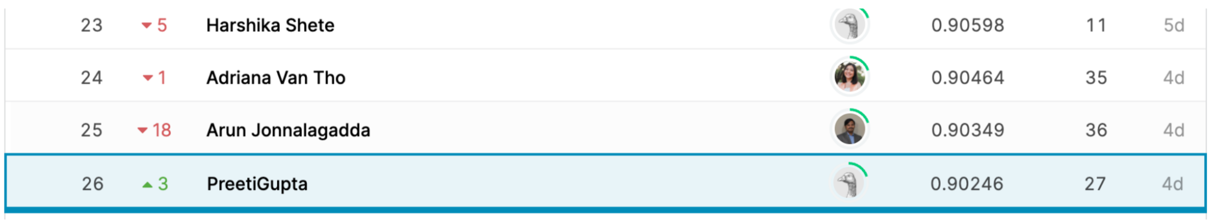
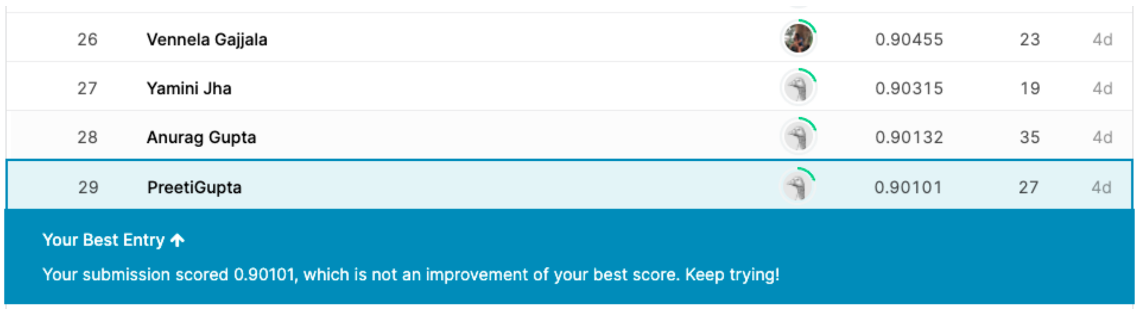
Preeti Gupta (pg25357)
Private leaderboard :



23	▼5	Harshika Shete	0.90598	11	5d
24	▼1	Adriana Van Tho	0.90464	35	4d
25	▼18	Arun Jonnalagadda	0.90349	36	4d
26	▲3	PreetiGupta	0.90246	27	4d

Public Leaderboard :



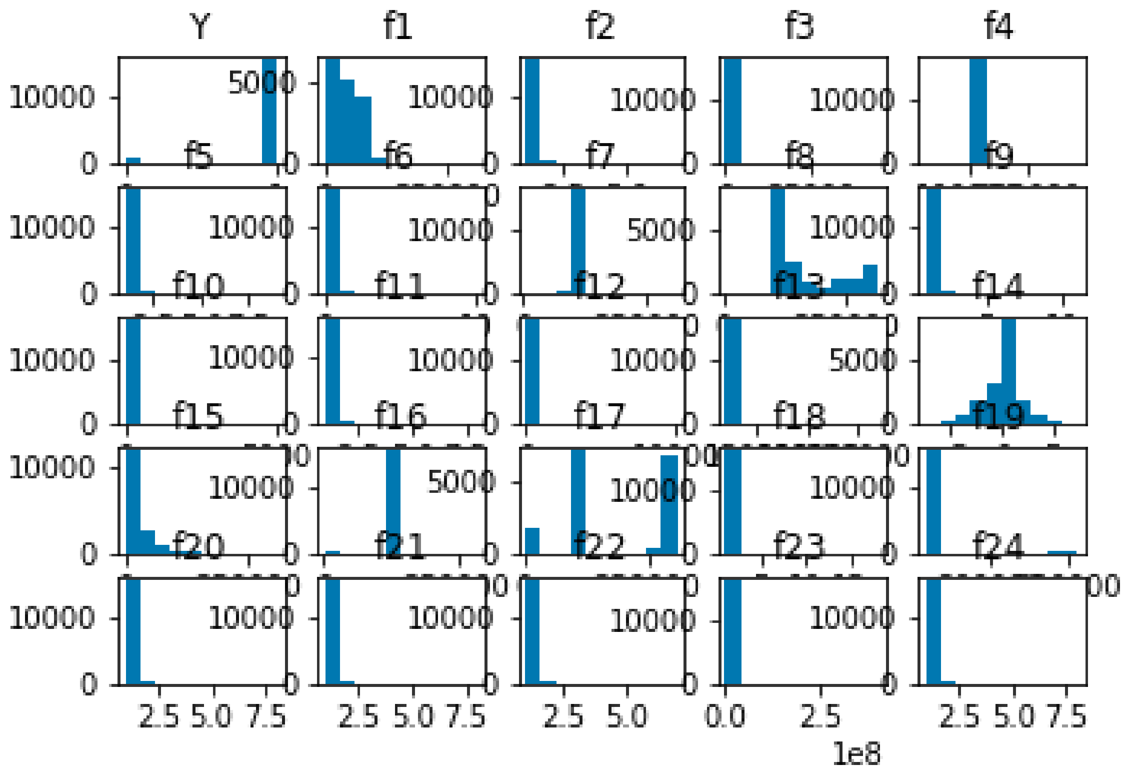
26	Vennela Gajjala	0.90455	23	4d
27	Yamini Jha	0.90315	19	4d
28	Anurag Gupta	0.90132	35	4d
29	PreetiGupta	0.90101	27	4d

Your Best Entry ↑
Your submission scored 0.90101, which is not an improvement of your best score. Keep trying!

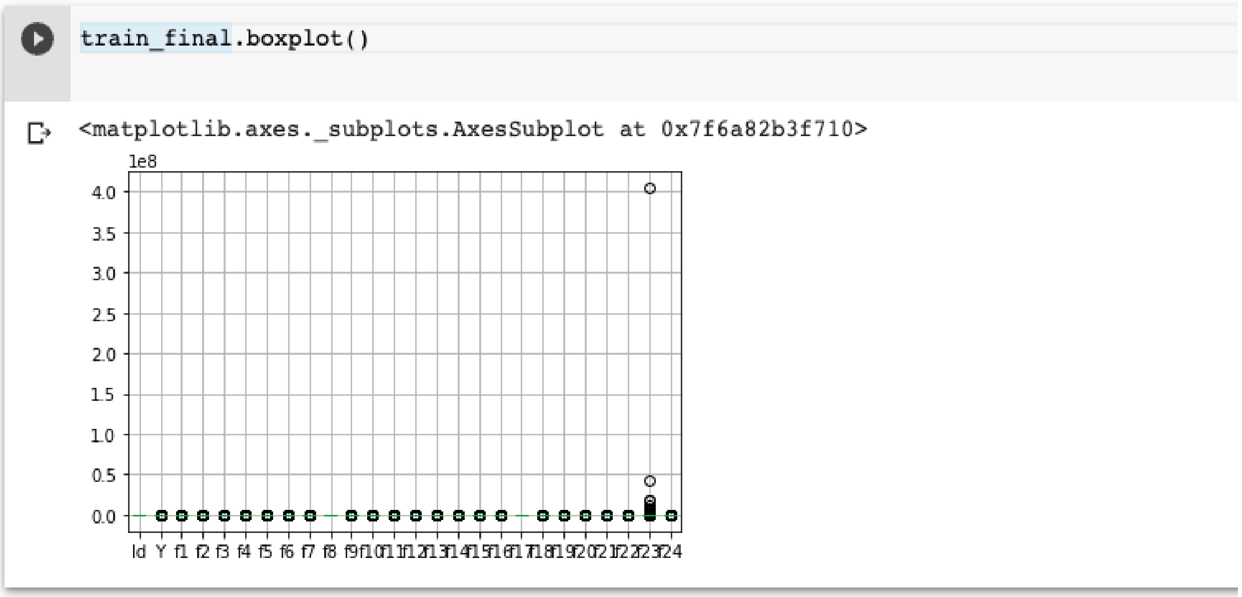
Private leaderboard score : 0.90246
Public leaderboard score : 0.90010

Some of the most innovative concepts applied and learned :
1. Generative adversarial networks for data generation,
2. Oversampling and undersampling of the data, for imbalance categorical data
3. Exploratory Data Analysis
4. XGBoost classifier,
5. Neural Networks
6. Hyperparameter tuning,
7. K fold Cross validation ,
8. stratified K fold cross validation.

Exploratory Data Analysis :
I plotted the histogram plot for different features.



Boxplot of the train data, clearly feature f23 has outliers that we need to remove or replace.



```
train_final.boxplot()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f6a82b3f710>
```

1e8
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Id Y f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 f11 f12 f13 f14 f15 f16 f17 f18 f19 f20 f21 f22 f23 f24

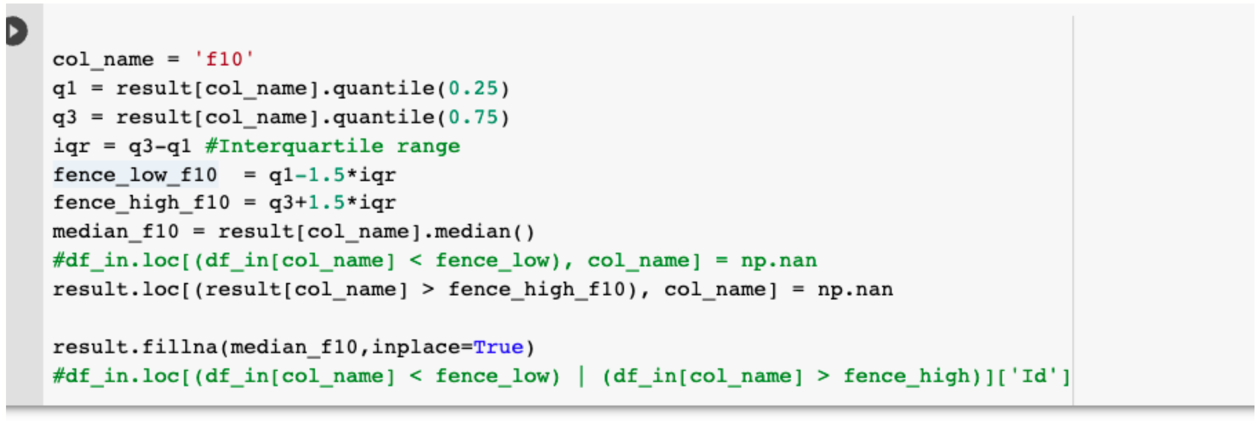
1. Here we can see that feature f17 can be categorical data since there are three widely separated bars that can be seen.
2. The feature f1 seems to be a good candidate for applying log as the data is skewed for f1
3. Similarly for feature f15, it seems that applying log transform can boost the prediction performance.
4. EDA : Feature f13 and f19 have a linear relationship, they have the same value for most of the points. This gave me intuition that probably only one of them is actually the correct feature and the other one is probably a polluted version of the correct feature.Feature f13 and f19 are highly correlated :

   By removing f13 and f19 individually the model is performing better when f13 is removed . Looks like the data has been tempered where some values in the actual feature f19 have been changed and named as f13.

   **This made me experiment with the model by training only with one of the two features at a time* *
5. The feature f3 is a categorical feature with fractional values. I multiplied it with 100 but this did not affect the overall score of the model.
6. 75% of the data points of the feature f23 belong to values [1, 2, 3, 4, 5, 6, 7, 8, 9] . Rest there were sparse values as large as 60926272. So it might be useful to convert all the values above 9 to a new category, say 10.
7. I also took the mean of the two different model predictions and used the mean of the two predictions for final predictions. I got slight boost in the performance with this approach

Removing Outlier: replacing with median:

For feature f10 and f12 I removed the outliers and made the prediction on the new dataset. This did enhance the performance. of the model.



```
col_name = 'f10'
q1 = result[col_name].quantile(0.25)
q3 = result[col_name].quantile(0.75)
iqr = q3-q1 #Interquartile range
fence_low_f10  = q1-1.5*iqr
fence_high_f10 = q3+1.5*iqr
median_f10 = result[col_name].median()
#df_in.loc[(df_in[col_name] < fence_low), col_name] = np.nan
result.loc[(result[col_name] > fence_high_f10), col_name] = np.nan

result.fillna(median_f10,inplace=True)
#df_in.loc[(df_in[col_name] < fence_low) | (df_in[col_name] > fence_high)]['Id']
```

# Applying different models:-

Models checked for:-

1. Ridge Regression

2. Linear models

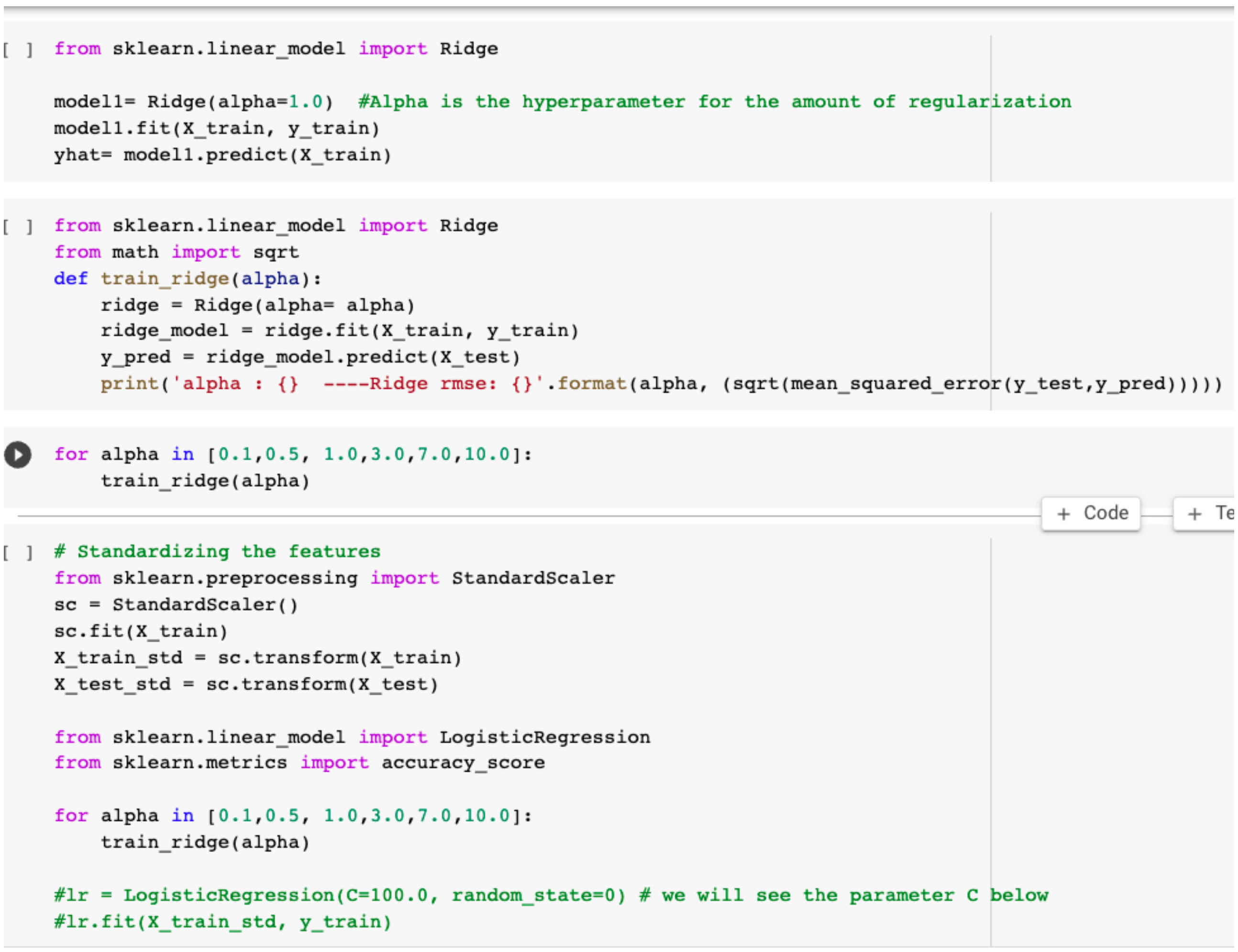
3. Lasso

4. Logistic Regression

5. Support Vector Mechanism

# Conclusion:- All the above based models gave me moderate results. Linear models gave score of 0.57 maximum hence i concluded that it's non-linear dataset and moved towards Non- Linear Models

# Logistic and SVM :- They gave better results in 0.70 and hence an indication Decision trees could be helpful.

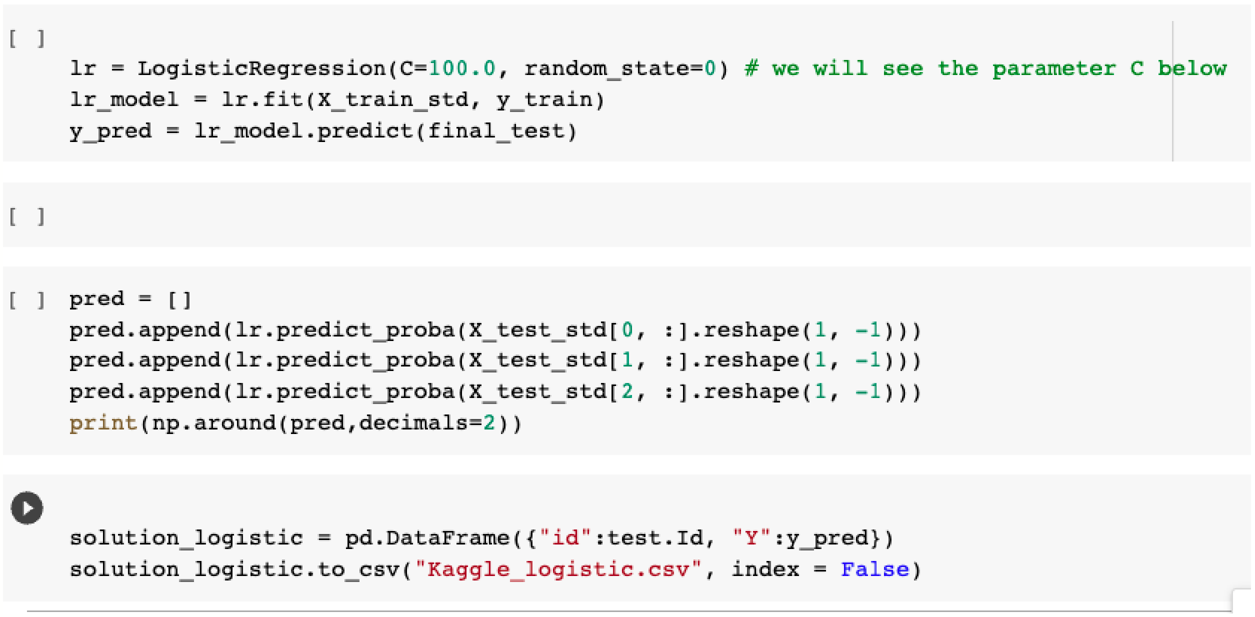


```
[ ] from sklearn.linear_model import Ridge

    model1= Ridge(alpha=1.0)  #Alpha is the hyperparameter for the amount of regularization
    model1.fit(X_train, y_train)
    yhat= model1.predict(X_train)
```

```
[ ] from sklearn.linear_model import Ridge
    from math import sqrt
    def train_ridge(alpha):
        ridge = Ridge(alpha= alpha)
        ridge_model = ridge.fit(X_train, y_train)
        y_pred = ridge_model.predict(X_test)
        print('alpha : {}  ----Ridge rmse: {}'.format(alpha, (sqrt(mean_squared_error(y_test,y_pred)))))
```

```
for alpha in [0.1,0.5, 1.0,3.0,7.0,10.0]:
    train_ridge(alpha)
```

```
[ ] # Standardizing the features
    from sklearn.preprocessing import StandardScaler
    sc = StandardScaler()
    sc.fit(X_train)
    X_train_std = sc.transform(X_train)
    X_test_std = sc.transform(X_test)

    from sklearn.linear_model import LogisticRegression
    from sklearn.metrics import accuracy_score

    for alpha in [0.1,0.5, 1.0,3.0,7.0,10.0]:
        train_ridge(alpha)

    #lr = LogisticRegression(C=100.0, random_state=0) # we will see the parameter C below
    #lr.fit(X_train_std, y_train)
```

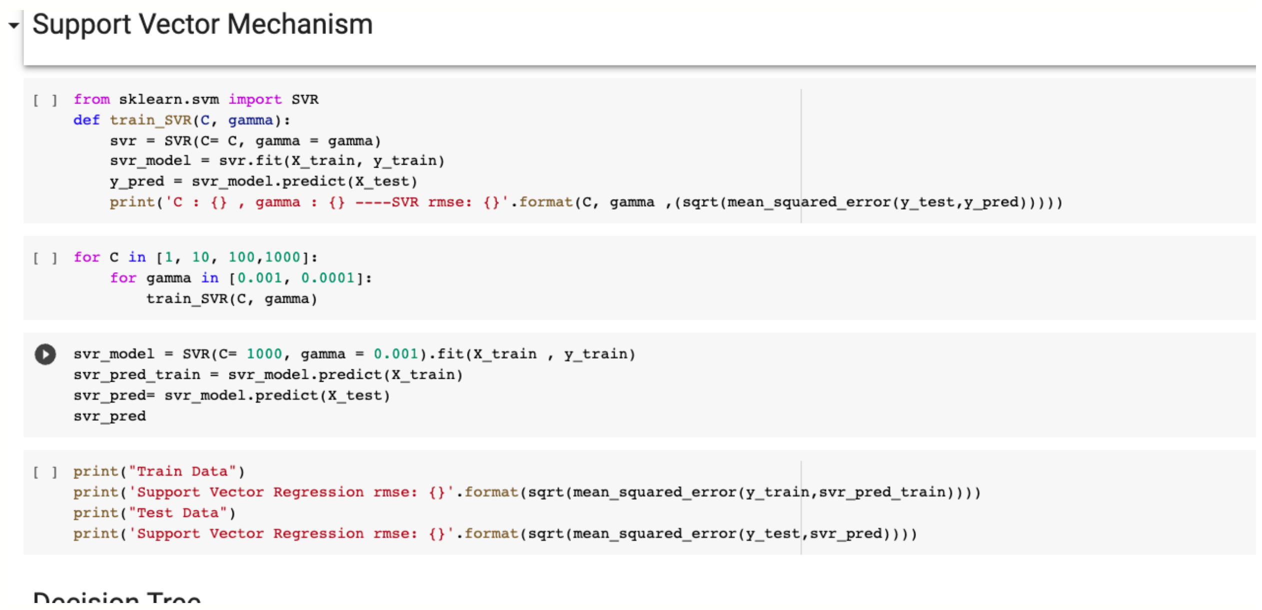


```
[ ]
    lr = LogisticRegression(C=100.0, random_state=0) # we will see the parameter C below
    lr_model = lr.fit(X_train_std, y_train)
    y_pred = lr_model.predict(final_test)
```

```
[ ]
```

```
[ ] pred = []
    pred.append(lr.predict_proba(X_test_std[0, :].reshape(1, -1)))
    pred.append(lr.predict_proba(X_test_std[1, :].reshape(1, -1)))
    pred.append(lr.predict_proba(X_test_std[2, :].reshape(1, -1)))
    print(np.around(pred,decimals=2))
```

```
solution_logistic = pd.DataFrame({"id":test.Id, "Y":y_pred})
solution_logistic.to_csv("Kaggle_logistic.csv", index = False)
```



Support Vector Mechanism

```
[ ] from sklearn.svm import SVR
    def train_SVR(C, gamma):
        svr = SVR(C= C, gamma = gamma)
        svr_model = svr.fit(X_train, y_train)
        y_pred = svr_model.predict(X_test)
        print('C : {} , gamma : {} ----SVR rmse: {}'.format(C, gamma ,(sqrt(mean_squared_error(y_test,y_pred)))))
```

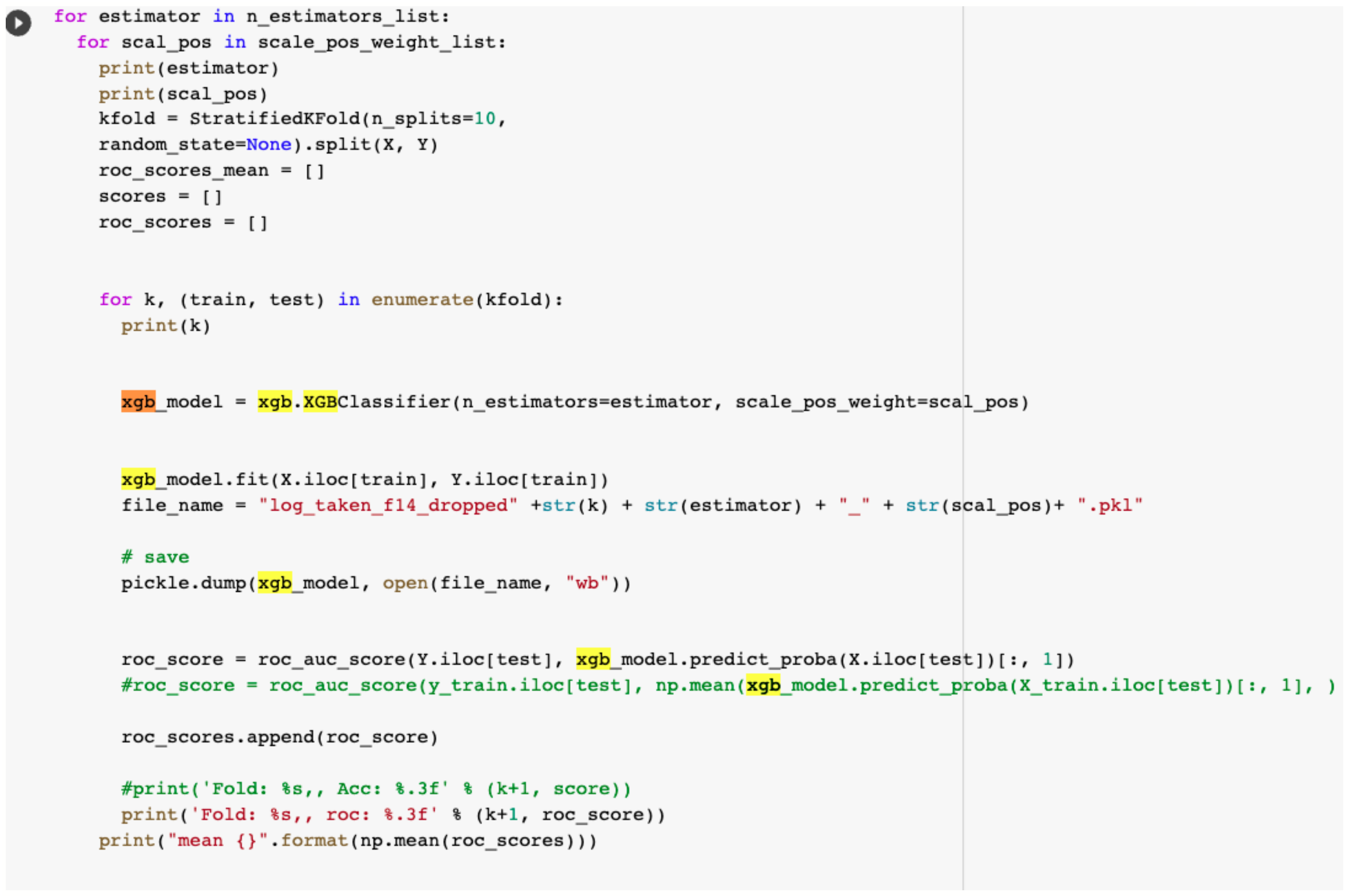
```
[ ] for C in [1, 10, 100,1000]:
        for gamma in [0.001, 0.0001]:
            train_SVR(C, gamma)
```

```
svr_model = SVR(C= 1000, gamma = 0.001).fit(X_train , y_train)
svr_pred_train = svr_model.predict(X_train)
svr_pred= svr_model.predict(X_test)
svr_pred
```

```
[ ] print("Train Data")
    print('Support Vector Regression rmse: {}'.format(sqrt(mean_squared_error(y_train,svr_pred_train))))
    print("Test Data")
    print('Support Vector Regression rmse: {}'.format(sqrt(mean_squared_error(y_test,svr_pred))))
```

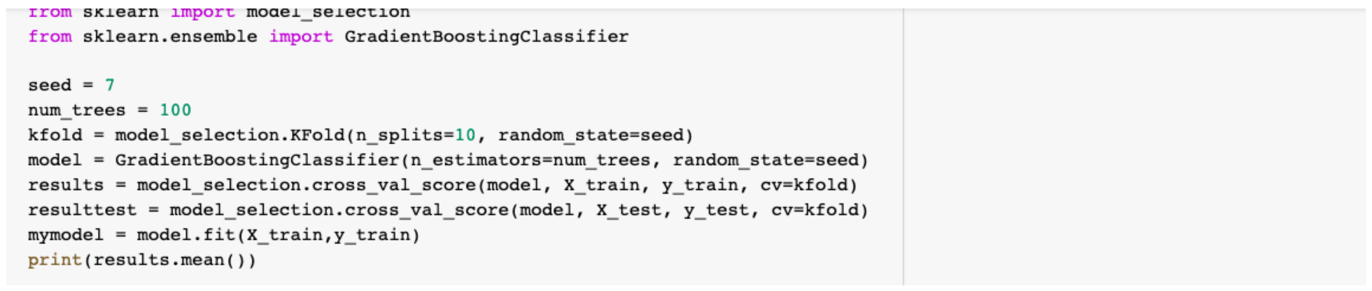
Decision Tree

XGboost classifier worked best for this dataset maximum score of 0.88 , further hyperparameter tuning and other techniques enhances the score to 0.90246

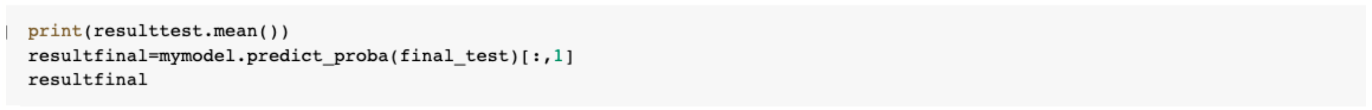


```
for estimator in n_estimators_list:
  for scal_pos in scale_pos_weight_list:
    print(estimator)
    print(scal_pos)
    kfold = StratifiedKFold(n_splits=10,
    random_state=None).split(X, Y)
    roc_scores_mean = []
    scores = []
    roc_scores = []


    for k, (train, test) in enumerate(kfold):
      print(k)


      xgb_model = xgb.XGBClassifier(n_estimators=estimator, scale_pos_weight=scal_pos)


      xgb_model.fit(X.iloc[train], Y.iloc[train])
      file_name = "log_taken_f14_dropped" +str(k) + str(estimator) + "_" + str(scal_pos)+ ".pkl"

      # save
      pickle.dump(xgb_model, open(file_name, "wb"))


      roc_score = roc_auc_score(Y.iloc[test], xgb_model.predict_proba(X.iloc[test])[:, 1])
      #roc_score = roc_auc_score(y_train.iloc[test], np.mean(xgb_model.predict_proba(X_train.iloc[test])[:, 1], )

      roc_scores.append(roc_score)

      #print('Fold: %s,, Acc: %.3f' % (k+1, score))
      print('Fold: %s,, roc: %.3f' % (k+1, roc_score))
    print("mean {}".format(np.mean(roc_scores)))
```

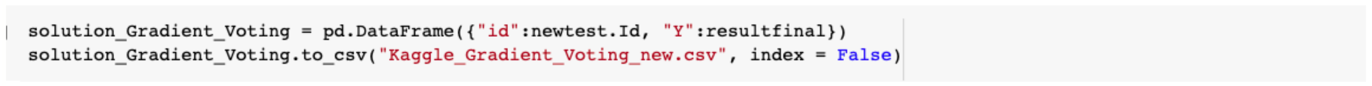
I also tried Stochastic Gradient Boosting after that and it also gave low score of 0.83 not bad though yet not good enough for me to persist and move forward as Gradient boosting with decision trees already gave me 0.88



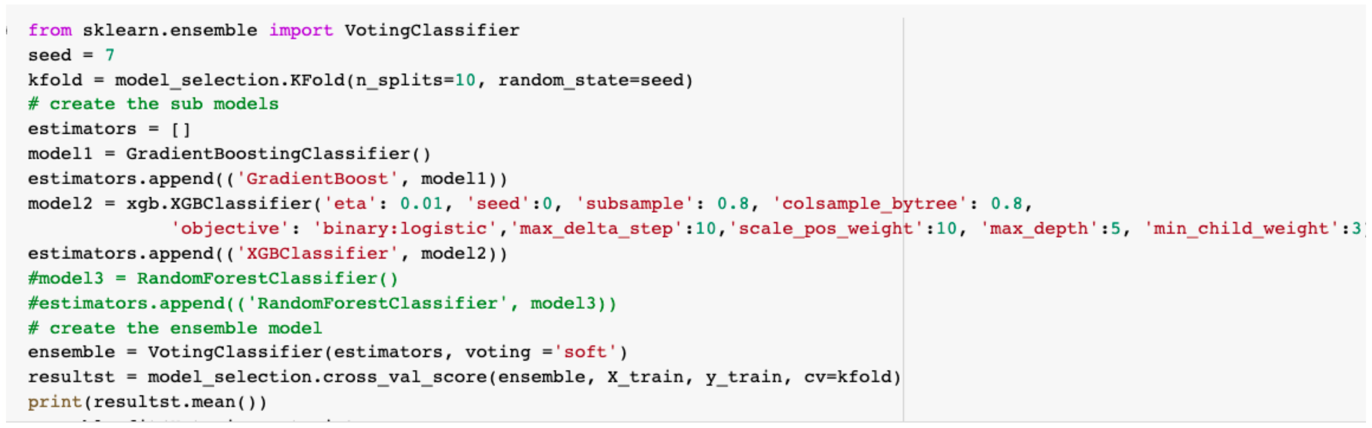
```
from sklearn import model_selection
from sklearn.ensemble import GradientBoostingClassifier

seed = 7
num_trees = 100
kfold = model_selection.KFold(n_splits=10, random_state=seed)
model = GradientBoostingClassifier(n_estimators=num_trees, random_state=seed)
results = model_selection.cross_val_score(model, X_train, y_train, cv=kfold)
resulttest = model_selection.cross_val_score(model, X_test, y_test, cv=kfold)
mymodel = model.fit(X_train,y_train)
print(results.mean())
```



```
print(resulttest.mean())
resultfinal=mymodel.predict_proba(final_test)[:,1]
resultfinal
```



```
solution_Gradient_Voting = pd.DataFrame({"id":newtest.Id, "Y":resultfinal})
solution_Gradient_Voting.to_csv("Kaggle_Gradient_Voting_new.csv", index = False)
```



```
from sklearn.ensemble import VotingClassifier
seed = 7
kfold = model_selection.KFold(n_splits=10, random_state=seed)
# create the sub models
estimators = []
model1 = GradientBoostingClassifier()
estimators.append(('GradientBoost', model1))
model2 = xgb.XGBClassifier('eta': 0.01, 'seed':0, 'subsample': 0.8, 'colsample_bytree': 0.8,
             'objective': 'binary:logistic','max_delta_step':10,'scale_pos_weight':10, 'max_depth':5, 'min_child_weight':3
estimators.append(('XGBClassifier', model2))
#model3 = RandomForestClassifier()
#estimators.append(('RandomForestClassifier', model3))
# create the ensemble model
ensemble = VotingClassifier(estimators, voting ='soft')
resultst = model_selection.cross_val_score(ensemble, X_train, y_train, cv=kfold)
print(resultst.mean())
```

I also experimented with neural Neural Network :

# Implementing Neural network:-

I chose 3 layers and used activation function as 'Relu' for first two and ' Sigmoid' for last layer

# Outputs:- 0.44 even though CV was accuracy 0.9643

# Learning:- Neural network is not that good for categorical datas and decision trees and XGBoost work best for categorical dataset

# Multiple tries:- I was able to tune my neural network to give me a result of 0.80 though after raising n_epoch = 1000. It took lot of time and seems increasing it helped train model better

steps_cal = (no of ex / batch_size) * no_of_epochs

1) Steps - number of times the training loop in your learning algorithm will run to update the parameters in the model. In each loop iteration, it will process a chunk of data, which is basically a batch. Usually, this loop is based on the Gradient Descent algorithm.

2) Batch size - the size of the chunk of data you feed in each loop of the learning algorithm. You can feed the whole data set, in which case the batch size is equal to the data set size.You can also feed one example at a time. Or you can feed some number N of examples.

3) Epoch - the number of times you run over the data set extracting batches to feed the learning algorithm.

Hyperparameter tuning with XGBoost classifier :

There were several hyperparameters that can be tuned in this model, I experimented with the following :

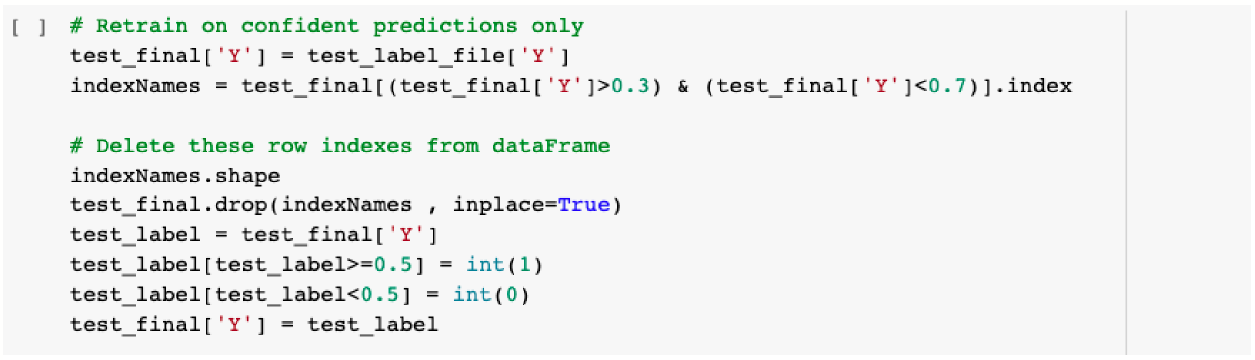
1. N_estimators, I tried different values between 10 to 5000. 2000 was giving the best result for this problem
2. learning_rate=0.1, I tried different values as 0.001 and 0.05 but the default gave the best results
3. reg_lambda=1, I tried to increase this value to increase regularization. But the default value gave the best result.
4. scale_pos_weight=1, I used values 0.04, 0.05, 0.062 and 16 (0.062 worked the best)

PSEUDO LABELING:

I used pseudo labeling to further enhance the score of the model. I tried two type of pseudo labeling

1. one where I used the entire test predictions by converting predictions above or equal to 0.5 as 1 and the ones below 0.5 as 0.
2. **Confident predictions only :** Second type of pseudo labeling took into account the confidence of the model predictions. I used only the predictions where the model was very confident . for this I used only the prediction rows where the model was
    a. Either 70% or more sure that the data point belongs to category 1
    b. Or 30% or lesser sure that the data point belongs to category 0 (this can be interpreted as 70% or more confident that the data point belongs to class 0)

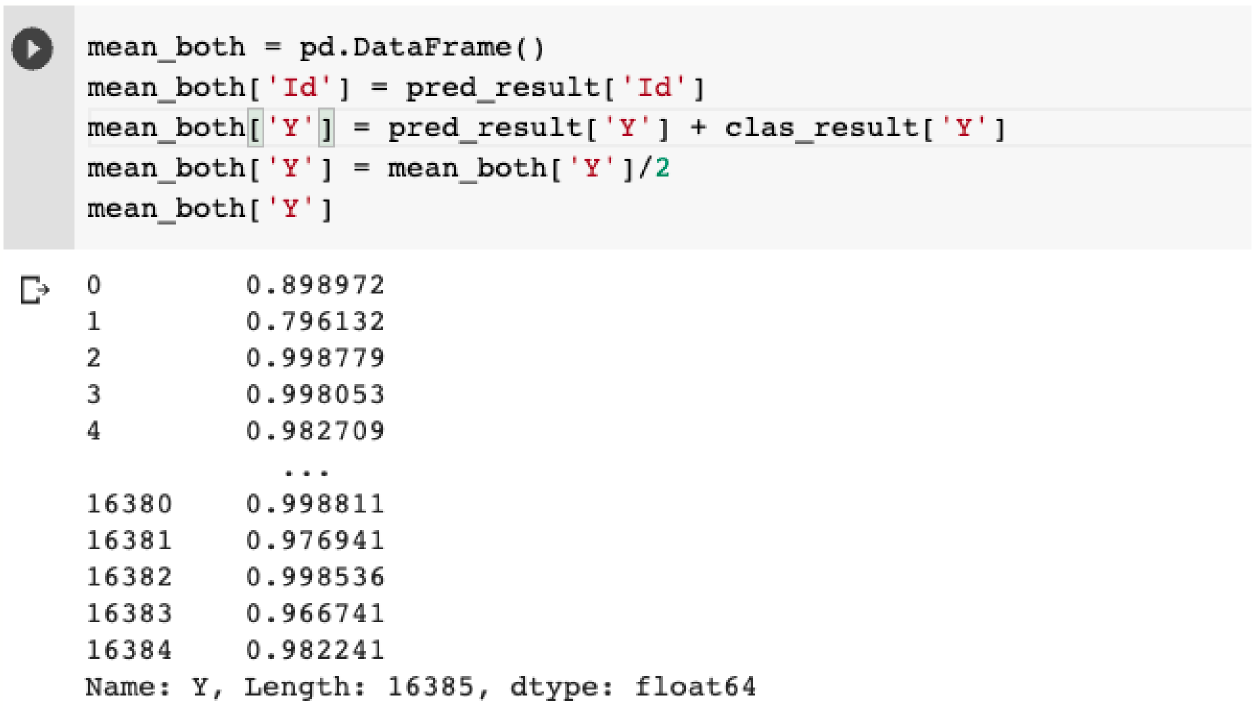
Keeping the confident predictions only further enhanced the AUC value.



```
# Retrain on confident predictions only
test_final['Y'] = test_label_file['Y']
indexNames = test_final[(test_final['Y']>0.3) & (test_final['Y']<0.7)].index

# Delete these row indexes from dataFrame
indexNames.shape
test_final.drop(indexNames , inplace=True)
test_label = test_final['Y']
test_label[test_label>=0.5] = int(1)
test_label[test_label<0.5] = int(0)
test_final['Y'] = test_label
```

Taking mean of the two best predictions :

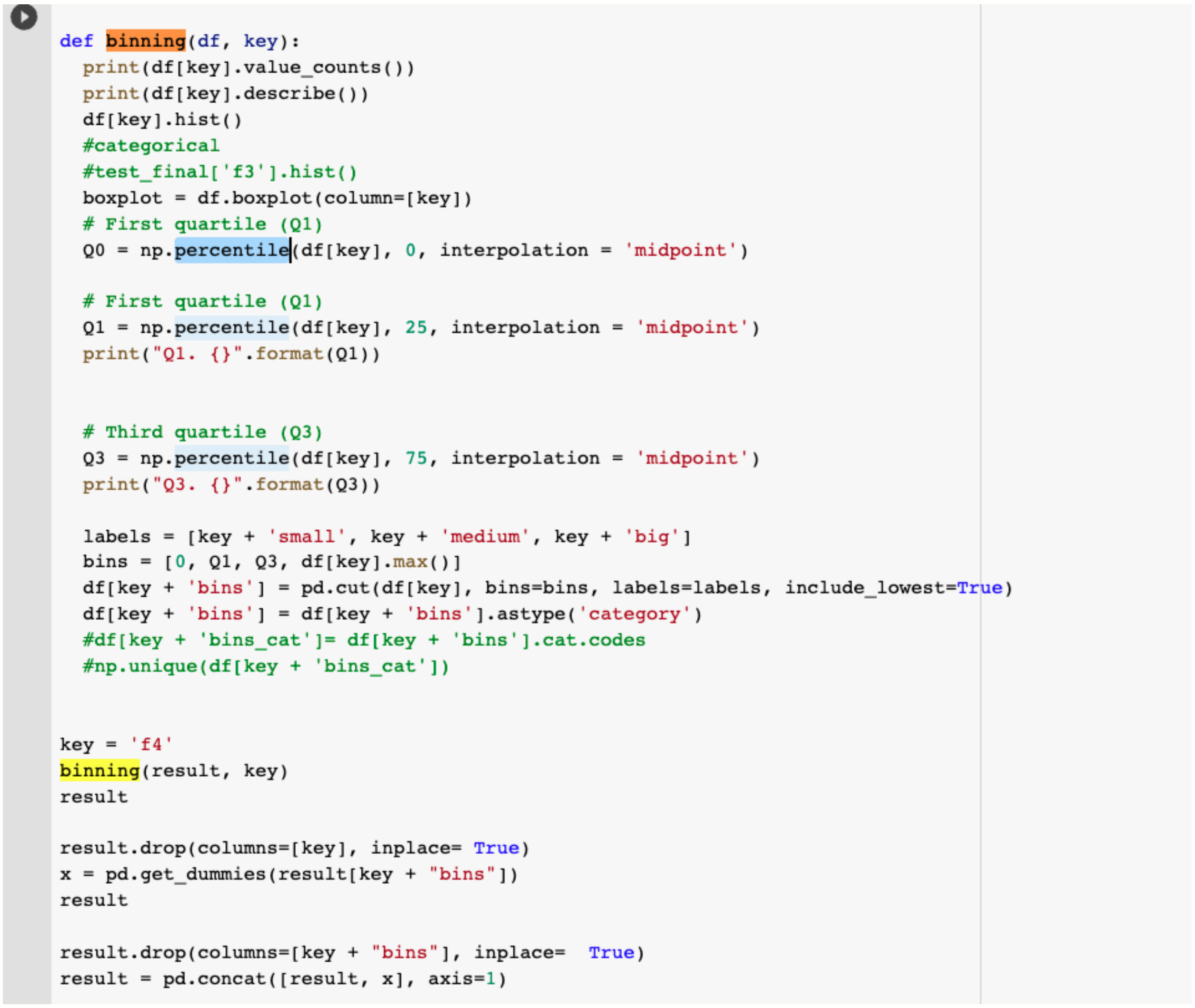
I took the mean of two best predictions and used that for predictions. This did give some boost to my predictions score : 0.89542



```
mean_both = pd.DataFrame()
mean_both['Id'] = pred_result['Id']
mean_both['Y'] = pred_result['Y'] + clas_result['Y']
mean_both['Y'] = mean_both['Y']/2
mean_both['Y']
```

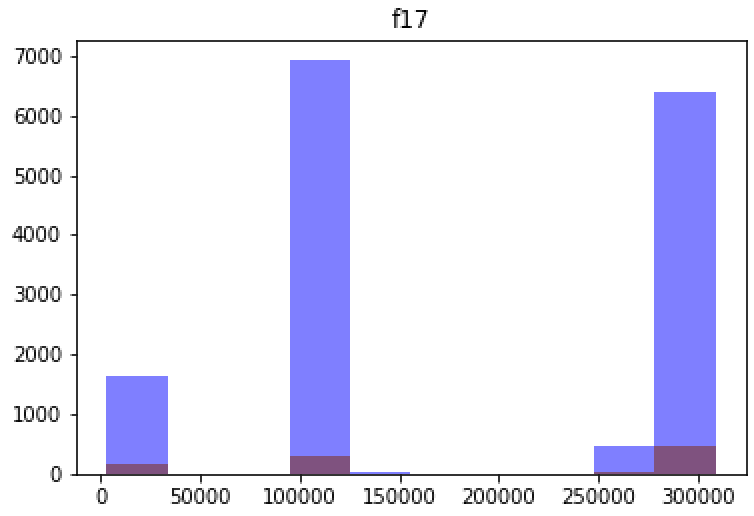
```
0        0.898972
1        0.796132
2        0.998779
3        0.998053
4        0.982709
           ...
16380    0.998811
16381    0.976941
16382    0.998536
16383    0.966741
16384    0.982241
Name: Y, Length: 16385, dtype: float64
```

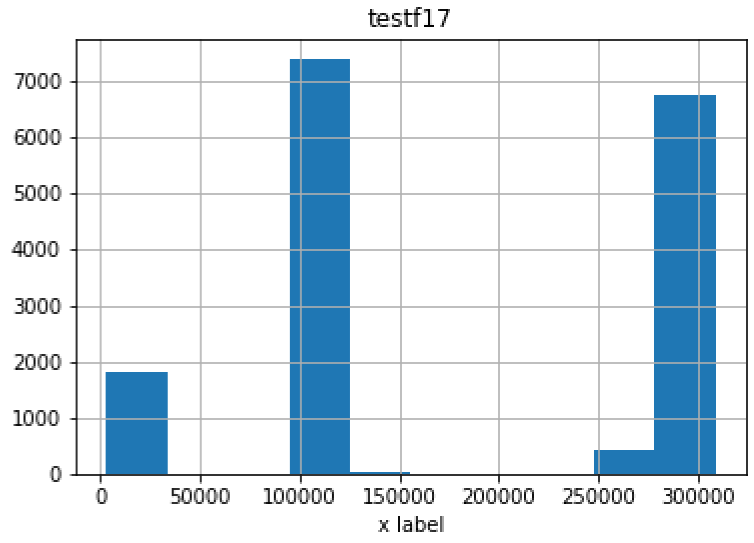
BINNING : I binned the features on the basis of minimum value, 25 percentile value, 75 percentile value and maximum value of the feature .



```
def binning(df, key):
  print(df[key].value_counts())
  print(df[key].describe())
  df[key].hist()
  #categorical
  #test_final['f3'].hist()
  boxplot = df.boxplot(column=[key])
  # First quartile (Q1)
  Q0 = np.percentile(df[key], 0, interpolation = 'midpoint')

  # First quartile (Q1)
  Q1 = np.percentile(df[key], 25, interpolation = 'midpoint')
  print("Q1. {}".format(Q1))


  # Third quartile (Q3)
  Q3 = np.percentile(df[key], 75, interpolation = 'midpoint')
  print("Q3. {}".format(Q3))

  labels = [key + 'small', key + 'medium', key + 'big']
  bins = [0, Q1, Q3, df[key].max()]
  df[key + 'bins'] = pd.cut(df[key], bins=bins, labels=labels, include_lowest=True)
  df[key + 'bins'] = df[key + 'bins'].astype('category')
  #df[key + 'bins_cat']= df[key + 'bins'].cat.codes
  #np.unique(df[key + 'bins_cat'])


key = 'f4'
binning(result, key)
result

result.drop(columns=[key], inplace= True)
x = pd.get_dummies(result[key + "bins"])
result

result.drop(columns=[key + "bins"], inplace=  True)
result = pd.concat([result, x], axis=1)
```

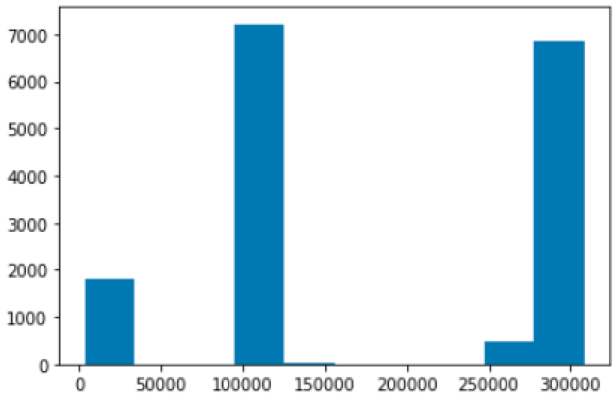
I also binned the data specially on the basis of the value of the data in the only zeros label dataframe and only ones label dataframe. Specially for feature f17 I used the above mentioned percentiles as well as two more cuts values 50000 and 200000. As can be seen from the plots these values act as major category generators. This did have an impact on the performance of the model


f17
7000
6000
5000
4000
3000
2000
1000
0
0
50000
100000
150000
200000
250000
300000

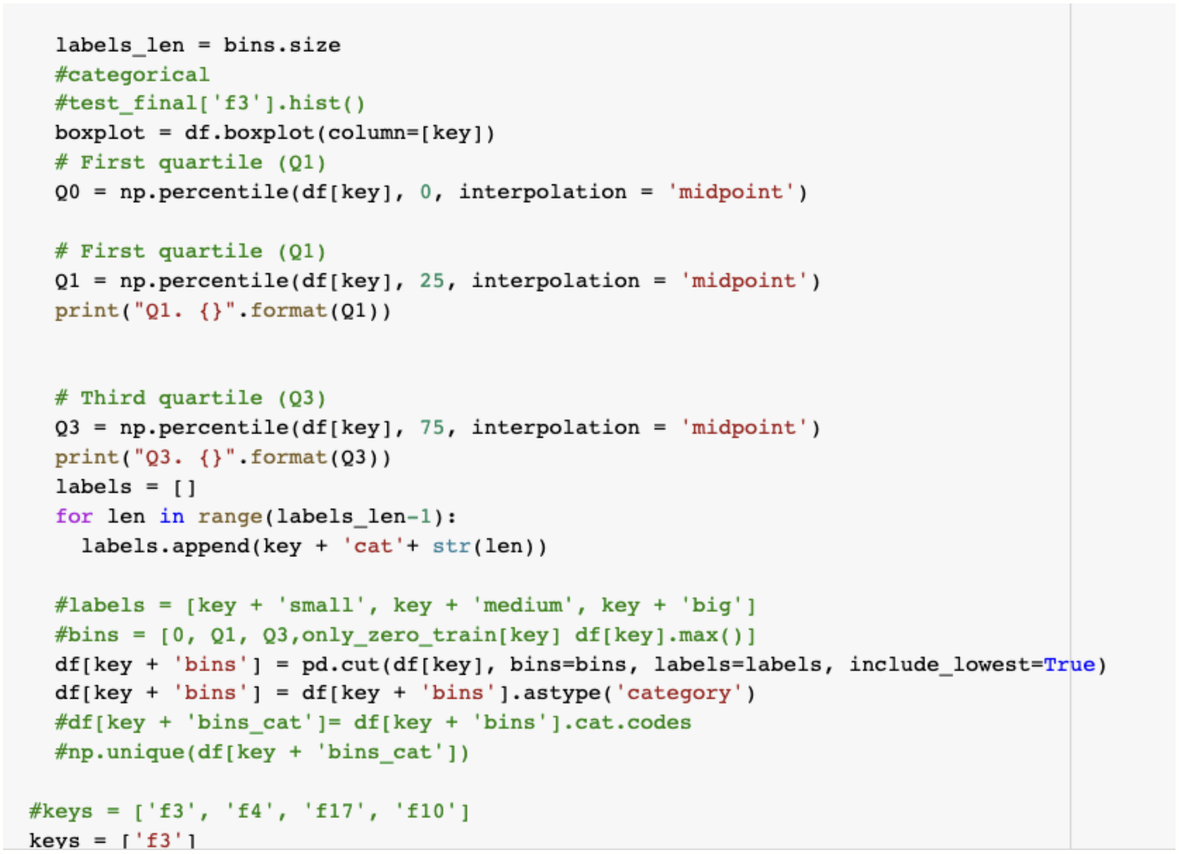

testf17
7000
6000
5000
4000
3000
2000
1000
0
0
50000
100000
150000
200000
250000
300000
x label

```
plt.hist(train_final['f17'])
```

```
(array([1.806e+03, 0.000e+00, 0.000e+00, 7.214e+03, 2.000e+01, 3.000e+00,
        0.000e+00, 0.000e+00, 4.930e+02, 6.847e+03]),
 array([  3130. ,  33674.4,  64218.8,  94763.2, 125307.6, 155852. ,
        186396.4, 216940.8, 247485.2, 278029.6, 308574. ]),
 <a list of 10 Patch objects>)
```



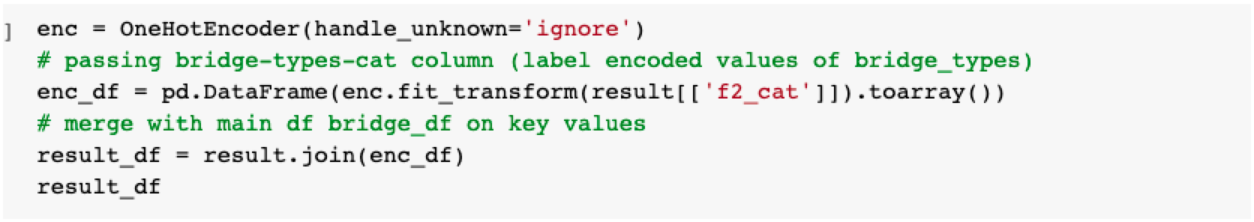
```
def binning_max_ones_zero_diff(df, key):
  print(df[key].value_counts())
  print(df[key].describe())
  df[key].hist()
  only_zero_train = train_final[train_final['Y']== 0]
  only_ones_train = train_final[train_final['Y']== 1]

  if key =='f17':
    bins = np.sort(np.unique(np.concatenate( ([0],[50000], [200000],np.array(only_ones_train[key].describe()[3:]),np.array(only_zero_train[key].de
  else:
    bins = np.sort(np.unique(np.concatenate( ([0],np.array(only_ones_train[key].describe()[3:]),np.array(only_zero_train[key].describe()[3:]) ))))
```



```
  labels_len = bins.size
  #categorical
  #test_final['f3'].hist()
  boxplot = df.boxplot(column=[key])
  # First quartile (Q1)
  Q0 = np.percentile(df[key], 0, interpolation = 'midpoint')

  # First quartile (Q1)
  Q1 = np.percentile(df[key], 25, interpolation = 'midpoint')
  print("Q1. {}".format(Q1))


  # Third quartile (Q3)
  Q3 = np.percentile(df[key], 75, interpolation = 'midpoint')
  print("Q3. {}".format(Q3))
  labels = []
  for len in range(labels_len-1):
    labels.append(key + 'cat'+ str(len))

  #labels = [key + 'small', key + 'medium', key + 'big']
  #bins = [0, Q1, Q3,only_zero_train[key] df[key].max()]
  df[key + 'bins'] = pd.cut(df[key], bins=bins, labels=labels, include_lowest=True)
  df[key + 'bins'] = df[key + 'bins'].astype('category')
  #df[key + 'bins_cat']= df[key + 'bins'].cat.codes
  #np.unique(df[key + 'bins_cat'])

#keys = ['f3', 'f4', 'f17', 'f10']
keys = ['f3']
```

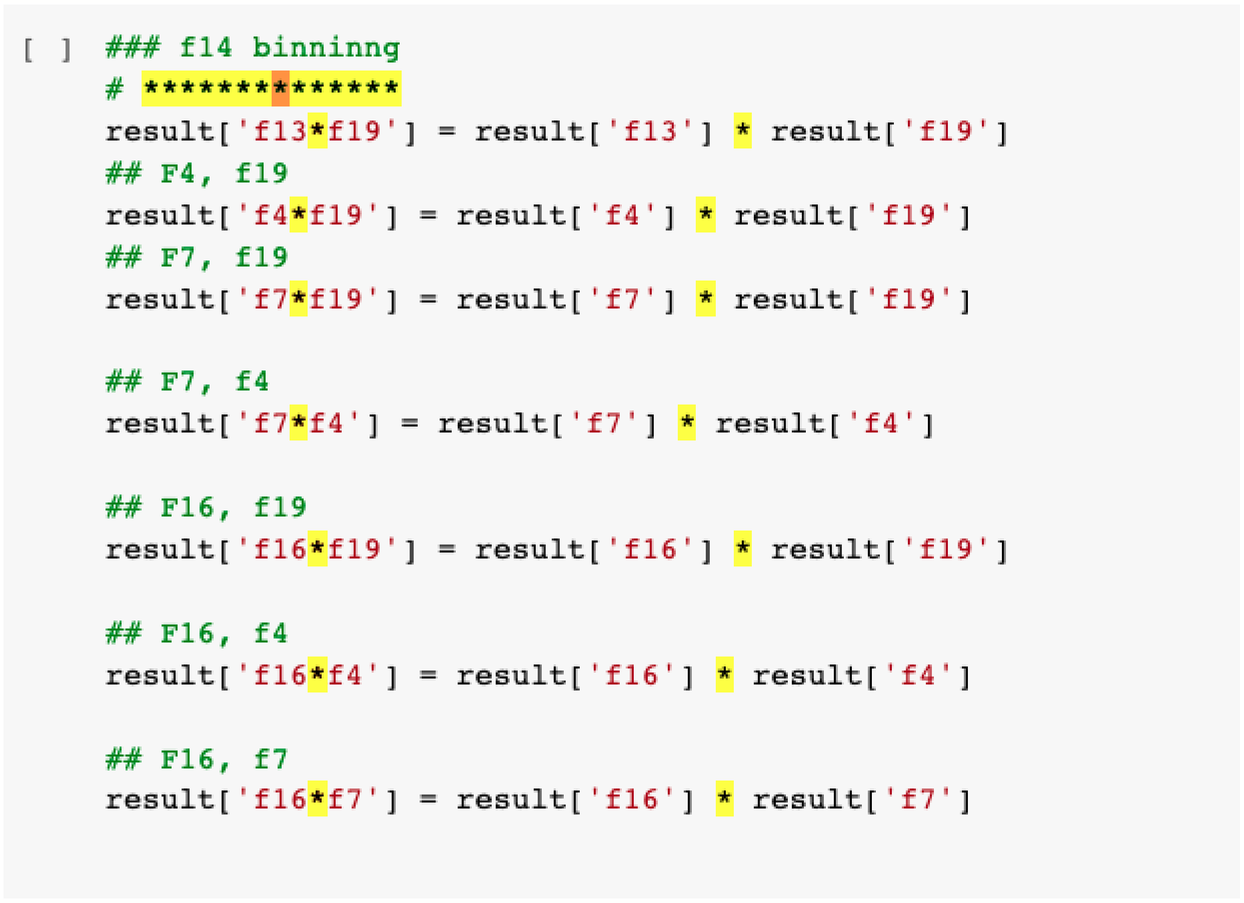
One HOT ENCODING:

I used one hot encoding for categorical data columns but it did not improve the model performance. The reason for this is that one hot encoding generates sparse columns.A tree algorithm will downplay and even ignore the one-hot encoded features especially if there are a lot of them. This is because Trees like variables that have high recall over those that have high precision but low recall.



```
] enc = OneHotEncoder(handle_unknown='ignore')
  # passing bridge-types-cat column (label encoded values of bridge_types)
  enc_df = pd.DataFrame(enc.fit_transform(result[['f2_cat']]).toarray())
  # merge with main df bridge_df on key values
  result_df = result.join(enc_df)
  result_df
```

Feature generation using pairwise feature multiplication:



```
[ ]  ### f14 binninng
     # **************
     result['f13*f19'] = result['f13'] * result['f19']
     ## F4, f19
     result['f4*f19'] = result['f4'] * result['f19']
     ## F7, f19
     result['f7*f19'] = result['f7'] * result['f19']

     ## F7, f4
     result['f7*f4'] = result['f7'] * result['f4']

     ## F16, f19
     result['f16*f19'] = result['f16'] * result['f19']

     ## F16, f4
     result['f16*f4'] = result['f16'] * result['f4']

     ## F16, f7
     result['f16*f7'] = result['f16'] * result['f7']
```

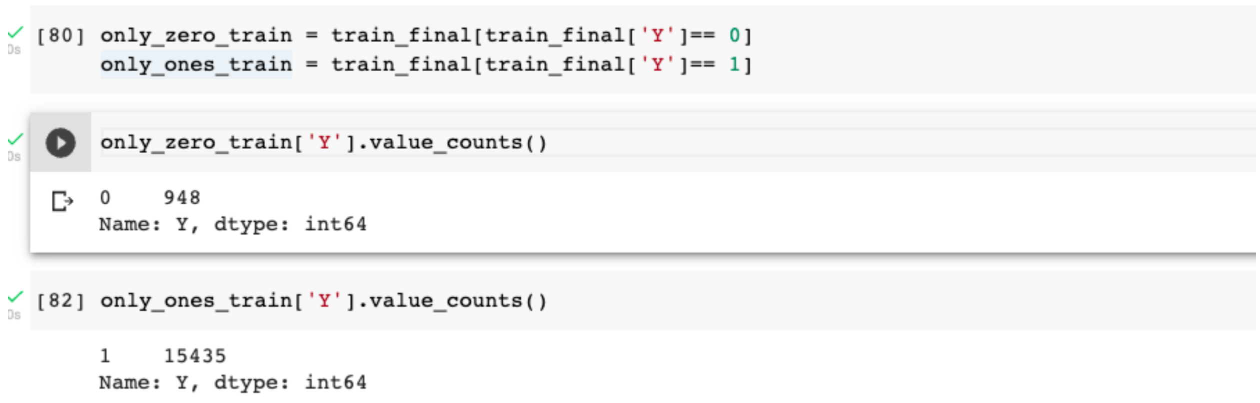
I tried generating more and more data using multiplying already present columns, although this seemed like a good approach it did not improve the final score.

I also generated pairwise features for all the features present in the train file, it resulted in 276 features (24 C 2). This did not help in the performance, too many features resulted in a complex model which was not performing well on the test data.

K-Fold Cross validation : I used K-fold (10 folds) cross validation to train the model and saved those models. Then for prediction I used those models and used the mean of these models to make the final prediction. This provided better performance on the test data since the predictions were more stable and a result of the average predictions of different models trained on different chunks of the data.

Feature removal I tried removing different features but it did not improve the model performance.

Imbalanced class data: The data was highly imbalanced: zeros to ones ratio being : 948/15435 approximately : 0.0614



```
[80] only_zero_train = train_final[train_final['Y']== 0]
     only_ones_train = train_final[train_final['Y']== 1]
```

```
only_zero_train['Y'].value_counts()
```

```
0    948
Name: Y, dtype: int64
```

```
[82] only_ones_train['Y'].value_counts()
```

```
1    15435
Name: Y, dtype: int64
```

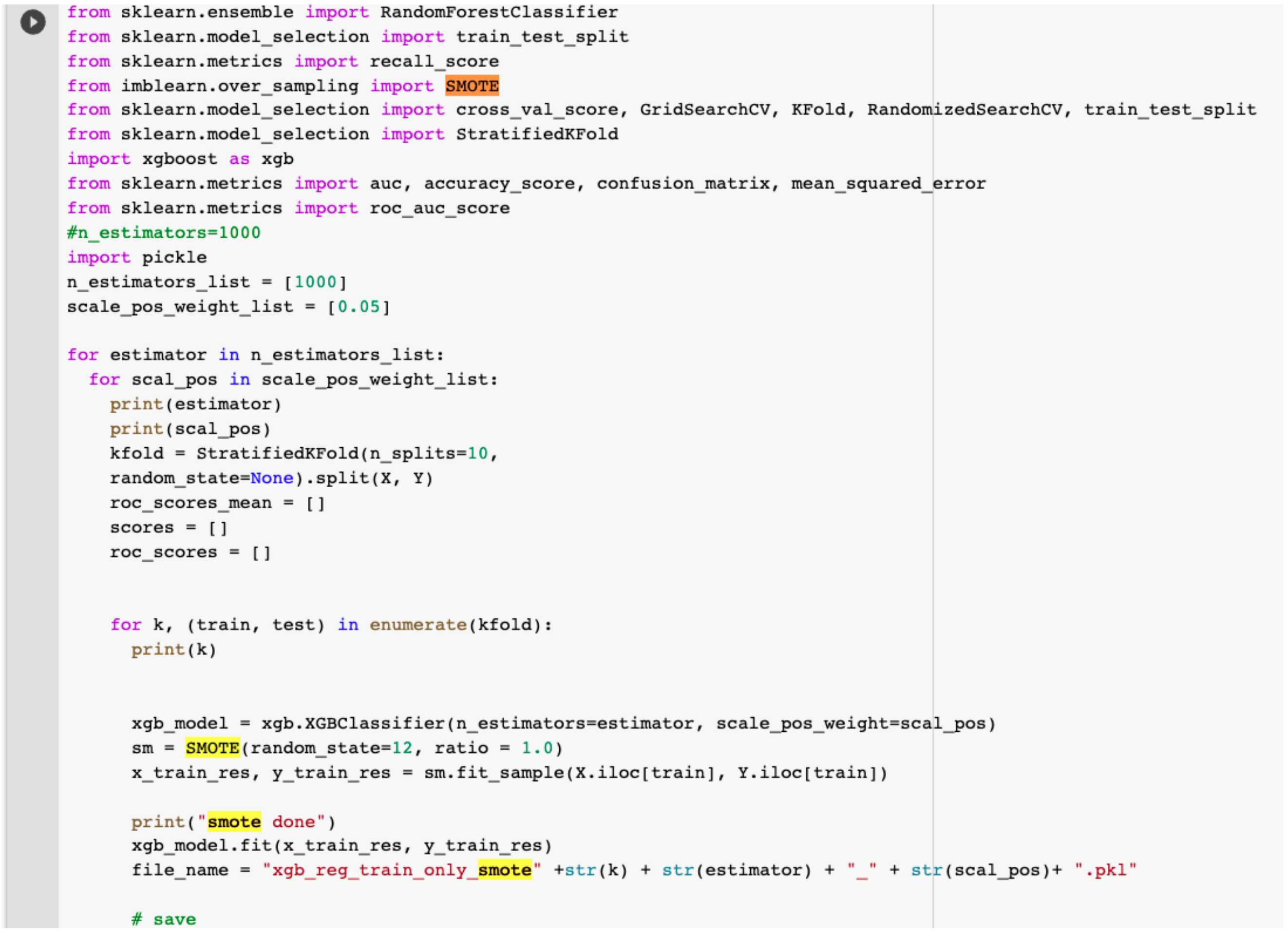
Solution :

1. **Data Generation:**

One approach to addressing the problem of class imbalance is to randomly resample the training dataset. The two main approaches to randomly resampling an imbalanced dataset are to delete examples from the majority class, called undersampling, and to duplicate examples from the minority class, called oversampling.

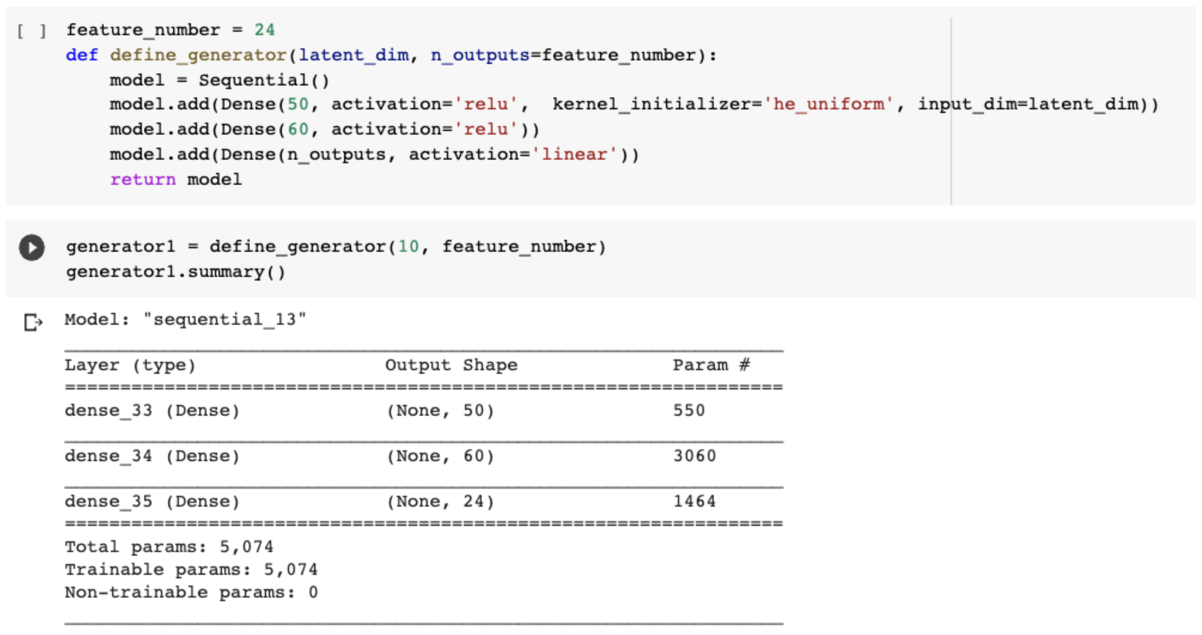
I used oversampling and undersampling to generate more data. I made use of the **SMOTE** library for oversampling of the data and I used **RandomUnderSampler** for the undersampling of the data.Here is the blog that I followed for this experiment : https://machinelearningmastery.com/smote-oversampling-for-imbalanced-classification/

Although this approach seemed promising, it could not improve the performance score. The major challenge was to generate the data with the same distribution as the original data.



```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import recall_score
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import cross_val_score, GridSearchCV, KFold, RandomizedSearchCV, train_test_split
from sklearn.model_selection import StratifiedKFold
import xgboost as xgb
from sklearn.metrics import auc, accuracy_score, confusion_matrix, mean_squared_error
from sklearn.metrics import roc_auc_score
#n_estimators=1000
import pickle
n_estimators_list = [1000]
scale_pos_weight_list = [0.05]

for estimator in n_estimators_list:
  for scal_pos in scale_pos_weight_list:
    print(estimator)
    print(scal_pos)
    kfold = StratifiedKFold(n_splits=10,
    random_state=None).split(X, Y)
    roc_scores_mean = []
    scores = []
    roc_scores = []


    for k, (train, test) in enumerate(kfold):
      print(k)


      xgb_model = xgb.XGBClassifier(n_estimators=estimator, scale_pos_weight=scal_pos)
      sm = SMOTE(random_state=12, ratio = 1.0)
      x_train_res, y_train_res = sm.fit_sample(X.iloc[train], Y.iloc[train])

      print("smote done")
      xgb_model.fit(x_train_res, y_train_res)
      file_name = "xgb_reg_train_only_smote" +str(k) + str(estimator) + "_" + str(scal_pos)+ ".pkl"

      # save
```

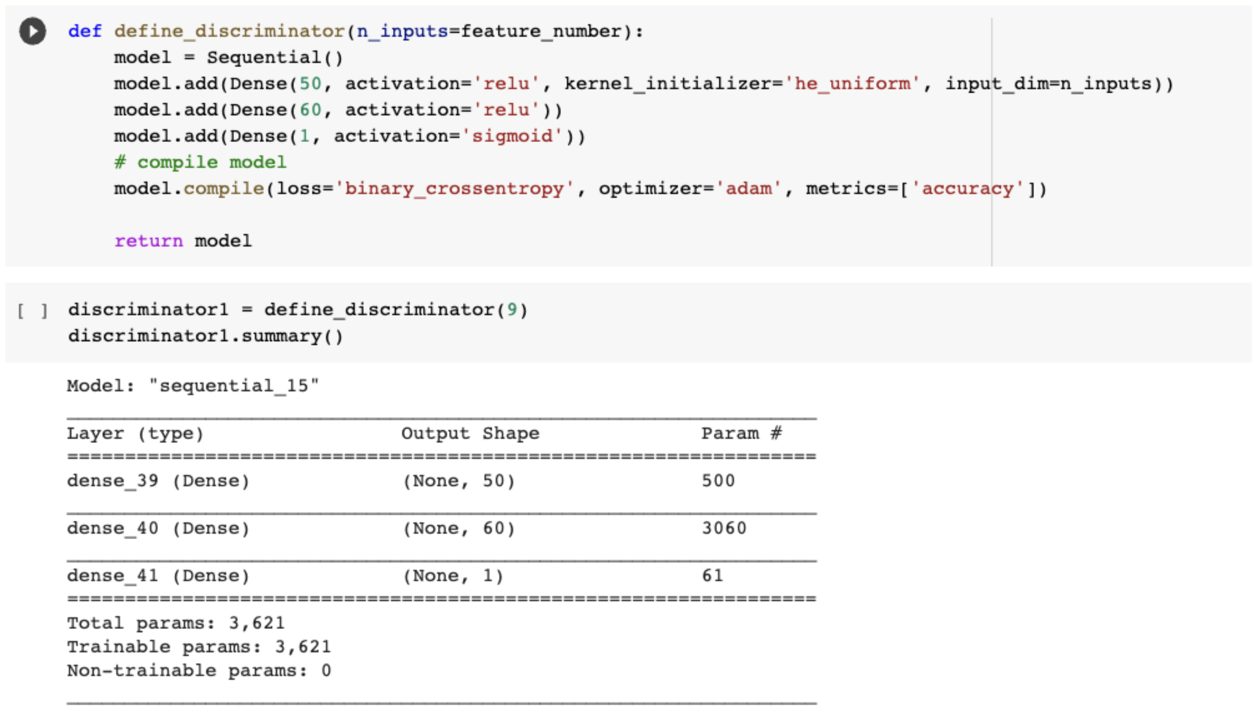
2. **GAN for generating Categorical data**: I also thought of generating more data using the current data with the help of generative adversarial networks (GAN). I also experimented with Generative adversarial networks to generate more data in order to enhance the model predictions. Although this could not work because of the GAN model not training properly, it was a great learning opportunity to study about Generative adversarial networks and their application in this problem statement.



```
[ ] feature_number = 24
    def define_generator(latent_dim, n_outputs=feature_number):
        model = Sequential()
        model.add(Dense(50, activation='relu',  kernel_initializer='he_uniform', input_dim=latent_dim))
        model.add(Dense(60, activation='relu'))
        model.add(Dense(n_outputs, activation='linear'))
        return model
```

```
generator1 = define_generator(10, feature_number)
generator1.summary()
```

```
Model: "sequential_13"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_33 (Dense)             (None, 50)                550
_________________________________________________________________
dense_34 (Dense)             (None, 60)                3060
_________________________________________________________________
dense_35 (Dense)             (None, 24)                1464
=================================================================
Total params: 5,074
Trainable params: 5,074
Non-trainable params: 0
_________________________________________________________________
```



```
def define_discriminator(n_inputs=feature_number):
    model = Sequential()
    model.add(Dense(50, activation='relu', kernel_initializer='he_uniform', input_dim=n_inputs))
    model.add(Dense(60, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    # compile model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

    return model
```

```
[ ] discriminator1 = define_discriminator(9)
    discriminator1.summary()
```

```
Model: "sequential_15"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_39 (Dense)             (None, 50)                500
_________________________________________________________________
dense_40 (Dense)             (None, 60)                3060
_________________________________________________________________
dense_41 (Dense)             (None, 1)                 61
=================================================================
Total params: 3,621
Trainable params: 3,621
Non-trainable params: 0
_________________________________________________________________
```

I am attaching the google colab notebook that I used for Generative adversarial networks.

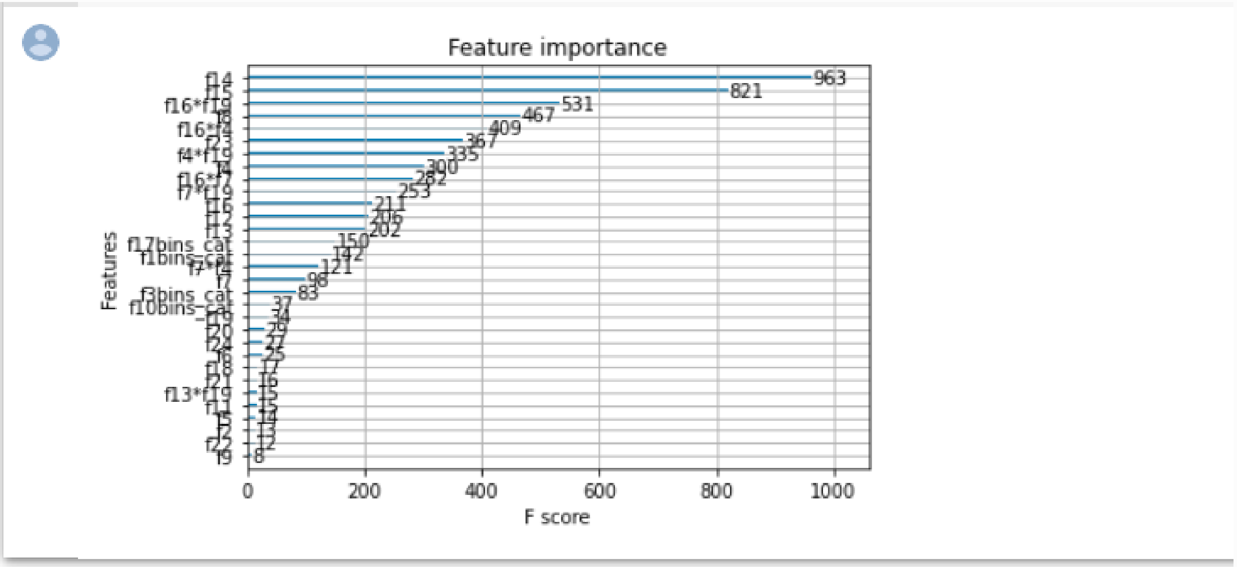
3. **Hyperparameter Tuning : I**n order to tackle this I used the scale_pos_weight parameter in the XGBoost to be close to 948/15435 ~0.062 and also

experimented with values like . The scale_pos_weight : controls the balance of positive and negative weights, useful for unbalanced classes. A typical value to consider: sum(negative instances) / sum(positive instances).

4. Stratified K-fold cross validation : This cross-validation object is a variation of KFold that returns stratified folds. The folds are made by preserving the percentage of samples for each class. Since the data was imbalanced it made sense to use stratified k-fold cross validation so that both the train data and validation data have similar proportions of both classes.

Plotting the feature importance of the xgb classifier model :

```
# plot feature importance

from xgboost import plot_importance

import matplotlib.pyplot as plt

plot_importance(xgb_model)

plt.show()
```



A brief discussion of post-closing of Kaggle competition reflections. If you selected the right submissions for the Private LB scoring, what you learned:

1. The kaggle competition was a good experience to have but it would have been better if the leaderboard was not continuously visible. It put me under a lot of pressure and I could not do as good as I could have.
2. I could have used a different file which could have given me a score of 0.90469. This was the file which was the average of the two best models. I was under confidence with this file since I thought it might be overfitting on the public leaderboard and might give worse results on the private leaderboard.
3. I think I should have used just one google colab for development purposes. Having multiple files resulted in confusion.