IBM NAAN MUDHALVAN

APPLIED DATA SCIENCE

Project: Credit Card Fraud Detection

Phase 2

Credit card fraud is any dishonest act and behaviour to obtain information without the proper authorization from the account holder for financial gain. Among different ways of frauds, Skimming is the most common one, which is the way of duplicating of information located on the magnetic strip of the card. Apart from this, the other ways are:

Manipulation/alteration of genuine cards

Creation of counterfeit cards

Stolen/lost credit cards

Fraudulent telemarketing

Data Understanding: Here, we need to load the data and understand the features present in it. This would help us choose the features that we will need for your final model.

Exploratory data analytics (EDA): Normally, in this step, we need to perform univariate and bivariate analyses of the data, followed by feature transformations, if necessary. For the current data set, because Gaussian variables are used, we do not need to perform Z-scaling. However, you can check if there is any skew-ness in the data and try to mitigate it, as it might cause problems during the model-building phase.

Train/Test Split: Now we are familiar with the train/test split, which we can perform in order to check the performance of our models with unseen data. Here, for validation, we can use the k-fold cross-validation method. We need to choose an appropriate k value so that the minority class is correctly represented in the test folds.

Model-Building/ hyper-parameter Tuning: This is the final step at which we can try different models and fine-tune their hyper-parameters until we get the desired level of performance on the given dataset. We should try and see if we get a better model by the various sampling techniques.

Model Evaluation: We need to evaluate the models using appropriate evaluation metrics. Note that since the data is imbalanced it is more important to identify which are fraudulent transactions accurately than the non-fraudulent. We need to choose an appropriate evaluation metric which reflects this business goal.

Source code

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

% matplotlib inline

import seaborn as sns

import warnings

warnings.filterwarnings('ignore')

pd.set option('display.max\_columns', 500)

*# Reading the dataset*

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

df**.**head()

df**.**shape

df**.**info()

df**.**describe()

*# Cheking percent of missing values in columns*

df\_missing\_columns **=** (round(((df**.**isnull()**.**sum()**/**len(df**.**index))**\***100),2)**.**to\_frame('null'))**.**sort\_values('null', ascending**=False**)

df\_missing\_columns

**Checking the distribution of the classes**

classes **=** df['Class']**.**value\_counts()

classes

normal\_share **=** round((classes[0]**/**df['Class']**.**count()**\***100),2)

normal\_share

fraud\_share **=** round((classes[1]**/**df['Class']**.**count()**\***100),2)

fraud\_share

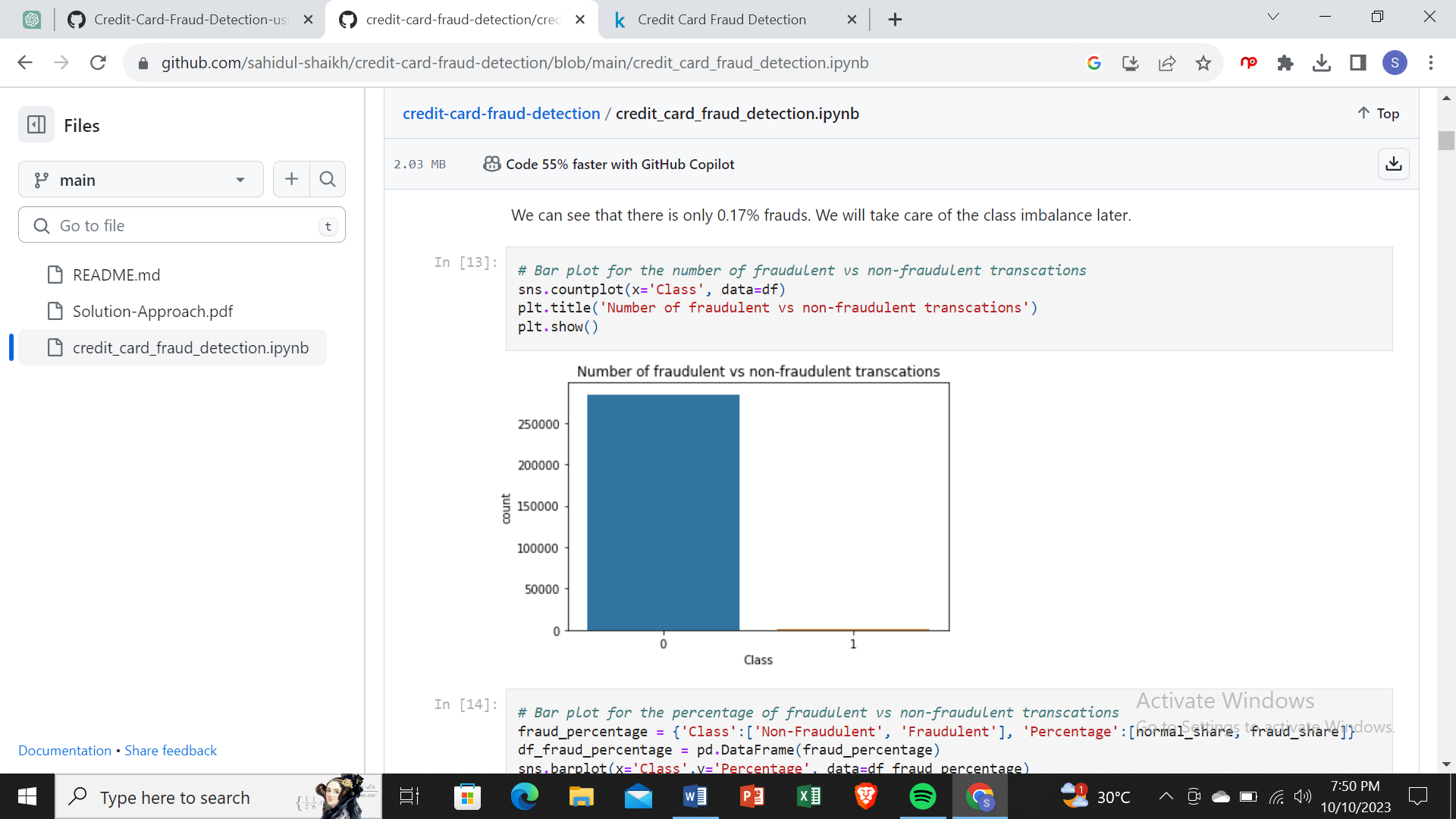
The dataset can be download using this: https://www.kaggle.com/mlg-ulb/creditcardfraud

*# Bar plot for the number of fraudulent vs non-fraudulent transcations*

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

plt**.**title('Number of fraudulent vs non-fraudulent transcations')

plt**.**show()



*# Bar plot for the percentage of fraudulent vs non-fraudulent transcations*

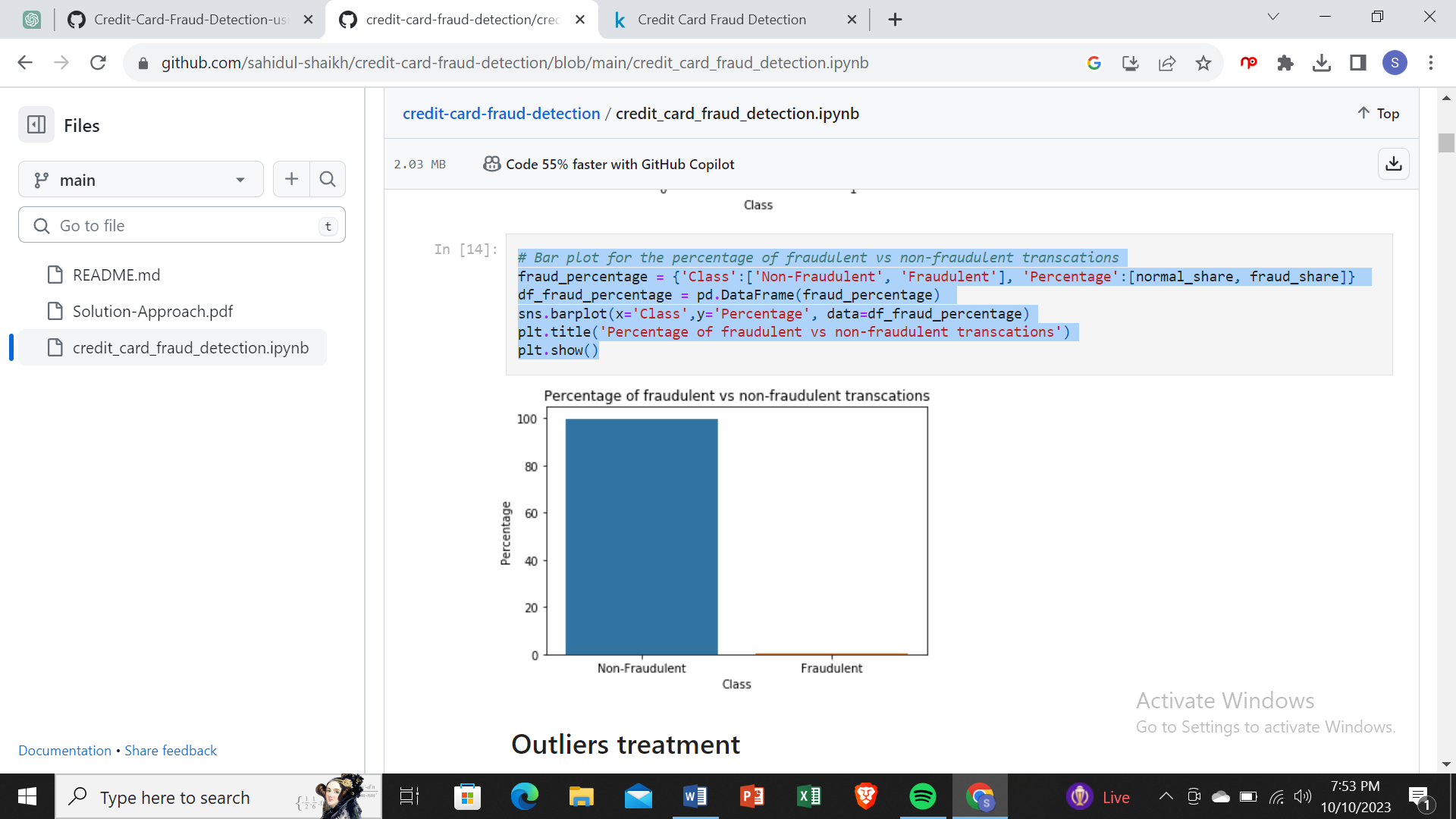
fraud\_percentage **=** {'Class':['Non-Fraudulent', 'Fraudulent'], 'Percentage':[normal\_share, fraud\_share]}

df\_fraud\_percentage **=** pd**.**DataFrame(fraud\_percentage)

sns**.**barplot(x**=**'Class',y**=**'Percentage', data**=**df\_fraud\_percentage)

plt**.**title('Percentage of fraudulent vs non-fraudulent transcations')

plt**.**show()



# Creating fraudulent dataframe

data\_fraud = df[df['Class'] == 1]

# Creating non fraudulent dataframe

data\_non\_fraud = df[df['Class'] == 0]

# Distribution plot

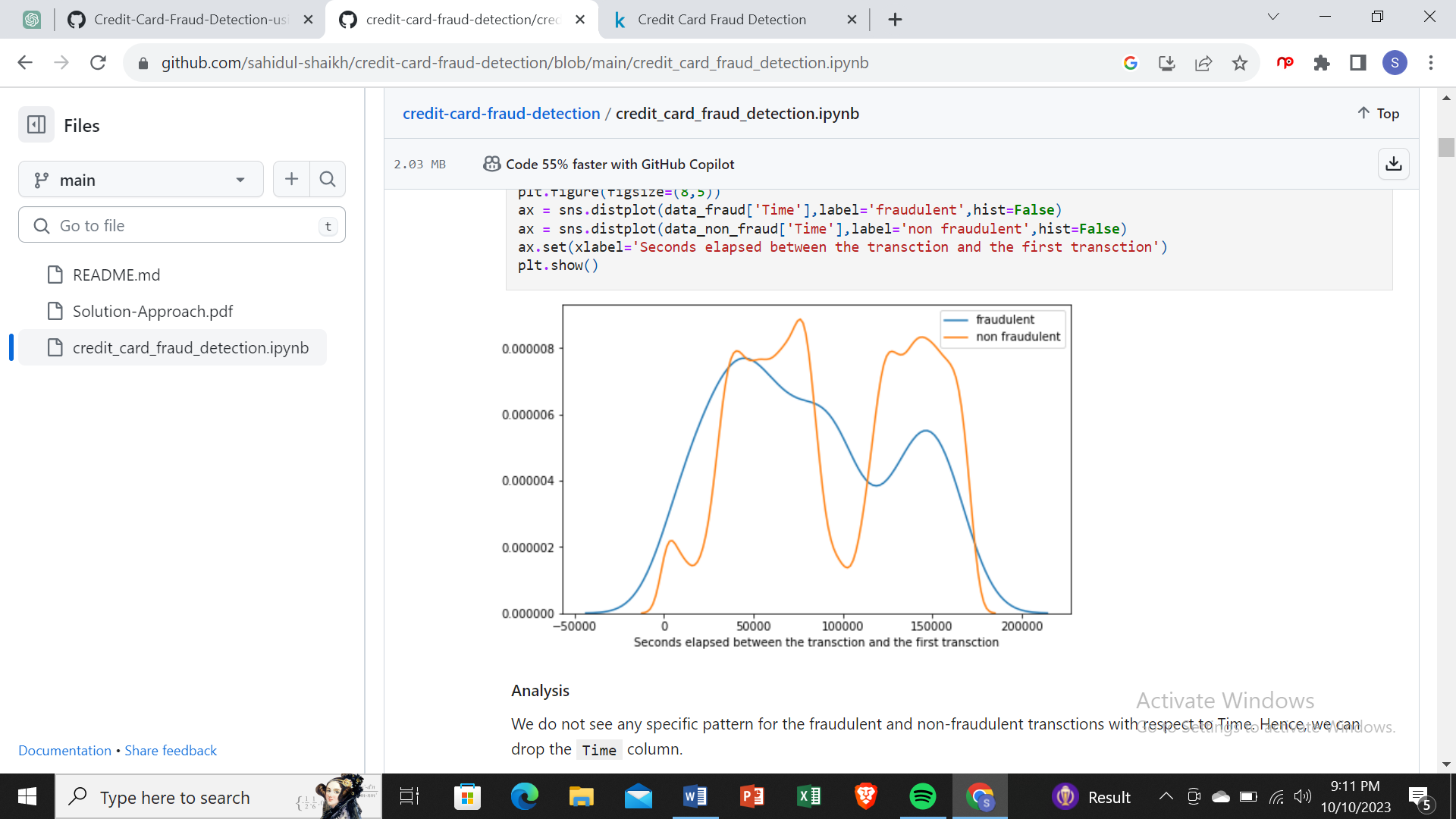
plt.figure(figsize=(8,5))

ax = sns.distplot(data\_fraud['Time'],label='fraudulent',hist=False)

ax = sns.distplot(data\_non\_fraud['Time'],label='non fraudulent',hist=False)

ax.set(xlabel='Seconds elapsed between the transction and the first transction')

plt.show()



Analysis

We do not see any specific pattern for the fraudulent and non-fraudulent transctions with respect to Time. Hence, we can drop the Time column.

# Dropping the Time column

df.drop('Time', axis=1, inplace=True)

Observe the distribution of classes with amount

# Distribution plot

plt.figure(figsize=(8,5))

ax = sns.distplot(data\_fraud['Amount'],label='fraudulent',hist=False)

ax = sns.distplot(data\_non\_fraud['Time'],label='non fraudulent',hist=False)

ax.set(xlabel='Transction Amount')

plt.show()

Analysis

We can see that the fraudulent transactions are mostly dense in the lower range of amount, whereas the non-fraudulent transactions are spreaded throughout low to high range of amount.

# Model building on imbalanced data

### Metric selection for heavily imbalanced data

As we have seen that the data is heavily imbalanced, where only 0.17% transaction are fraudulent, we should not consider Accuracy as a good measure for evaluating the model. Because in the case of all the data points return a particular class (1/0) irrespective of any prediction, still the model will result more than 99% Accuracy.

Hence, we have to measure the ROC-AUC score for fair evaluation of the model. The ROC curve is used to understand the strength of the model by evaluating the performance of the model at all the classification thresholds. The default threshold of 0.5 is not always the ideal threshold to find the best classification label of the test point. Because the ROC curve is measured at all thresholds, the best threshold would be one at which the TPR is high and FPR is low, i.e., misclassifications are low. After determining the optimal threshold, we can calculate the F1 score of the classifier to measure the precision and recall at the selected threshold.

#### Why SVM was not tried for model building and Random Forest was not tried for few cases?

In the dataset we have 284807 data points and in the case of Oversampling we would have even more number of data points. SVM is not very efficient with large number of data points because it takes lot of computational power and resources to make the transformation. When we perform the cross validation with K-Fold for hyper-parameter tuning, it takes lot of computational resources and it is very time consuming. Hence, because of the unavailability of the required resources and time SVM was not tried.

For the same reason Random forest was also not tried for model building in few of the hyper-parameter tuning for oversampling technique.

#### Why KNN was not used for model building?

KNN is not memory efficient. It becomes very slow as the number of data points increases as the model needs to store all the data points. It is computationally heavy because for a single data point the algorithm has to calculate the distance of all the data points and find the nearest neighbors.

### Logistic regression

*# Importing scikit logistic regression module*

**from** sklearn. Linear \_ model **import** Logistic Regression

*# Importing metrics*

**from** sklearn **import** metrics

**from** sklearn.metrics **import** confusion\_matrix

**from** sklearn.metrics **import** f1\_score

**from** sklearn.metrics **import** classification\_report

#### Tuning hyperparameter C

C is the the inverse of regularization strength in Logistic Regression. Higher values of C correspond to less regularization.

*# Importing libraries for cross validation*

**from** sklearn.model\_selection **import** KFold

**from** sklearn.model\_selection **import** cross\_val\_score

**from** sklearn.model\_selection **import** GridSearchCV

*# Creating KFold object with 5 splits*

folds **=** KFold(n\_splits**=**5, shuffle**=True**, random\_state**=**4)

*# Specify params*

params **=** {"C": [0.01, 0.1, 1, 10, 100, 1000]}

*# Specifing score as recall as we are more focused on acheiving the higher sensitivity than the accuracy*

model\_cv **=** GridSearchCV(estimator **=** LogisticRegression(),

param\_grid **=** params,

scoring**=** 'roc\_auc',

cv **=** folds,

verbose **=** 1,

return\_train\_score**=True**)

*# Fit the model*

model\_cv**.**fit(X\_train, y\_train)

Fitting 5 folds for each of 6 candidates, totalling 30 fits

[Parallel(n\_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.

[Parallel(n\_jobs=1)]: Done 30 out of 30 | elapsed: 43.0s finished

GridSearchCV(cv=KFold(n\_splits=5, random\_state=4, shuffle=True),

error\_score=nan,

estimator=LogisticRegression(C=1.0, class\_weight=None, dual=False,

fit\_intercept=True,

intercept\_scaling=1, l1\_ratio=None,

max\_iter=100, multi\_class='auto',

n\_jobs=None, penalty='l2',

random\_state=None, solver='lbfgs',

tol=0.0001, verbose=0,

warm\_start=False),

iid='deprecated', n\_jobs=None,

param\_grid={'C': [0.01, 0.1, 1, 10, 100, 1000]},

pre\_dispatch='2\*n\_jobs', refit=True, return\_train\_score=True,

scoring='roc\_auc', verbose=1)

cv\_results **=** pd**.**DataFrame(model\_cv**.**cv\_results

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

lt**.**figur

e(figsize**=**(8, 6))

plt**.**plot(cv\_results['param\_C'], cv\_results['mean\_test\_score'])

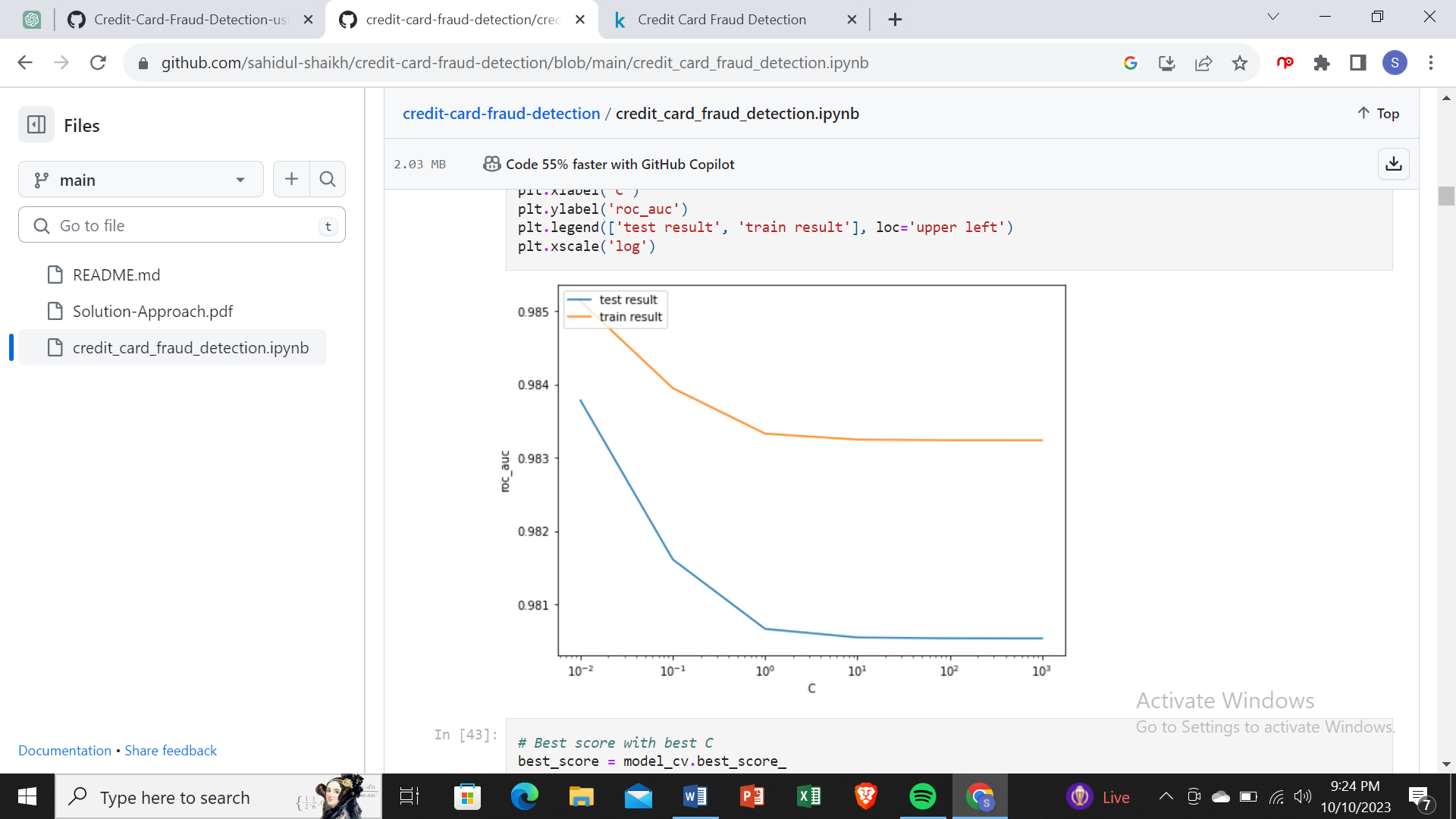
plt**.**plot(cv\_results['param\_C'], cv\_results['mean\_train\_score'])

plt**.**xlabel('C')

plt**.**ylabel('roc\_auc')

plt**.**legend(['test result', 'train result'], loc**=**'upper left')

plt**.**xscale('log')



# Best score with best C

best\_score = model\_cv.best\_score\_

best\_C = model\_cv.best\_params\_['C']

print(" The highest test roc\_auc is {0} at C = {1}".format(best\_score, best\_C))

The highest test roc\_auc is 0.9837811907775487 at C = 0.01

Logistic regression with optimal C

# Instantiate the model with best C

logistic\_imb = LogisticRegression(C=0.01)

# Fit the model on the train set

logistic\_imb\_model = logistic\_imb.fit(X\_train, y\_train)

Prediction on the train set

# Predictions on the train set

y\_train\_pred = logistic\_imb\_model.predict(X\_train)

# Confusion matrix

confusion = metrics.confusion\_matrix(y\_train, y\_train\_pred)

print(confusion)

[[227427 22]

[ 135 261]]

TP = confusion[1,1] # true positive

TN = confusion[0,0] # true negatives

FP = confusion[0,1] # false positives

FN = confusion[1,0] # false negatives

# Accuracy

print("Accuracy:-",metrics.accuracy\_score(y\_train, y\_train\_pred))

# Sensitivity

print("Sensitivity:-",TP / float(TP+FN))

# Specificity

print("Specificity:-", TN / float(TN+FP))

# F1 score

print("F1-Score:-", f1\_score(y\_train, y\_train\_pred))

Accuracy:- 0.9993109350655051

Sensitivity:- 0.6590909090909091

Specificity:- 0.9999032750198946

F1-Score:- 0.7687776141384388

# classification\_report

print(classification\_report(y\_train, y\_train\_pred))

precision recall f1-score support

0 1.00 1.00 1.00 227449

1 0.92 0.66 0.77 396

accuracy 1.00 227845

macro avg 0.96 0.83 0.88 227845

weighted avg 1.00 1.00 1.00 227845

ROC on the train set

# ROC Curve function

def draw\_roc( actual, probs ):

fpr, tpr, thresholds = metrics.roc\_curve( actual, probs,

drop\_intermediate = False )

auc\_score = metrics.roc\_auc\_score( actual, probs )

plt.figure(figsize=(5, 5))

plt.plot( fpr, tpr, label='ROC curve (area = %0.2f)' % auc\_score )

plt.plot([0, 1], [0, 1], 'k--')

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate or [1 - True Negative Rate]')

plt.ylabel('True Positive Rate')

plt.title('Receiver operating characteristic example')

plt.legend(loc="lower right")

plt.show()

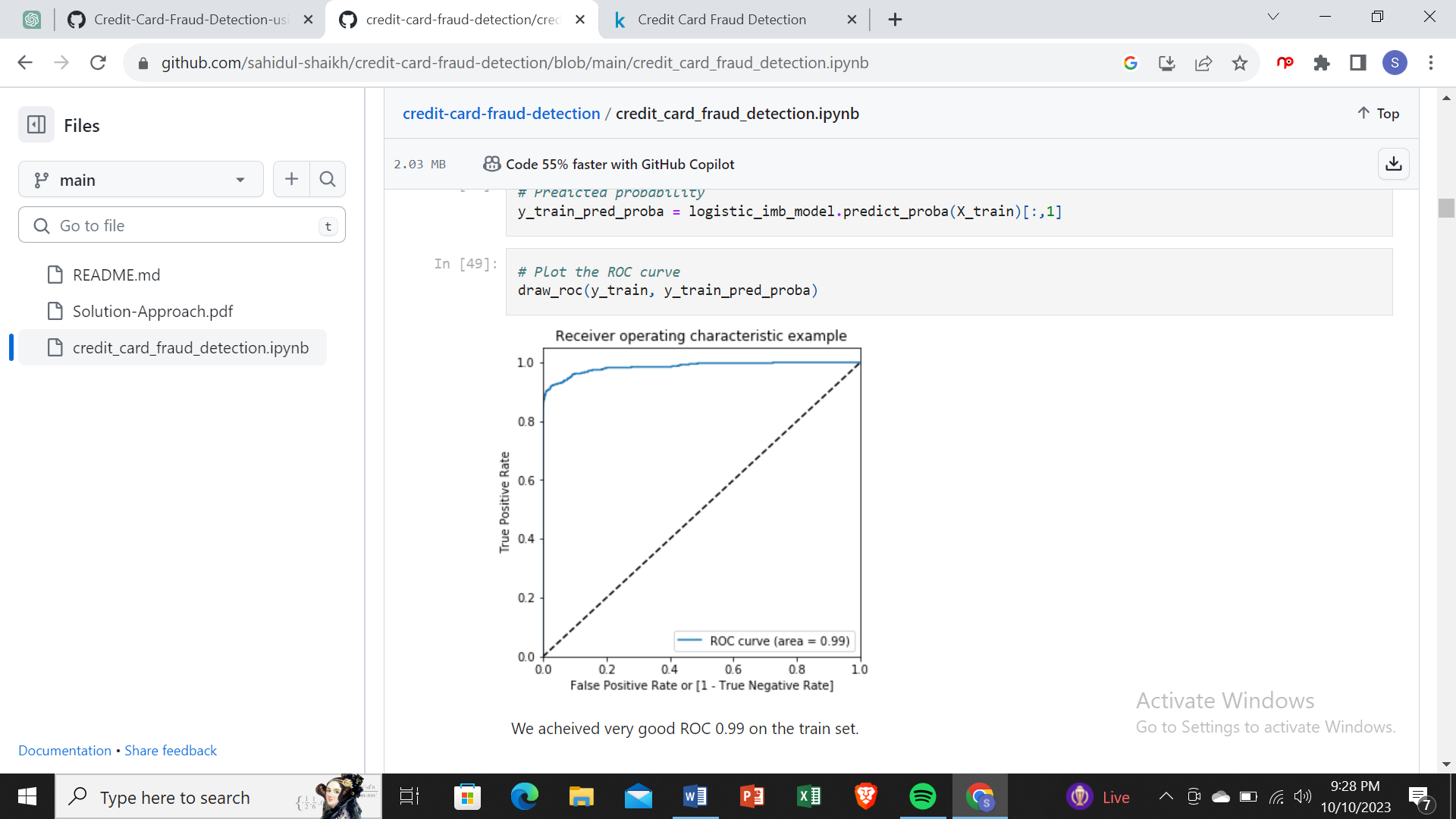
return None

# Predicted probability

y\_train\_pred\_proba = logistic\_imb\_model.predict\_proba(X\_train)[:,1]

# Plot the ROC curve

draw\_roc(y\_train, y\_train\_pred\_proba)



Prediction on the test set

# Prediction on the test set

y\_test\_pred = logistic\_imb\_model.predict(X\_test)

# Confusion matrix

confusion = metrics.confusion\_matrix(y\_test, y\_test\_pred)

print(confusion)

[[56850 16]

[ 42 54]]

TP = confusion[1,1] # true positive

TN = confusion[0,0] # true negatives

FP = confusion[0,1] # false positives

FN = confusion[1,0] # false negatives

# Accuracy

print("Accuracy:-",metrics.accuracy\_score(y\_test, y\_test\_pred))

# Sensitivity

print("Sensitivity:-",TP / float(TP+FN))

# Specificity

print("Specificity:-", TN / float(TN+FP))

# F1 score

print("F1-Score:-", f1\_score(y\_test, y\_test\_pred))

Accuracy:- 0.9989817773252344

Sensitivity:- 0.5625

Specificity:- 0.9997186367952731

F1-Score:- 0.6506024096385543

# classification\_report

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

*# classification\_report*

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

precision recall f1-score support

0 1.00 1.00 1.00 56866

1 0.77 0.56 0.65 96

accuracy 1.00 56962

macro avg 0.89 0.78 0.83 56962

weighted avg 1.00 1.00 1.00 56962

ROC on the test set

# Predicted probability

y\_test\_pred\_proba = logistic\_imb\_model.predict\_proba(X\_test)[:,1]

# Plot the ROC curve

draw\_roc(y\_test, y\_test\_pred\_proba)

We can see that we have very good ROC on the test set 0.97, which is almost close to 1.

\*Model summary\*

Train set

Accuracy = 0.99

Sensitivity = 0.70

Specificity = 0.99

F1-Score = 0.76

ROC = 0.99

Test set

Accuracy = 0.99

Sensitivity = 0.77

Specificity = 0.99

F1-Score = 0.65

ROC = 0.97

Overall, the model is performing well in the test set, what it had learnt from the train set.

XGBoost

# Importing XGBoost

from xgboost import XGBClassifier

Tuning the hyperparameters

# hyperparameter tuning with XGBoost

# creating a KFold object

folds = 3

# specify range of hyperparameters

param\_grid = {'learning\_rate': [0.2, 0.6],

'subsample': [0.3, 0.6, 0.9]}

# specify model

xgb\_model = XGBClassifier(max\_depth=2, n\_estimators=200)

# set up GridSearchCV()

model\_cv = GridSearchCV(estimator = xgb\_model,

param\_grid = param\_grid,

scoring= 'roc\_auc',

cv = folds,

verbose = 1,

return\_train\_score=True)

# fit the model

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

Fitting 3 folds for each of 6 candidates, totalling 18 fits

[Parallel(n\_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.

[Parallel(n\_jobs=1)]: Done 18 out of 18 | elapsed: 12.6min finished

GridSearchCV(cv=3, error\_score=nan,

estimator=XGBClassifier(base\_score=0.5, booster='gbtree',

colsample\_bylevel=1, colsample\_bynode=1,

colsample\_bytree=1, gamma=0,

learning\_rate=0.1, max\_delta\_step=0,

max\_depth=2, min\_child\_weight=1,

missing=None, n\_estimators=200, n\_jobs

nthread=None, objective='binary:logistic

random\_state=0,reg\_alpha=0,reg\_lambda=1,

scale\_pos\_weight=1, seed=None, silentne,

subsample=1, verbosity=1),

iid='deprecated', n\_jobs=None,

param\_grid={'learning\_rate': [0.2, 0.6],

'subsample': [0.3, 0.6, 0.9]},

pre\_dispatch='2\*n\_jobs', refit=True, return\_train\_score=True,

scoring='roc\_auc', verbose=1)

# cv results

cv\_results = pd.DataFrame(model\_cv.cv\_results\_)

cv\_results

…………………..

# # plotting

plt.figure(figsize=(16,6))

param\_grid = {'learning\_rate': [0.2, 0.6],

'subsample': [0.3, 0.6, 0.9]}

for n, subsample in enumerate(param\_grid['subsample']):

# subplot 1/n

plt.subplot(1,len(param\_grid['subsample']), n+1)

df = cv\_results[cv\_results['param\_subsample']==subsample]

plt.plot(df["param\_learning\_rate"], df["mean\_test\_score"])

plt.plot(df["param\_learning\_rate"], df["mean\_train\_score"])

plt.xlabel('learning\_rate')

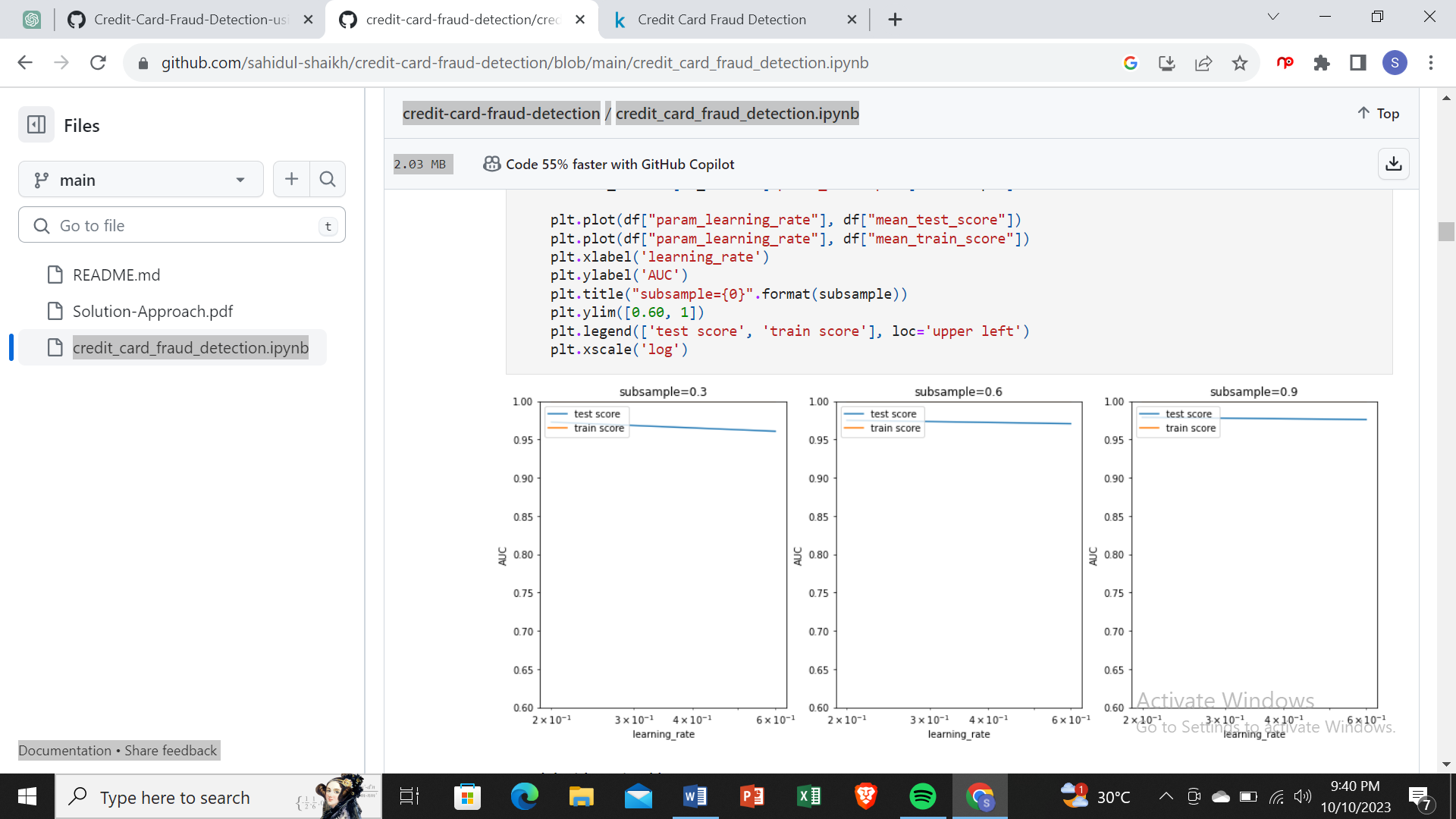
plt.ylabel('AUC')

plt.title("subsample={0}".format(subsample))

plt.ylim([0.60, 1])

plt.legend(['test score', 'train score'], loc='upper left')

plt.xscale('log')



Model with optimal hyperparameters

We see that the train score almost touches to 1. Among the hyperparameters, we can choose the best parameters as learning\_rate : 0.2 and subsample: 0.3

model\_cv.best\_params\_

{'learning\_rate': 0.2, 'subsample': 0.9}

# chosen hyperparameters

# 'objective':'binary:logistic' outputs probability rather than label, which we need for calculating auc

params = {'learning\_rate': 0.2,

'max\_depth': 2,

'n\_estimators':200,

'subsample':0.9,

'objective':'binary:logistic'}

# fit model on training data

xgb\_imb\_model = XGBClassifier(params = params)

xgb\_imb\_model.fit(X\_train, y\_train)

XGBClassifier(base\_score=0.5, booster=None, colsample\_bylevel=1,

colsample\_bynode=1, colsample\_bytree=1, gamma=0, gpu\_id=-1,

importance\_type='gain', interaction\_constraints=None,

learning\_rate=0.300000012, max\_delta\_step=0, max\_depth=6,

min\_child\_weight=1, missing=nan, monotone\_constraints=None,

n\_estimators=100, n\_jobs=0, num\_parallel\_tree=1,

objective='binary:logistic',

params={'learning\_rate': 0.2, 'max\_depth': 2, 'n\_estimators': 200,

'objective': 'binary:logistic', 'subsample': 0.9},

random\_state=0, reg\_alpha=0, reg\_lambda=1, scale\_pos\_weight=1,

subsample=1, tree\_method=None, validate\_parameters=False,

verbosity=None)

Prediction on the train set

# Predictions on the train set

y\_train\_pred = xgb\_imb\_model.predict(X\_train)

# Confusion matrix

confusion = metrics.confusion\_matrix(y\_train, y\_train\_pred)

print(confusion)

[[227449 0]

[ 0 396]]

TP = confusion[1,1] # true positive

TN = confusion[0,0] # true negatives

FP = confusion[0,1] # false positives

FN = confusion[1,0] # false negatives

# Accuracy

print("Accuracy:-",metrics.accuracy\_score(y\_train, y\_train\_pred))

# Sensitivity

print("Sensitivity:-",TP / float(TP+FN))

# Specificity

print("Specificity:-", TN / float(TN+FP))

# F1 score

print("F1-Score:-", f1\_score(y\_train, y\_train\_pred))

Accuracy:- 1.0

Sensitivity:- 1.0

Specificity:- 1.0

F1-Score:- 1.0

# classification\_report

print(classification\_report(y\_train, y\_train\_pred))

precision recall f1-score support

0 1.00 1.00 1.00 227449

1 1.00 1.00 1.00 396

accuracy 1.00 227845

macro avg 1.00 1.00 1.00 227845

weighted avg 1.00 1.00 1.00 227845

# Predicted probability

y\_train\_pred\_proba\_imb\_xgb = xgb\_imb\_model.predict\_proba(X\_train)[:,1]

# roc\_auc

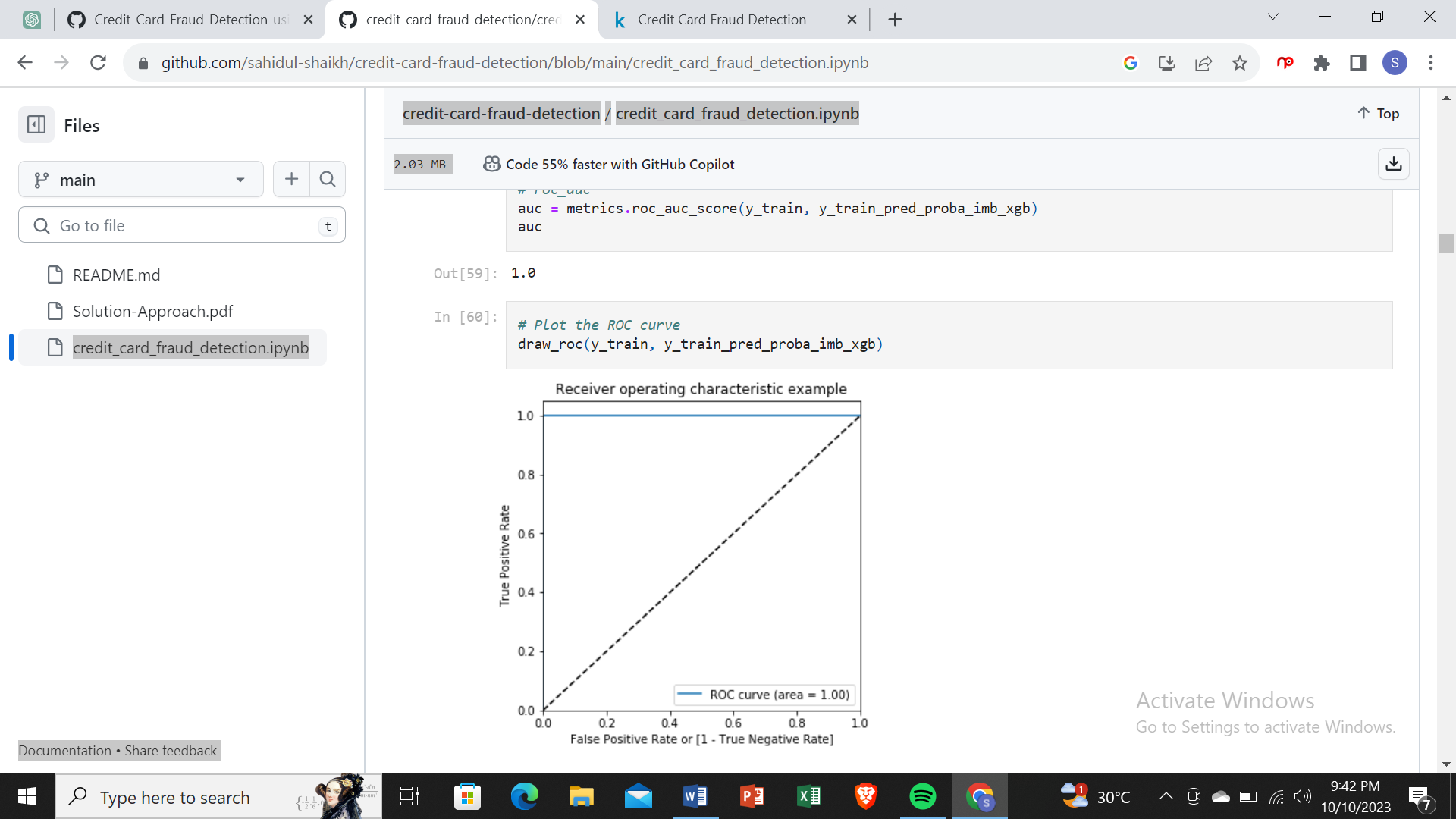
auc = metrics.roc\_auc\_score(y\_train, y\_train\_pred\_proba\_imb\_xgb)

auc

1.0

# Plot the ROC curve

draw\_roc(y\_train, y\_train\_pred\_proba\_imb\_xgb)



Prediction on the test set

# Predictions on the test set

y\_test\_pred = rfc\_imb\_model.predict(X\_test)

# Confusion matrix

confusion = metrics.confusion\_matrix(y\_test, y\_test\_pred)

print(confusion)

[[56841 25]

[ 36 60]]

TP = confusion[1,1] # true positive

TN = confusion[0,0] # true negatives

FP = confusion[0,1] # false positives

FN = confusion[1,0] # false negatives

# Accuracy

print("Accuracy:-",metrics.accuracy\_score(y\_test, y\_test\_pred))

# Sensitivity

print("Sensitivity:-",TP / float(TP+FN))

# Specificity

print("Specificity:-", TN / float(TN+FP))

# F1 score

print("F1-Score:-", f1\_score(y\_train, y\_train\_pred))

Accuracy:- 0.9989291106351603

Sensitivity:- 0.625

Specificity:- 0.9995603699926142

F1-Score:- 0.7983761840324763

# classification\_report

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

precision recall f1-score support

0 1.00 1.00 1.00 56866

1 0.71 0.62 0.66 96

accuracy 1.00 56962

macro avg 0.85 0.81 0.83 56962

weighted avg 1.00 1.00 1.00 56962

# Predicted probability

y\_test\_pred\_proba = rfc\_imb\_model.predict\_proba(X\_test)[:,1]

# roc\_auc

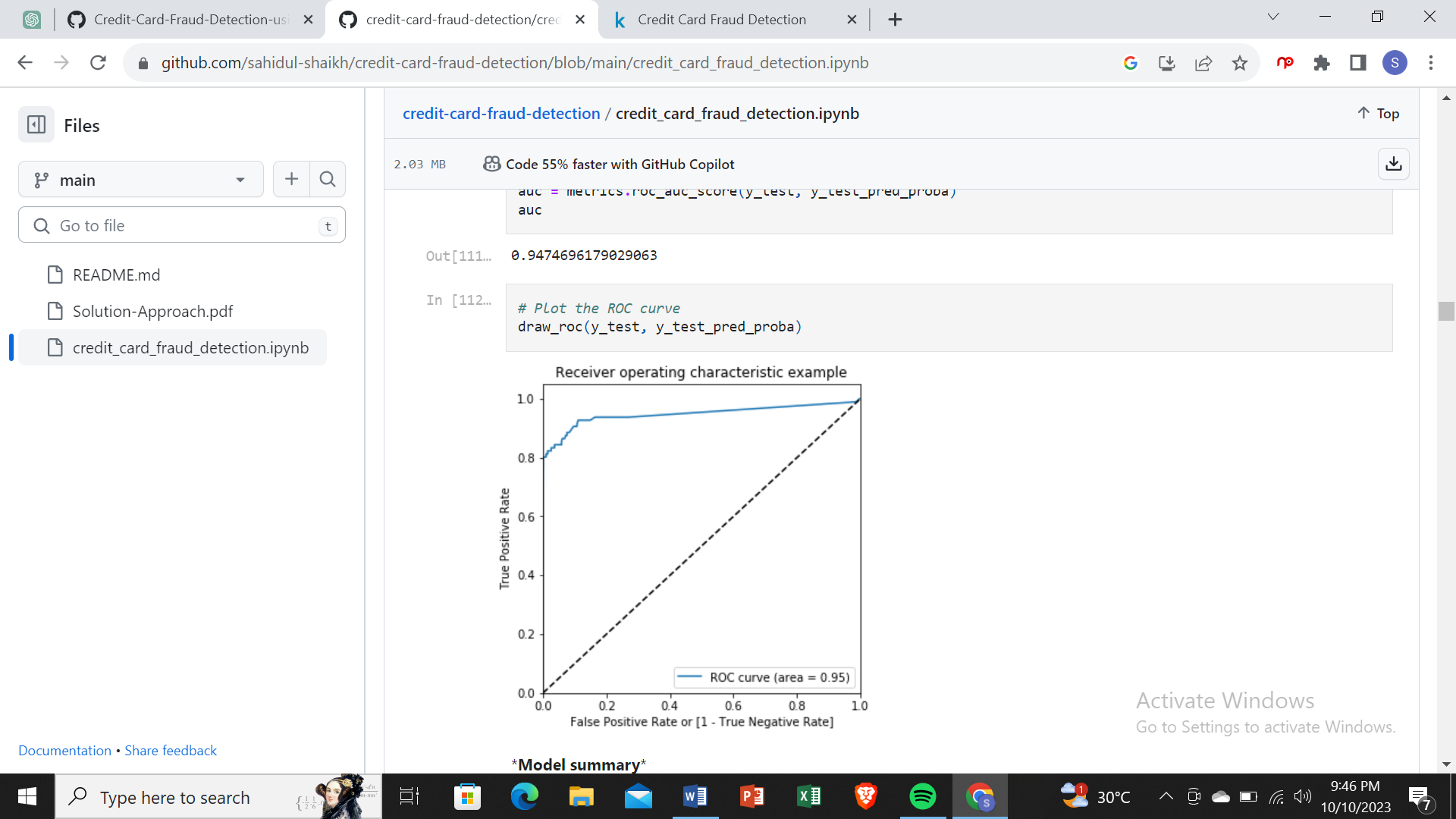
auc = metrics.roc\_auc\_score(y\_test, y\_test\_pred\_proba)

auc

0.9474696179029063

# Plot the ROC curve

draw\_roc(y\_test, y\_test\_pred\_proba)



\*Model summary\*

Train set

Accuracy = 0.99

Sensitivity = 1.0

Specificity = 1.0

F1-Score = 0.80

ROC-AUC = 0.98

Test set

Accuracy = 0.99

Sensitivity = 0.62

Specificity = 0.99

F-1 Score = 0.75

ROC-AUC = 0.96

Choosing best model on the imbalanced data

We can see that among all the models we tried (Logistic, XG-Boost, Decision Tree, and Random Forest), almost all of them have performed well. More specifically logistic regression and XG-Boost performed best in terms of ROC-AUC score.

But as we have to choose one of them, we can go for the best as XG-Boost, which gives us ROC score of 1.0 on the train data and 0.98 on the test data.

Keep in mind that XG-Boost requires more resource utilization than Logistic model. Hence building XG-Boost model is more costly than the Logistic model. But XG-Boost having ROC score 0.98, which is 0.01 more than the Logistic model. The 0.01 increase of score may convert into huge amount of saving for the bank.

Print the important features of the best model to understand the dataset

This will not give much explanation on the already transformed dataset

But it will help us in understanding if the dataset is not PCA transformed

# Features of XG-Boost model

var\_imp = []

for i in xgb\_imb\_model.feature\_importances\_:

var\_imp.append(i)

print('Top var =', var\_imp.index(np.sort(xgb\_imb\_model.feature\_importances\_)[-1])+1)

print('2nd Top var =', var\_imp.index(np.sort(xgb\_imb\_model.feature\_importances\_)[-2])+1)

print('3rd Top var =', var\_imp.index(np.sort(xgb\_imb\_model.feature\_importances\_)[-3])+1)

# Variable on Index-16 and Index-13 seems to be the top 2 variables

top\_var\_index = var\_imp.index(np.sort(xgb\_imb\_model.feature\_importances\_)[-1])

second\_top\_var\_index = var\_imp.index(np.sort(xgb\_imb\_model.feature\_importances\_)[-2])

X\_train\_1 = X\_train.to\_numpy()[np.where(y\_train==1.0)]

X\_train\_0 = X\_train.to\_numpy()[np.where(y\_train==0.0)]

np.random.shuffle(X\_train\_0)

import matplotlib.pyplot as plt

%matplotlib inline

plt.rcParams['figure.figsize'] = [20, 20]

plt.scatter(X\_train\_1[:, top\_var\_index], X\_train\_1[:, second\_top\_var\_index], label='Actual Class-1 Examples')

plt.scatter(X\_train\_0[:X\_train\_1.shape[0], top\_var\_index], X\_train\_0[:X\_train\_1.shape[0], second\_top\_var\_index],

label='Actual Class-0 Examples')

plt.legend()

Top var = 17

2nd Top var = 14

3rd Top var = 10

<matplotlib.legend.Legend at 0x11887c88>

Print the FPR,TPR & select the best threshold from the roc curve for the best model

print('Train auc =', metrics.roc\_auc\_score(y\_train, y\_train\_pred\_proba\_imb\_xgb))

fpr, tpr, thresholds = metrics.roc\_curve(y\_train, y\_train\_pred\_proba\_imb\_xgb)

threshold = thresholds[np.argmax(tpr-fpr)]

print("Threshold=",threshold)

Train auc = 1.0

Threshold= 0.8474788

We can see that the threshold is 0.85, for which the TPR is the highest and FPR is the lowest and we got the best ROC score.

Handling data imbalance

As we see that the data is heavily imbalanced. We will try several approaches for handling data imbalance.

Under-Sampling:- Here for balancing the class distribution, the non-fraudulent transaction count will be reduced to 396 (similar count of fraudulent transactions.)

Over-Sampling:- Here we will make the same count of non-fraudulent transaction as fraudulent transactions.

SMOTE: Synthetic minority oversampling technique. It is another oversampling technique, which uses nearest neighbor algorithm to create synthetic data.

ADSAYAN: This is similar to SMOTE with minor changes that the new synthetic data is generated on the region of low density of imbalanced data points.

Undersampling

# Importing undersampler library

from imblearn.under\_sampling import RandomUnderSampler

from collections import Counter

# instantiating the random undersampler

rus = RandomUnderSampler()

# resampling X, y

X\_train\_rus, y\_train\_rus = rus.fit\_resample(X\_train, y\_train)

# Befor sampling class distribution

print('Before sampling class distribution:-',Counter(y\_train))

# new class distribution

print('New class distribution:-',Counter(y\_train\_rus))

Before sampling class distribution:- Counter({0: 227449, 1: 396})

New class distribution:- Counter({0: 396, 1: 396})

Model building on balanced data with Undersampling

Logistic Regression

# Creating KFold object with 5 splits

folds = KFold(n\_splits=5, shuffle=True, random\_state=4)

# Specify params

params = {"C": [0.01, 0.1, 1, 10, 100, 1000]}

# Specifing score as roc-auc

model\_cv = GridSearchCV(estimator = LogisticRegression(),

param\_grid = params,

scoring= 'roc\_auc',

cv = folds,

verbose = 1,

return\_train\_score=True)

# Fit the model

model\_cv.fit(X\_train\_rus, y\_train\_rus)

Fitting 5 folds for each of 6 candidates, totalling 30 fits

[Parallel(n\_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.

[Parallel(n\_jobs=1)]: Done 30 out of 30 | elapsed: 0.7s finished

GridSearchCV(cv=KFold(n\_splits=5, random\_state=4, shuffle=True),

error\_score=nan,

estimator=LogisticRegression(C=1.0, class\_weight=None, dual=False,

fit\_intercept=True,

intercept\_scaling=1, l1\_ratio=None,

max\_iter=100, multi\_class='auto',

n\_jobs=None, penalty='l2',

random\_state=None, solver='lbfgs',

tol=0.0001, verbose=0,

warm\_start=False),

iid='deprecated', n\_jobs=None,

param\_grid={'C': [0.01, 0.1, 1, 10, 100, 1000]},

pre\_dispatch='2\*n\_jobs', refit=True, return\_train\_score=True,

scoring='roc\_auc', verbose=1)

# results of grid search CV

cv\_results = pd.DataFrame(model\_cv.cv\_results\_)

cv\_results

# plot of C versus train and validation scores

plt.figure(figsize=(8, 6))

plt.plot(cv\_results['param\_C'], cv\_results['mean\_test\_score'])

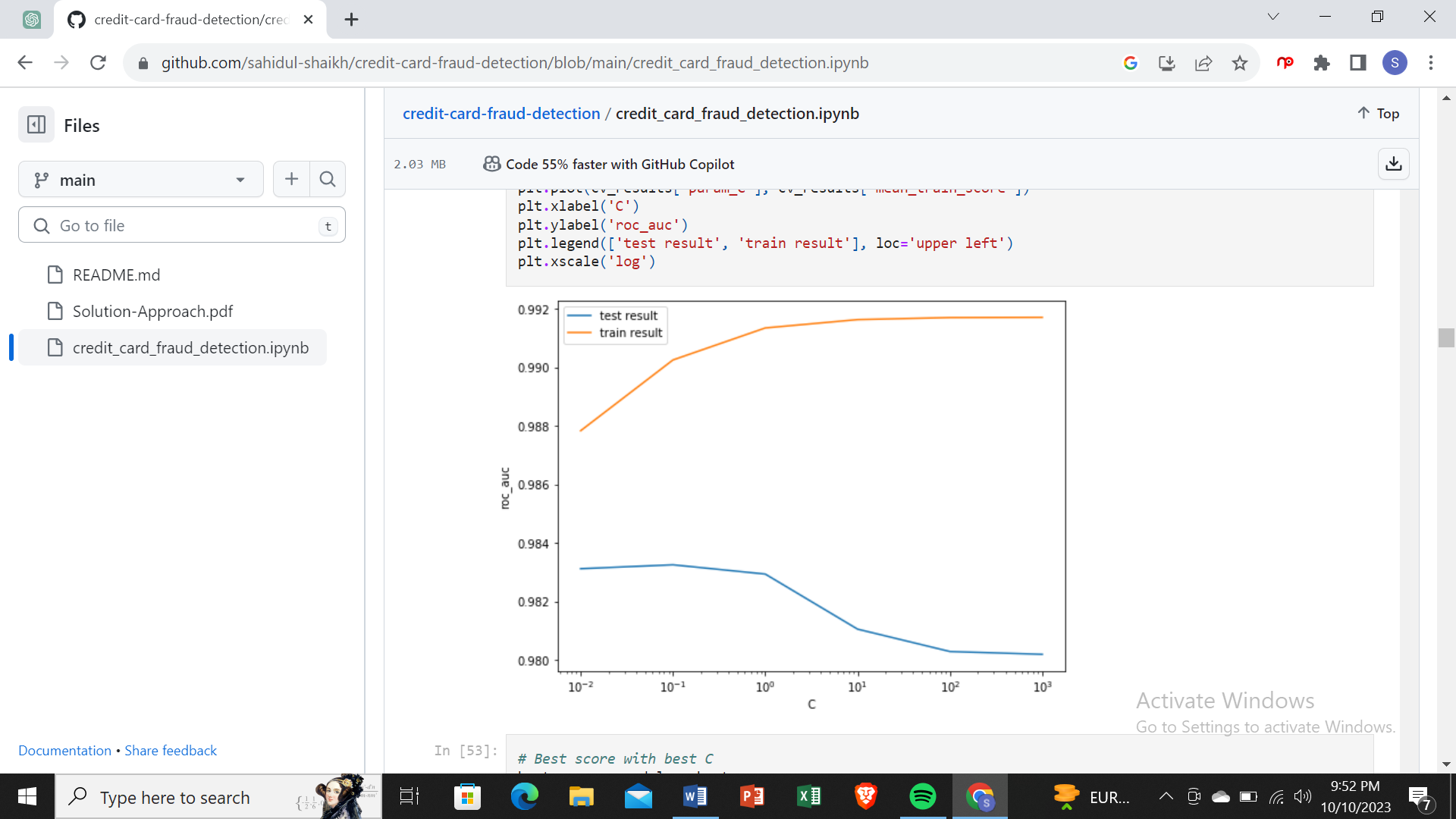
plt.plot(cv\_results['param\_C'], cv\_results['mean\_train\_score'])

plt.xlabel('C')

plt.ylabel('roc\_auc')

plt.legend(['test result', 'train result'], loc='upper left')

plt.xscale('log')



# Best score with best C

best\_score = model\_cv.best\_score\_

best\_C = model\_cv.best\_params\_['C']

print(" The highest test roc\_auc is {0} at C = {1}".format(best\_score, best\_C))

The highest test roc\_auc is 0.9832637280039689 at C = 0.1

Logistic regression with optimal C

# Instantiate the model with best C

logistic\_bal\_rus = LogisticRegression(C=0.1)

# Fit the model on the train set

logistic\_bal\_rus\_model = logistic\_bal\_rus.fit(X\_train\_rus, y\_train\_rus)

Prediction on the train set

# Predictions on the train set

y\_train\_pred = logistic\_bal\_rus\_model.predict(X\_train\_rus)

# Confusion matrix

confusion = metrics.confusion\_matrix(y\_train\_rus, y\_train\_pred)

print(confusion)

[[391 5]

[ 32 364]]

TP = confusion[1,1] # true positive

TN = confusion[0,0] # true negatives

FP = confusion[0,1] # false positives

FN = confusion[1,0] # false negatives

# Accuracy

print("Accuracy:-",metrics.accuracy\_score(y\_train\_rus, y\_train\_pred))

# Sensitivity

print("Sensitivity:-",TP / float(TP+FN))

# Specificity

print("Specificity:-", TN / float(TN+FP))

# F1 score

print("F1-Score:-", f1\_score(y\_train\_rus, y\_train\_pred))

Accuracy:- 0.9532828282828283

Sensitivity:- 0.9191919191919192

Specificity:- 0.9873737373737373

F1-Score:- 0.9516339869281046

# classification\_report

print(classification\_report(y\_train\_rus, y\_train\_pred))

precision recall f1-score support

0 0.92 0.99 0.95 396

1 0.99 0.92 0.95 396

accuracy 0.95 792

macro avg 0.96 0.95 0.95 792

weighted avg 0.96 0.95 0.95 792

# Predicted probability

y\_train\_pred\_proba = logistic\_bal\_rus\_model.predict\_proba(X\_train\_rus)[:,1]

# roc\_auc

auc = metrics.roc\_auc\_score(y\_train\_rus, y\_train\_pred\_proba)

auc

0.9892230384654627

# Plot the ROC curve

draw\_roc(y\_train\_rus, y\_train\_pred\_proba)

Prediction on the test set

# Prediction on the test set

y\_test\_pred = logistic\_bal\_rus\_model.predict(X\_test)

# Confusion matrix

confusion = metrics.confusion\_matrix(y\_test, y\_test\_pred)

print(confusion)

[[55658 1208]

[ 13 83]]

TP = confusion[1,1] # true positive

TN = confusion[0,0] # true negatives

FP = confusion[0,1] # false positives

FN = confusion[1,0] # false negatives

# Accuracy

print("Accuracy:-",metrics.accuracy\_score(y\_test, y\_test\_pred))

# Sensitivity

print("Sensitivity:-",TP / float(TP+FN))

# Specificity

print("Specificity:-", TN / float(TN+FP))

Accuracy:- 0.9785646571398476

Sensitivity:- 0.8645833333333334

Specificity:- 0.978757078043119

# classification\_report

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

precision recall f1-score support

0 1.00 0.98 0.99 56866

1 0.06 0.86 0.12 96

accuracy 0.98 56962

macro avg 0.53 0.92 0.55 56962

weighted avg 1.00 0.98 0.99 56962

Choosing best model on the balanced data

He we balanced the data with various approach such as Undersampling, Oversampling, SMOTE and Adasy. With every data balancing thechnique we built several models such as Logistic, XGBoost, Decision Tree, and Random Forest.

We can see that almost all the models performed more or less good. But we should be interested in the best model.

Though the Undersampling technique models performed well, we should keep mind that by doing the undersampling some imformation were lost. Hence, it is better not to consider the undersampling models.

Whereas the SMOTE and Adasyn models performed well. Among those models the simplist model Logistic regression has ROC score 0.99 in the train set and 0.97 on the test set. We can consider the Logistic model as the best model to choose because of the easy interpretation of the models and also the resourse requirements to build the mdoel is lesser than the other heavy models such as Random forest or XGBoost.

Hence, we can conclude that the Logistic regression model with SMOTE is the best model for its simlicity and less resource requirement.

Print the FPR,TPR & select the best threshold from the roc curve for the best model

print('Train auc =', metrics.roc\_auc\_score(y\_train\_smote, y\_train\_pred\_proba\_log\_bal\_smote))

fpr, tpr, thresholds = metrics.roc\_curve(y\_train\_smote, y\_train\_pred\_proba\_log\_bal\_smote)

threshold = thresholds[np.argmax(tpr-fpr)]

print("Threshold=",threshold)

Train auc = 0.9897539730968845

Threshold= 0.5311563613510013

We can see that the threshold is 0.53, for which the TPR is the highest and FPR is the lowest and we got the best ROC score.

Cost benefit analysis

We have tried several models till now with both balanced and imbalanced data. We have noticed most of the models have performed more or less well in terms of ROC score, Precision and Recall.

But while picking the best model we should consider few things such as whether we have required infrastructure, resources or computational power to run the model or not. For the models such as Random forest, SVM, XGBoost we require heavy computational resources and eventually to build that infrastructure the cost of deploying the model increases. On the other hand the simpler model such as Logistic regression requires less computational resources, so the cost of building the model is less.

We also have to consider that for little change of the ROC score how much monetary loss of gain the bank incur. If the amount if huge then we have to consider building the complex model even though the cost of building the model is high.