48	95.48	99.99	4.52	1.56	0.00	97.34	93.75	93.51	97.44	95.39	99.23	100.00
decTree_tune	95.13	95.13	96.94	1.81	1.68	1.06	95.10	95.16	95.17	95.09	95.13	98.82	99.69
decTree	94.99	94.99	100.00	5.01	1.73	0.00	95.34	94.65	94.61	95.37	94.97	94.99	100.00
cnn	87.83	87.83	88.68	0.86	0.29	0.28	96.56	81.85	78.45	97.20	86.57	95.27	95.65
SVM_tuned	87.60	87.60	92.44	4.84	4.28	2.61	93.59	83.05	80.73	94.47	86.69	93.32	97.51
ann_tuned	86.43	86.43	87.25	0.82	0.30	0.29	93.10	81.54	78.69	94.16	85.29	94.08	94.22
SVM1	86.24	86.24	86.96	0.72	4.75	4.50	97.13	79.43	74.69	97.79	84.44	88.59	92.06
cnn_tuned	85.89	85.89	86.13	0.24	0.35	0.34	96.59	79.19	74.41	97.37	84.06	91.12	92.01
ann	85.86	85.86	86.86	1.00	0.31	0.30	91.72	81.44	78.83	92.88	84.79	93.78	94.03
knn_tuned	85.32	85.32	85.72	0.40	5.07	4.93	96.46	78.49	73.34	97.31	83.33	89.95	91.07
logreg_tuned	85.10	85.10	85.56	0.46	5.15	4.99	93.65	79.35	75.31	94.89	83.48	83.88	84.70
Logreg	84.56	84.56	85.06	0.49	5.33	5.16	91.85	79.44	75.86	93.26	83.09	84.31	85.19
KNN	83.15	83.15	92.06	8.91	5.82	2.74	83.50	82.80	82.63	83.66	83.06	88.98	97.59

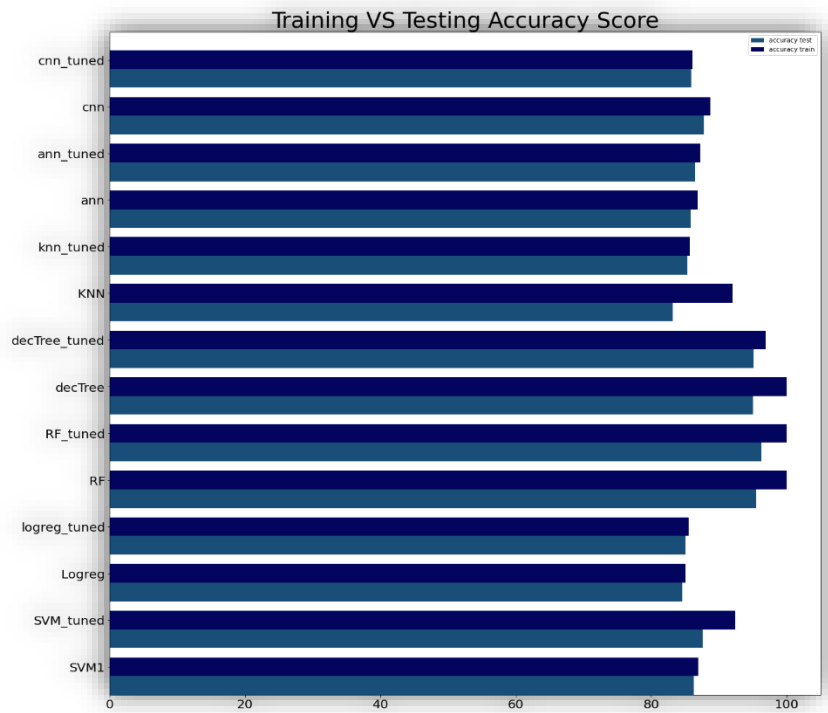
Instead of choosing the model with the highest accuracy yet the overfitting percentage above 1%, we will be choosing the model with a low overfitting percentage and considerable accuracy score. This narrow to Convolutional Neural Network with an accuracy score of 85.89% and overfitting is 0.24%, K-Nearest Neighbour with an accuracy score of 85.32% and overfitting is 0.4%, and the last is Logistic Regression with an accuracy score of 85.1% and overfitting is 0.46%. The details of the algorithm sorted by overfitting percentage from the lowest to the highest are shown in the table below.

algo	c_matrix	acc_test	acc_train	Overfit	loss_test	loss_train	prec1	prec0	recall1	recall0	F1	roctest	roctrain
cnn_tuned	85.89	85.89	86.13	0.24	0.35	0.34	96.59	79.19	74.41	97.37	84.06	91.12	92.01
knn_tuned	85.32	85.32	85.72	0.40	5.07	4.93	96.46	78.49	73.34	97.31	83.33	89.95	91.07
logreg_tuned	85.10	85.10	85.56	0.46	5.15	4.99	93.65	79.35	75.31	94.89	83.48	83.88	84.70
Logreg	84.56	84.56	85.06	0.49	5.33	5.16	91.85	79.44	75.86	93.26	83.09	84.31	85.19
SVM1	86.24	86.24	86.96	0.72	4.75	4.50	97.13	79.43	74.69	97.79	84.44	88.59	92.06
ann_tuned	86.43	86.43	87.25	0.82	0.30	0.29	93.10	81.54	78.69	94.16	85.29	94.08	94.22
cnn	87.83	87.83	88.68	0.86	0.29	0.28	96.56	81.85	78.45	97.20	86.57	95.27	95.65
ann	85.86	85.86	86.86	1.00	0.31	0.30	91.72	81.44	78.83	92.88	84.79	93.78	94.03
decTree_tune	95.13	95.13	96.94	1.81	1.68	1.06	95.10	95.16	95.17	95.09	95.13	98.82	99.69
RF_tuned	96.24	96.24	100.00	3.76	1.30	0.00	97.48	95.05	94.92	97.55	96.19	99.43	100.00
RF	95.48	95.48	99.99	4.52	1.56	0.00	97.34	93.75	93.51	97.44	95.39	99.23	100.00
SVM_tuned	87.60	87.60	92.44	4.84	4.28	2.61	93.59	83.05	80.73	94.47	86.69	93.32	97.51
decTree	94.99	94.99	100.00	5.01	1.73	0.00	95.34	94.65	94.61	95.37	94.97	94.99	100.00
KNN	83.15	83.15	92.06	8.91	5.82	2.74	83.50	82.80	82.63	83.66	83.06	88.98	97.59

Plotting Testing and Training Accuracy

"The accuracy_score function computes the accuracy, either the fraction (default) or the count (normalize=False) of correct predictions. In multilabel classification, the function returns the subset accuracy. If the entire set of predicted labels for a sample strictly match with the true set of labels, then the subset accuracy is 1.0 otherwise, it is 0.0. If $\hat{y}_i$ is the predicted value of the i-th sample and y_i is the corresponding true value, then the fraction of correct predictions over $n_{samples}$ is defined as:"[21]

$$accuracy(y, \hat{y}) = \frac{1}{n_{samples}} \sum_{i=0}^{n_{samples}-1} 1(\hat{y}_i = y_i)$$



From the chart above, Random Forest and Decision Tree showed Perfect Accuracy for the training data (seen data). This is a sign of overfitting.

Logistic Regression, KNN, ANN and CNN models showed relatively higher accuracy in unseen data. However, testing accuracy for KNN and ANN decreases slightly when for the other models.

[21] Accuracy score, scikit-learn.org, <https://scikit-learn.org/stable/modules/model_evaluation.html#accuracy-score>, viewed 28 February 2022

Plotting Testing and Training Loss

"Log loss, aka logistic loss or cross-entropy loss is the loss function used in (multinomial) logistic regression and extensions of it such as neural networks, defined as the negative log-likelihood of a logistic model that returns y_pred probabilities for its training data y_true. The log loss is only defined for two or more labels. For a single sample with true label $y \in \{0,1\}$ and a probability estimate $p = \Pr(y = 1)$, the log loss formula is:"[22]

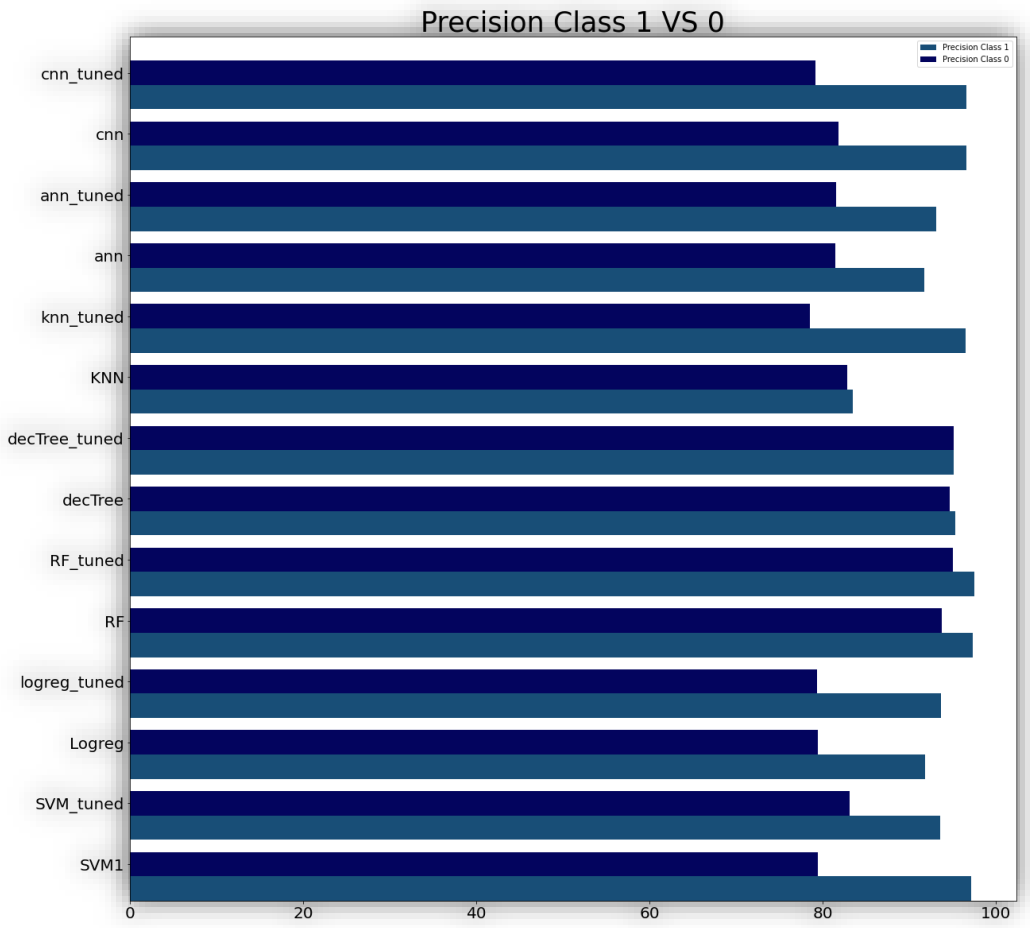
$$L_{\log}(y, p) = -(y \log(p) + (1 - y) \log(1 - p))$$

From the figure above, we can see that Decision Tree (before tuning) and Random Forests have 0 log loss during the training phase whereas high during the testing phase. In contrast to Machine Learning algorithms, Deep Learning such as CNN and ANN have a lower log loss.

[22] *sklearn.metrics.log_loss*, scikit-learn.org, <https://scikit-learn.org/stable/modules/generated/sklearn.metrics.log_loss.html>, viewed 28 February 2022

Plotting Precision Class 0 and Class 1

"The precision is the ratio $\frac{TP}{(TP+FP)}$ where TP is the number of true positives and FP the number of false positives. The precision is intuitively the ability of the classifier not to label as positive a sample that is negative. The best value is 1 (100%) and the worst value is 0."[23]

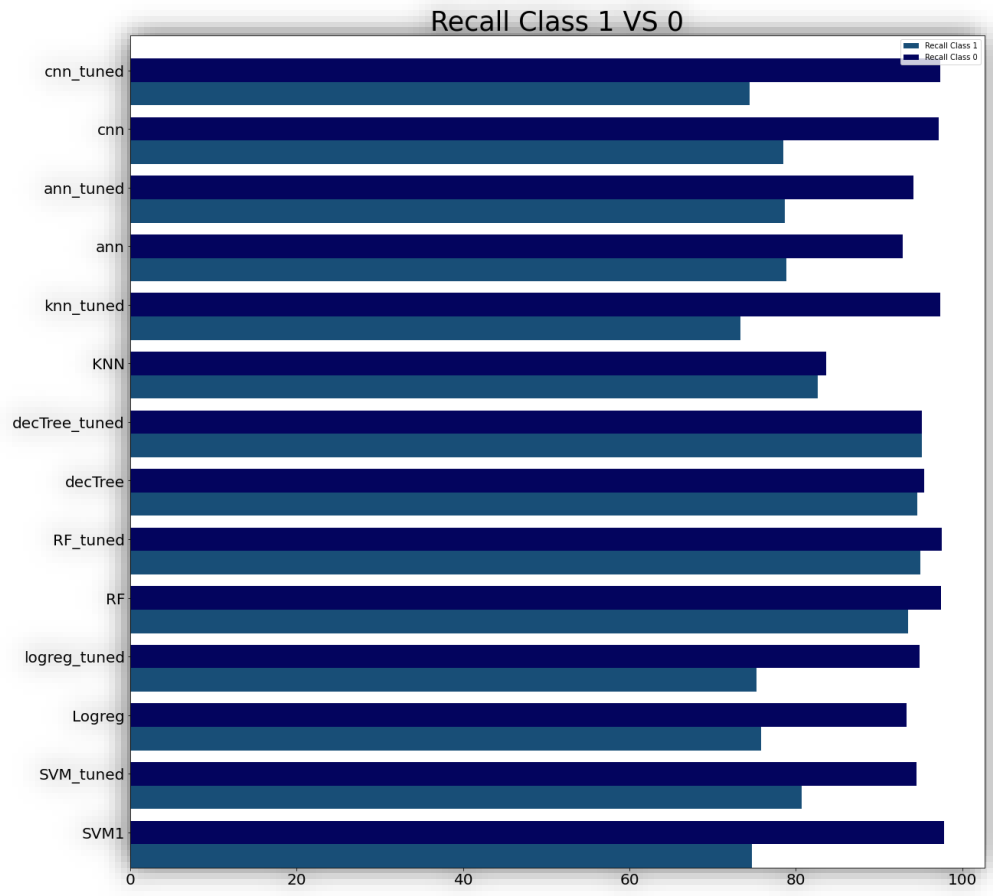


From the chart above, we can see that the Decision Tree has a balance precision score for class 1 and class 0.

[23] *sklearn.metrics.precision_score*, scikit-learn.org, < https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_score.html?highlight=precision#sklearn.metrics.precision_score>, viewed 28 February 2022

Plotting Recall Class 0 and Class 1

"The recall is the ratio $\frac{TP}{(TP+FN)}$ where tp is the number of true positives and fn the number of false negatives. The recall is intuitively the ability of the classifier to find all the positive samples. The best value is 1 and the worst value is 0."[24]

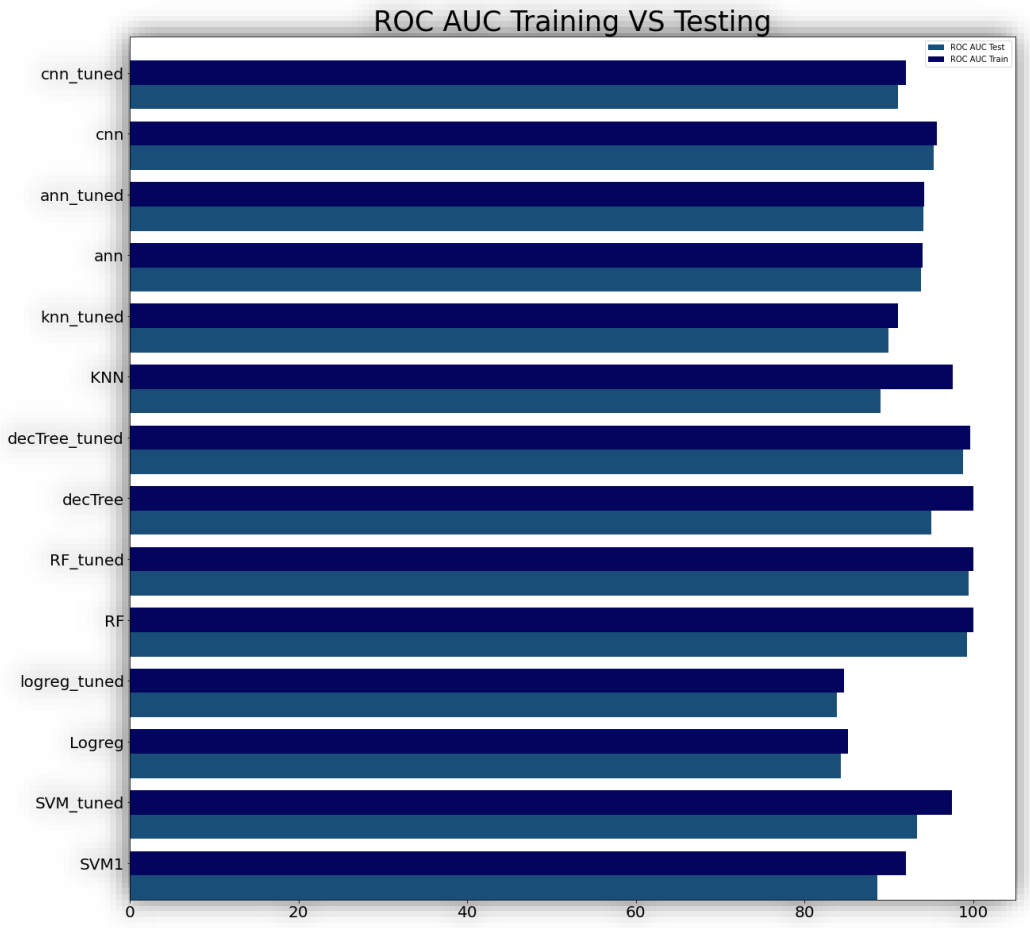


From the chart above, we can see that the Decision Tree also has a balance recall score for class 1 and class 0.

[24] *sklearn.metrics.recall_score*, scikit-learn.org, < https://scikit-learn.org/stable/modules/generated/sklearn.metrics.recall_score.html#sklearn.metrics.recall_score>, viewed 28 February 2022

Plotting ROC AUC Training Vs Testing

"ROC analysis is a useful tool for evaluating the performance of diagnostic tests and more generally for evaluating the accuracy of a statistical model (e.g., logistic regression, linear discriminant analysis) that classifies subjects into 1 of 2 categories."[25]



From the chart above, we can see that Random Forest has the best ROC AUC score in the training and testing phase compared to other algorithms.

[25] K H Zou, A J O'Malley, and L Mauri, *Receiver-Operating Characteristic Analysis for Evaluating Diagnostic Tests and Predictive Models*, 2007, <https://www.ahajournals.org/doi/10.1161/CIRCULATIONAHA.105.594929>, viewed 28 February 2022

CONCLUSION

Seven algorithms from Machine Learning and Deep Learning were conducted to determine the most predictive classifier for credit card fraud detection. These projects constructed from two datasets and combine into one, with extreme imbalance classification problems that solved using Imblearn's Random Under Sampling to make the class is balanced. Some models have a high accuracy score; however, the overfitting cannot be avoided. For example, Decision Tree and Random Forest have a high accuracy score of more than 90% yet their overfitting percentage is almost 5%. In contrast to Decision Tree and Random Forest, KNN has the lowest accuracy score which is 83.15% and the overfitting percentage is the highest with 8.91%.

Hyperparameter tuning and K-Fold cross-validation are applied to cope with the overfitting problem and to improve the accuracy. As a result, the overfit in KNN dropped from 8.91% to 0.40% by increasing the number of Neighbour and some changes in the parameters, the accuracy score is also increased around 2% to 85.32%. Similar to KNN, the overfitting percentage also reduced from 5.01% to 1.81% for the Decision Tree and the accuracy score is slightly rise.

For future benefits, we selected three algorithms with the lowest percentage of overfitting as they will generalise well to new data. If the model can generalise the data well, it also could perform the classification or prediction task that was intended for. Consequently, we picked KNN, Logistic Regression, and Convolutional Neural Networks as our final algorithm for the credit card fraud detection problem.