Compare Different Methods of Credit Card Classification

Name: Li Ming Kei
SID: 20698877
HKUST Email: mkliac@connect.ust.hk

Description

In this project, I perform a binary classification task to classify whether an applicant with particular features X is a risky client from the perspective of the financial industry. To achieve this, I have done the following tasks:

(i) Explore the correlation between each specific feature and risk using a linear classifier.
(ii) Explore the 2D and 3D representation of the dataset using PCA.
(iii) Compare different sampling methods with their results in dealing with imbalanced data.
(iv) Use different machine learning models to tackle the classification task and find the best one for it.

Besides the above, I'm also interested in the relationship between age and the total income of a client. Therefore, I have another extra task:

(v) Build clustering on the feature's age and the total income.

This project is conducted in Colab.

Description of the Dataset

I choose the "credit card classification" dataset from the project description as my base dataset. It is a clean version dataset that contains 20 columns with 9709 rows where the raw dataset is from https://www.kaggle.com/datasets/rikdifos/credit-card-approval-prediction. The columns are below:

```
['ID', 'Gender', 'Own_car', 'Own_property', 'Work_phone', 'Phone',
'Email', 'Unemployed', 'Num_children', 'Num_family', 'Account_length',
'Total_income', 'Age', 'Years_employed', 'Income_type',
'Education_type', 'Family_status', 'Housing_type', 'Occupation_type',
'Target']
```

The first 19 columns are the feature X of the applicant and within it, columns {Income_type, Education_type, Family_status, Housing_type, Occupation_type} are all categorical data while the others are numeric data. The last column "Target" is the risk of the applicant where 0 stand for low risk and 1 stand for high risk. It is treated as Y for my classification task.

Modification of the Dataset

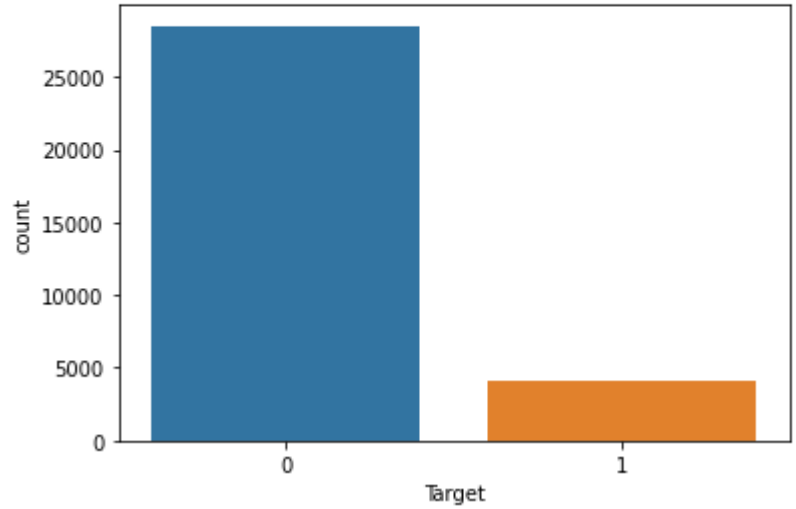
Before preprocessing, I modified the way the author clean the raw dataset as I spot a mistake the author made in cleaning the dataset. In the raw dataset, there are two datasets namely "application_record" and "credit_record". Both of the datasets contain rows that each of them is the features on a corresponding ID. It is expected that after merging the two datasets by ID we can get the complete dataset. However, the author cleans the dataset in this way:

(1) drop all the duplicates in "application_record"
(2) preprocess the "credit_record" and merge it with "application_record"
(3) …

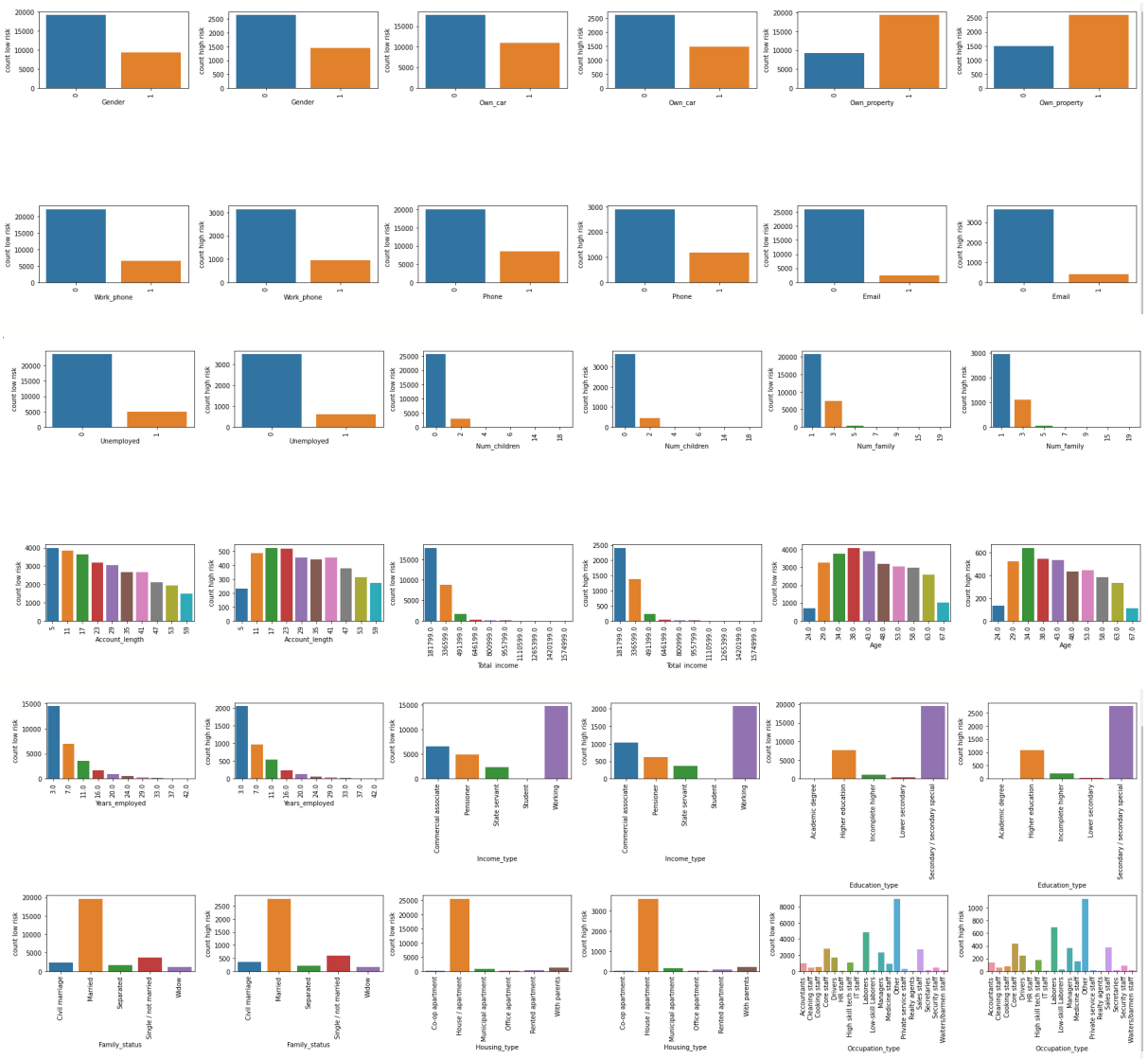
The order of (1) and (2) is wrong as we can't treat rows in "application_record" as duplicated without merging them with "credit_record" first. This step caused a lot of information loss and I failed to build any valuable models from this dataset originally (the F1 score is always < 0.25). Therefore, I choose to use the raw dataset and clean it in (2) -> (3) -> (1) order (all other cleaning methods inside remain the same as the author).
By doing this, now I have a dataset with a shape (32595,20) (all rows contain a unique ID) that is three times larger than the original dataset. And more importantly, I got more "high risk" cases which is the minority class and these data are important for me to build meaningful models.

(i) Explore the correlation between each specific feature and risk

```
High Risk and Low Risk Percentage:
0    0.874797
1    0.125203
Name: Target, dtype: float64
<matplotlib.axes._subplots.AxesSubplot at 0x7fc01ab21e90>
```



This shows that the dataset is imbalanced.



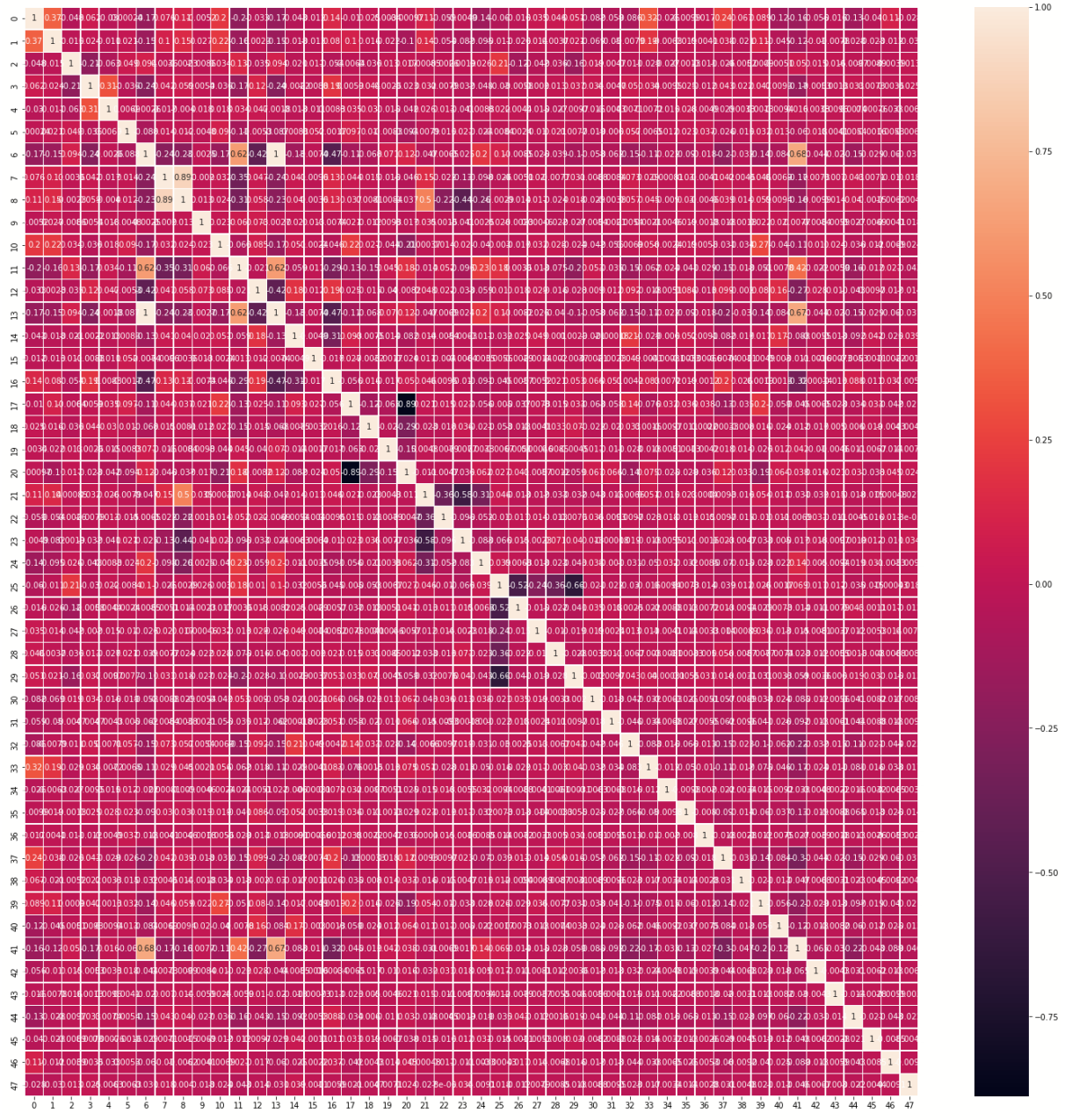
In the above picture, there are 6x6 (ignore ID, 18 features x 2 graphs) bar charts. In each row, columns {1,2}, {3,4}, {5,6} correspond to a feature where the left graph is “feature vs the number of low-risk” and the right graph is “feature vs the number of high-risk”. It is observed that each pair of graphs is very similar in shape meaning that it is expected that it won’t be possible to linearly classify whether a client is risky using only one feature.

Data Preprocessing

Before performing any further tasks, I first preprocessed my dataset. The dataset is clean already (no missing values and duplicates).

(1) dataset = dataset excludes the [“ID”] columns. (ID is meaningless in learning as it is randomly assigned to the applicant)
(2) drop duplicated on the dataset (just to double-check)
(3) dataset_x = dataset excludes the [“Target”] columns.
(4) dataset_y = [“Target”] columns in the dataset.
(5) One hot encoding on all categorical data of dataset_x. (I personally think education type can use Label Encoder such as 0 = lower secondary, 1 = higher education, 2 = academic degree, to represent their level of education. However, since other categorical columns haven’t this kind of “level” characteristic, I use one hot for all categorical columns for simplicity) After one hot encoding, the dataset now has 49 columns.
(6) split the dataset in training and testing set with test_size = 0.2 with shuffle and stratify = dataset_y.
(7) fit StandardScaler with train_x and transform it, then transform the test_x with the same scaler.

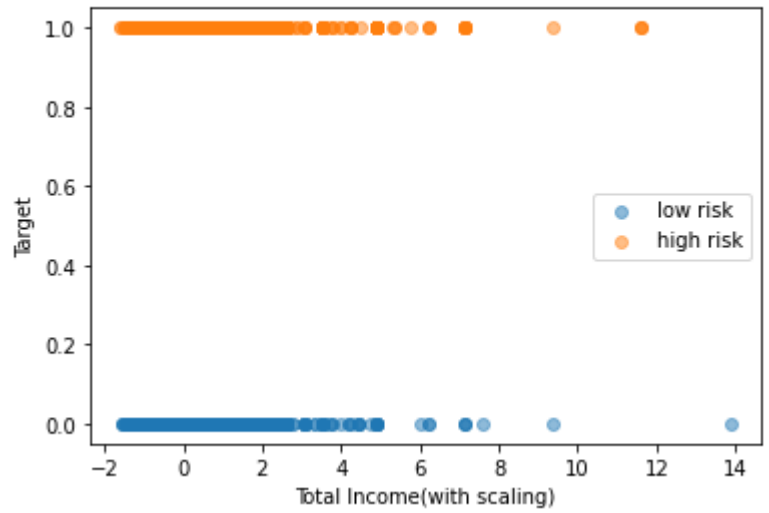
Heatmap of all train_x columns.



(i) Linear classifier on each feature vs risk

Before fitting the data into the classifier, I first under-sampling the training data so as to train the model without being biased toward the majority class.

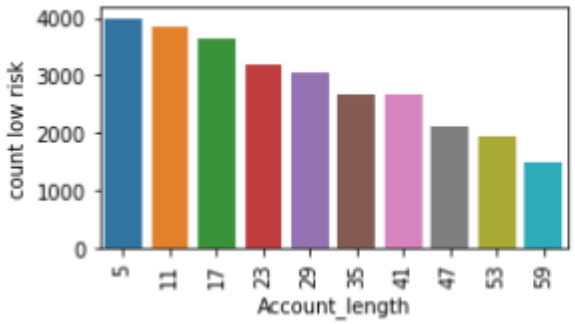
```
score: 0.519448698315467
<matplotlib.legend.Legend at 0x7fc018eb5c10>
```

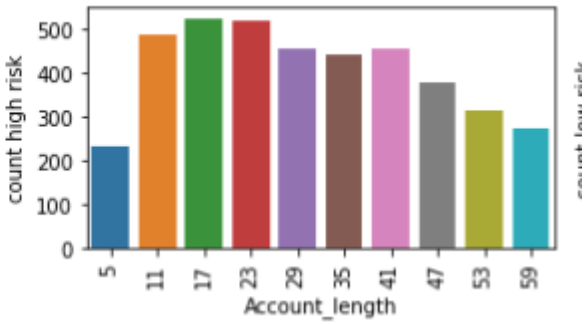


The training score is around 0.5. It is expected as mentioned in the previous section (i), it is impossible to linear separate these data points. For all the other features, they yield similar training scores.

```
score with Account_length: 0.5396630934150076
```

The score with “Account_length” is the highest. This may be due to the clear difference between the graph shown in the previous section (i).

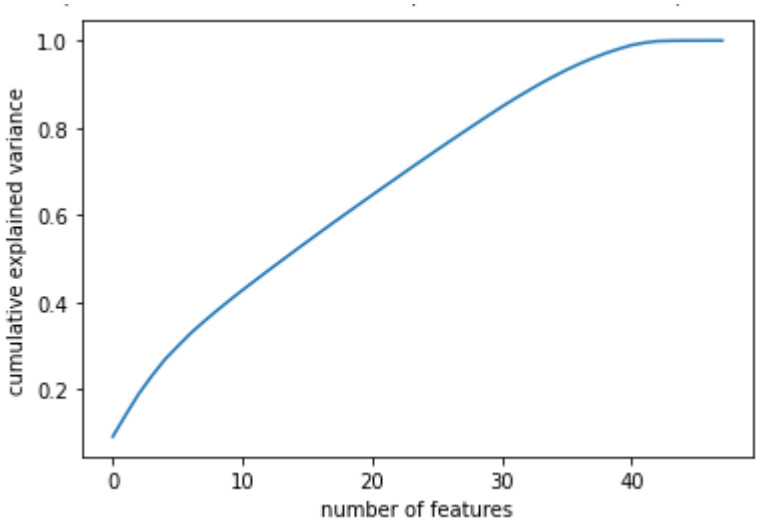




It is also understandable in common sense that it shares some correlation with the risk as “Account_length” refer to the “number of months credit card has been owned”. However, the score is still low. Therefore, for the classification tasks, I choose to use non-linear models such as random forest models and MLP later to study the complex relationship behind them.

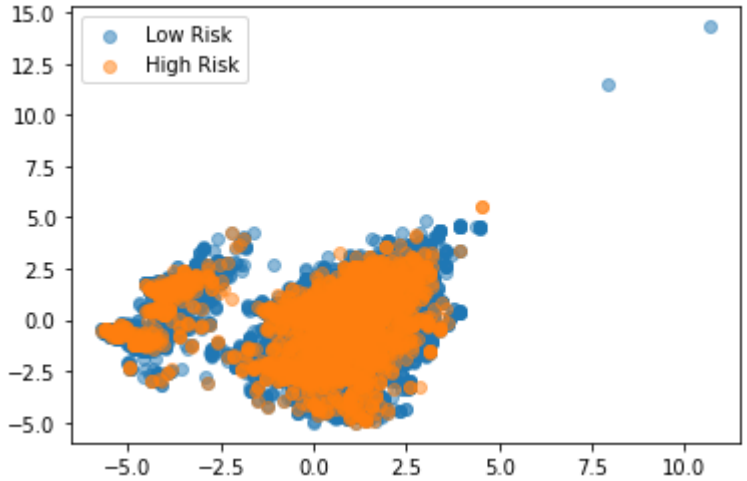
(ii) PCA

In this part, I first plot the number of features vs cumulative explained variance graph by using PCA on train_x.

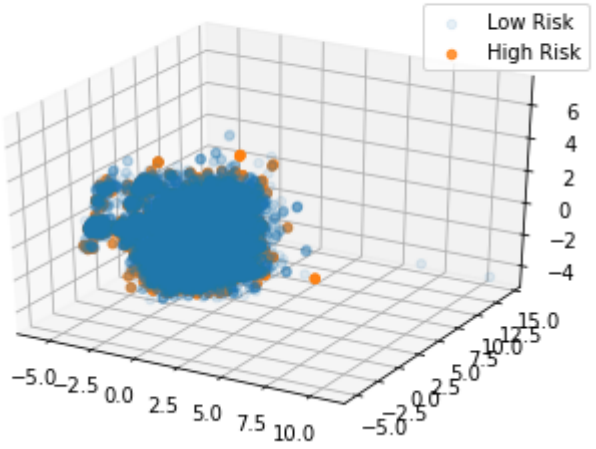



```
number of features remain using PCA: 34
```
(PoV >= 90%)

From x = 0 to x = 35, the curve is still steep. This means that PCA might not be helpful in our dataset in reducing the feature dimensions while retaining most information. It doesn't matter for training purposes as 48 columns are small enough which is fine. However, if we want to plot the 2D and 3D representations of the dataset, it only explained around 0.25 of the variance. Therefore, the graph looks bad like below:



```
<matplotlib.legend.Legend at 0x7fc018d4f250>
<Figure size 432x288 with 0 Axes>
```



We cannot separate low-risk and high-risk data points with a low number of the principal component. Therefore, I decided to not use PCA on my training set before training models to preserve all information.

(iii) Compare different methods of dealing with imbalanced data

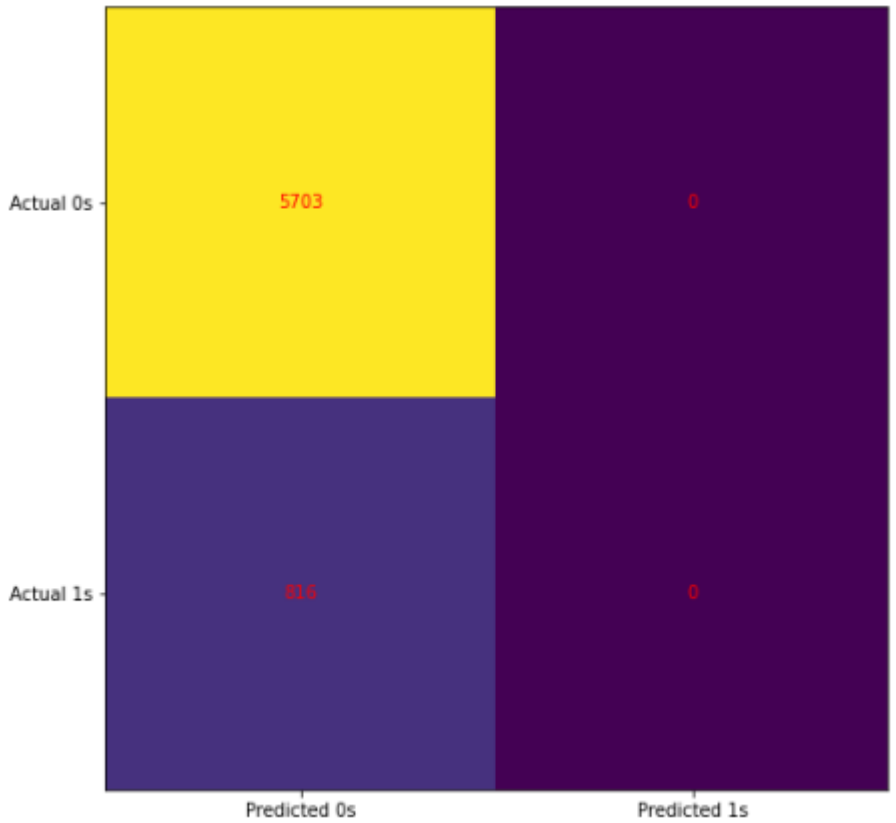
Based on the methods illustrated in https://www.kdnuggets.com/2017/06/7-techniques-handle-imbalanced-data.html, I choose a few ways from it to deal with imbalanced data. I apply these methods only on RandomForestClassifier for simplicity. After evaluating all these methods, I choose the best one and use it for building other models. First, I show what is the result when we blindly train a model without using any method to deal with imbalanced data:

```
accuracy: 0.8748274275195582
F1: 0.0
              precision    recall  f1-score   support

           0       0.87      1.00      0.93      5703
           1       0.00      0.00      0.00       816

    accuracy                           0.87      6519
   macro avg       0.44      0.50      0.47      6519
weighted avg       0.77      0.87      0.82      6519

True Positive:0
True Negative:5703
False Positve:0
False Negative:816
/usr/local/lib/python3.7/dist-packages/sklearn/metrics/_classification.py
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.7/dist-packages/sklearn/metrics/_classification.py
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.7/dist-packages/sklearn/metrics/_classification.py
  _warn_prf(average, modifier, msg_start, len(result))
```



The model always predicts 0 as it is the majority class and the model is biased.

Method 1: Add class weight to the minority class which means the model will get more penalty if they wrongly predict the minority class as the majority class.

I perform a grid search on the following parameter:

```
n_estimators = [int(x) for x in np.linspace(start=100,stop=500,num=50)]
max_features = ['auto', 'sqrt']
max_depth = [int(x) for x in np.linspace(10,100,num=10)]
min_samples_leaf = [int(x) for x in np.linspace(start=5,stop=25,num=5)]
bootstrap = [True, False]
class_weight =
[{0:1,1:5},{0:1,1:10},{0:1,1:15},{0:1,1:20},{0:1,1:25},{0:1,1:30},{0:1,
1:35},{0:1,1:40}]
```

I use RandomizedGridSearchCV with cv=KFold(3, shuffle=True, random_state=4211), n_iter=100, and scoring =”f1”. I prefer to use the random one instead of GridSearchCV because it is time-consuming to try all the combinations. This random grid search takes me only 30 mins to find the “best” params.

```
{'n_estimators': 271,
 'min_samples_leaf': 20,
 'max_features': 'sqrt',
 'max_depth': 40,
 'class_weight': {0: 1, 1: 10},
 'bootstrap': True}
```

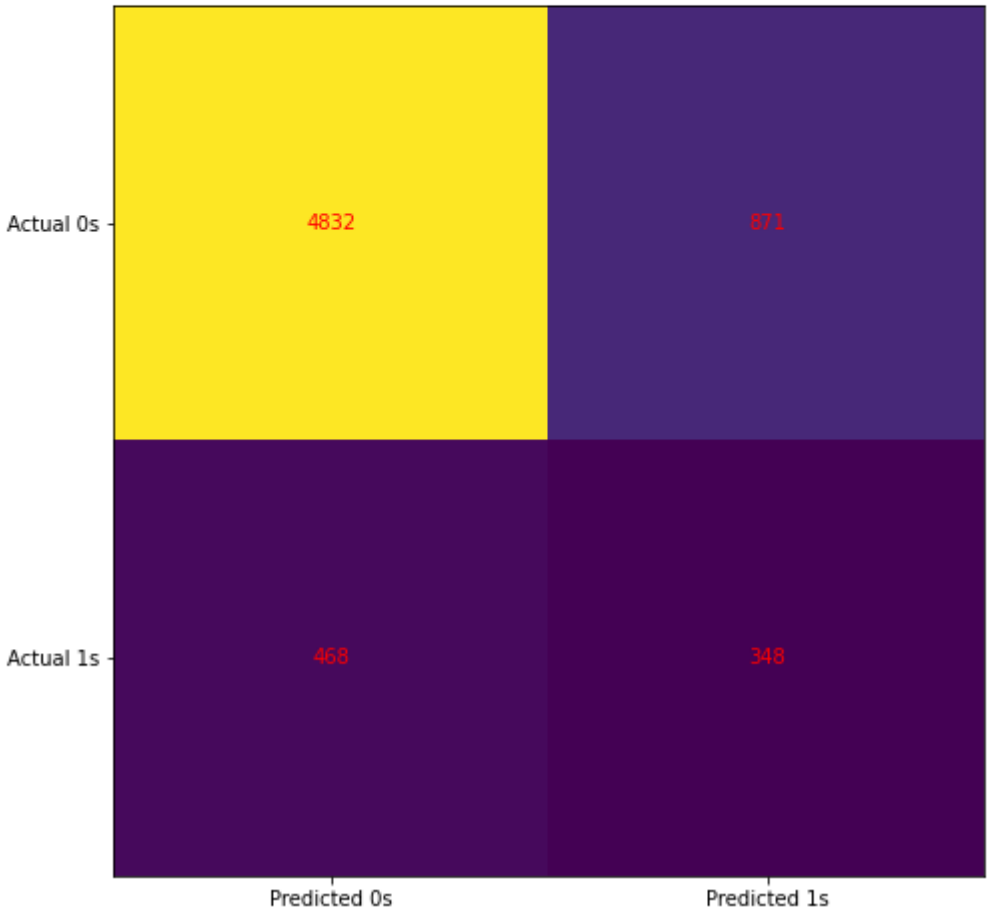
This is the best params it found. I have tried class_weight = {0:1, 1:i} where i = 5,6,7,8,9,10 manually and i=7 is the best one. Below are the results:

```
accuracy: 0.794600398834177
F1: 0.34201474201474197
              precision    recall  f1-score   support

           0       0.91      0.85      0.88      5703
           1       0.29      0.43      0.34       816

    accuracy                           0.79      6519
   macro avg       0.60      0.64      0.61      6519
weighted avg       0.83      0.79      0.81      6519

True Positive:348
True Negative:4832
False Positve:871
False Negative:468
```



Method 2: Use random oversampling to randomly duplicate the minority class data until the number of minority cases reaches a certain ratio (r) of the majority class. After that, perform under-sampling on the majority class cases (randomly drop the majority class cases until the number of cases is equal to the number of minority class cases).

I first use r = 0.5 and grid search the best params.

```
n_estimators = [int(x) for x in np.linspace(start=100,stop=500,num=50)]
max_features = ['auto', 'sqrt']
max_depth = [int(x) for x in np.linspace(10,100,num=10)]
min_samples_leaf = [int(x) for x in np.linspace(start=5,stop=25,num=5)]
bootstrap = [True, False]
```

Same setting on RandomizedGridSearchCV with Method 1. i.e. cv=KFold(3, shuffle=True, random_state=4211), n_iter=100, and scoring ="f1".

```
{'n_estimators': 483,
 'min_samples_leaf': 5,
 'max_features': 'sqrt',
 'max_depth': 100,
 'bootstrap': False}
```

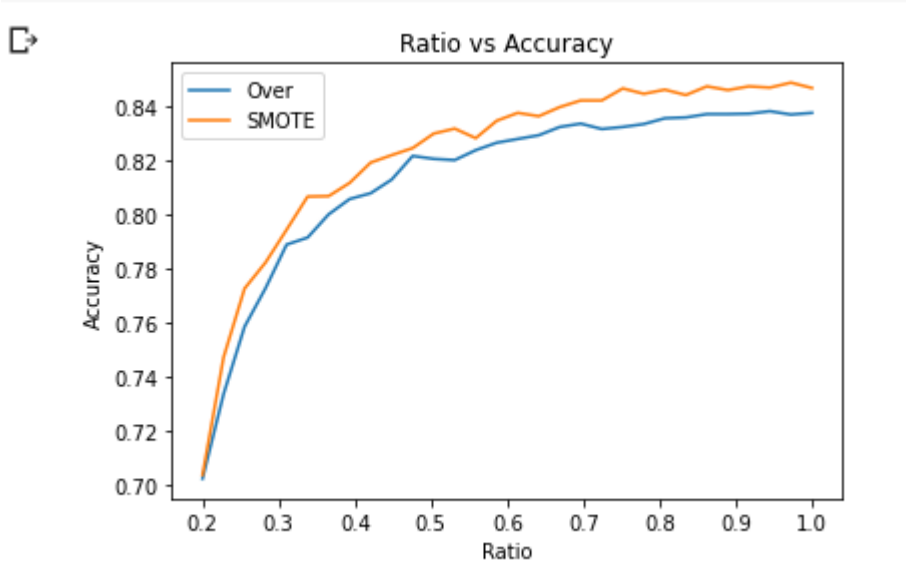
This is the best params. Before showing the result, I first illustrate and grid search on method 3 for better comparison.

Method 3: Similar to method 2, I use SMOTE oversampling which synthesizes new minority cases by using the information on the training set instead of duplicating the minority cases.

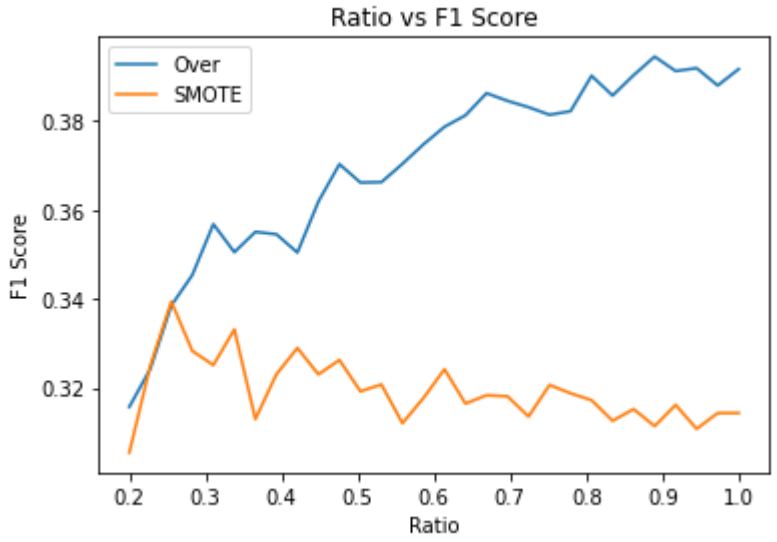
Perform the same grid search as method 2. The results are below:

```
{'n_estimators': 393,
 'min_samples_leaf': 5,
 'max_features': 'auto',
 'max_depth': 30,
 'bootstrap': False}
```

This is the best params. Before showing the results, I was also curious about how the ratio r affects the accuracy and F1 score on the testing set. By using the best params in method 2 and method 3, I plot the two graphs below:




```
<matplotlib.legend.Legend at 0x7f723aa94c50>
```



From the graph, the accuracy increase with r in both methods. This is expected as higher r meaning there are fewer majority class cases being removed in the latter undersampling.

For method 2, its F1 score also increases with r. It can be explained by the weight of the minority class is increased (duplicated minority cases -> weight increased) during learning so the model is better at classifying minority classes.

Interestingly, for method 3, its F1 score did not increase with r which is different from method 2. This may due to the "newly synthesized minority class cases" are too fake that doesn't well represent minority cases in the testing set so there is not much change.

Even though the average accuracy of method 3 is higher than method 2, I prefer method 2 as it has a higher F1 score on average than method 3 which is much more meaningful than the accuracy value when dealing with imbalanced data.

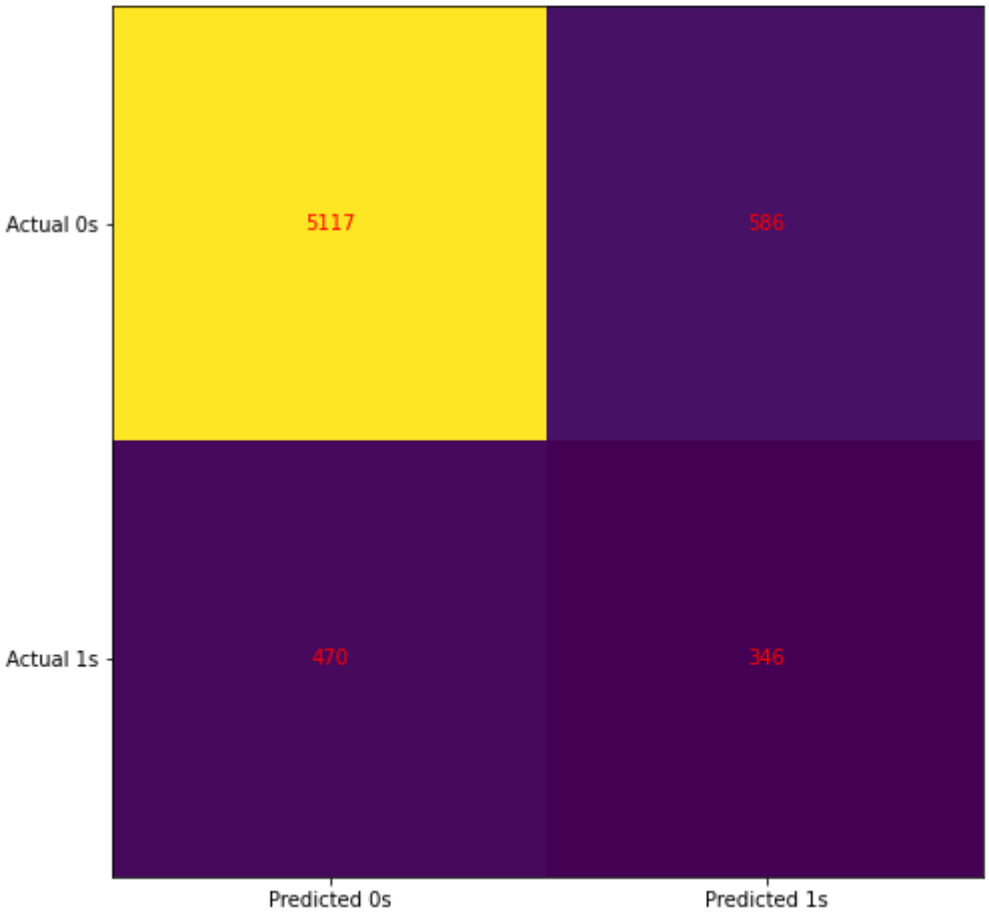
I search around r = 0.85 to 0.9, and 0.88 is the best one using method 2. Below is the result:

```
accuracy: 0.8380119650253106
F1: 0.39588100686498856
              precision    recall  f1-score   support

           0       0.92      0.90      0.91      5703
           1       0.37      0.42      0.40       816

    accuracy                           0.84      6519
   macro avg       0.64      0.66      0.65      6519
weighted avg       0.85      0.84      0.84      6519

True Positive:346
True Negative:5117
False Positve:586
False Negative:470
```



It has F1 ~ 0.39. Therefore, method 2 seems to be the best method in this dataset. I am going to use method 2 with r = 0.88 for training other models.

(iv) Train machine learning models to tackle the classification task

Apart from the random forest classifier I used in (iii), I also trained two more models to compare their results.

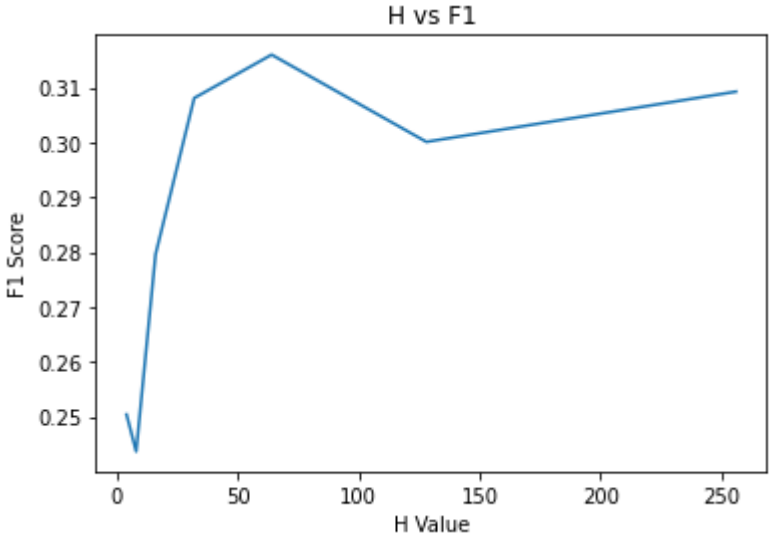
(1) MLP
I first grid search on some parameters with the same cv and scoring setting in (iii) for building an MLPClassifier. The parameters are below:

```
hidden_layer_sizes =
[(4,4,4),(8,8,8),(16,16,16),(32,32,32),(64,64,64),(24,12,6),(24,12),(12
,6)]
learning_rate_init = [0.005,0.001,0.01]
max_iter = [500,1000,1500]
early_stopping = [True]
random_state = [0]
```

As the number of combinations is relatively small (72 combinations), I use GridSearchCV instead of the random one to go through all the combinations. The best param is

```
{'early_stopping': True,
 'hidden_layer_sizes': (64, 64, 64),
 'learning_rate_init': 0.005,
 'max_iter': 500,
 'random_state': 0}
```

However, it seems like I can get a higher score by using more units per hidden layer. Therefore, I also test hidden_layer_sizes = (128,128,128) and (256,256,256) with the other parameters the same as the best param above and get the score on the testing set. The graph is below:



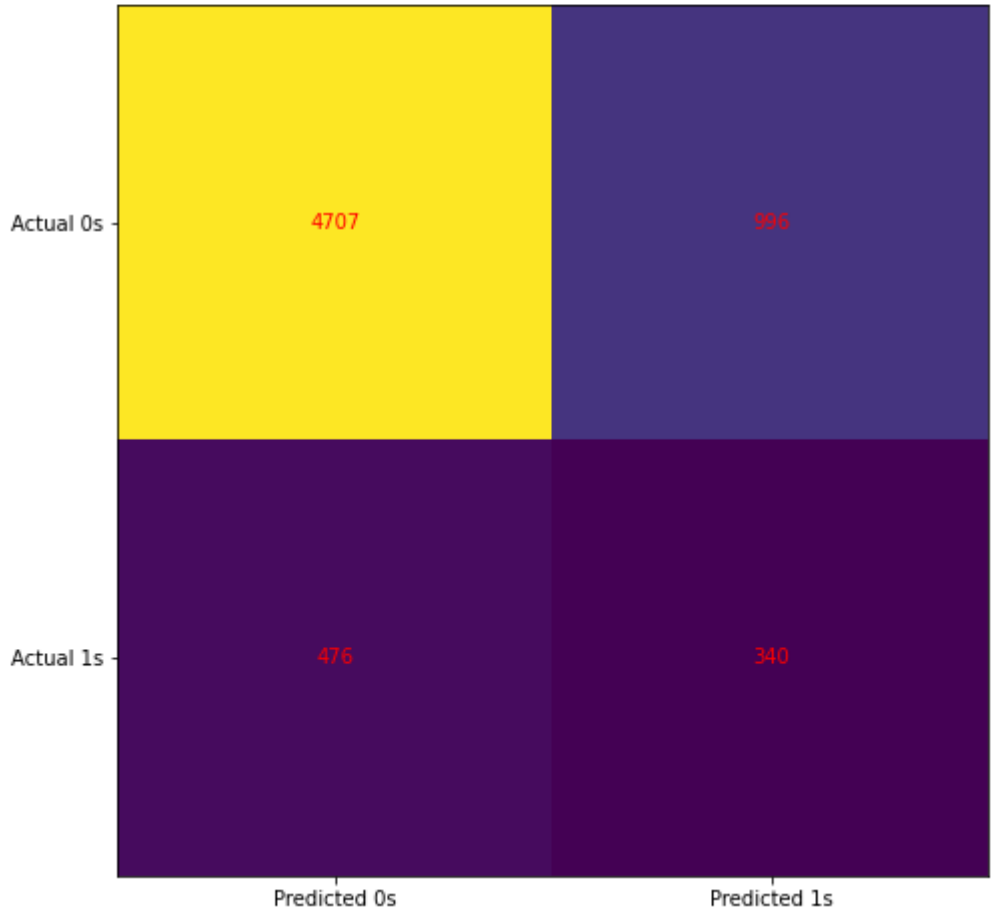
It Seems (64,64,64) is already optimal. Using the best param setting, I get the result:

```
accuracy: 0.7741984967019482
F1: 0.3159851301115242
              precision    recall  f1-score   support

           0       0.91      0.83      0.86      5703
           1       0.25      0.42      0.32       816

    accuracy                           0.77      6519
   macro avg       0.58      0.62      0.59      6519
weighted avg       0.83      0.77      0.80      6519

True Positive:340
True Negative:4707
False Positve:996
False Negative:476
```



It is not better than a random forest classifier.

(2) SVM

I perform a grid search on the following parameters (same cv and scoring with previous grid search):

```
C = [0.1,1,10,100]
kernel = ["rbf"] # non-linear kernel
```

Best param = `{'C': 100, 'kernel': 'rbf'}`

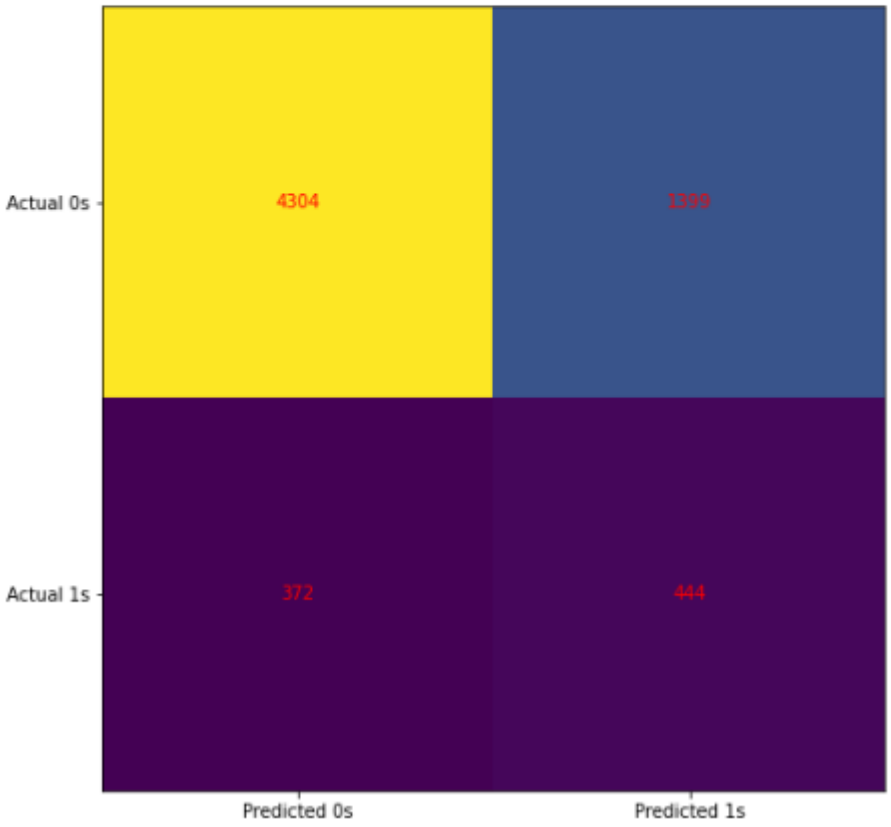
Result using best param:

```
accuracy: 0.7283325663445314
F1: 0.3339601353892441
              precision    recall  f1-score   support

           0       0.92      0.75      0.83      5703
           1       0.24      0.54      0.33       816

    accuracy                           0.73      6519
   macro avg       0.58      0.65      0.58      6519
weighted avg       0.84      0.73      0.77      6519

True Positive:444
True Negative:4304
False Positve:1399
False Negative:372
```



It is still not better than a random forest classifier. Even if it seems would have an even better result using a larger “C”, it is still not good as its training time is too long while most of the iterations in other models take less than 10s only to train. It might be because the training set is quite large and SVM non-linear algorithm complexity largely depends on the number of samples.

Larger C needs more training time:

```
Fitting 3 folds for each of 4 candidates, totalling 12 fits
[CV 1/3] END .................C=0.1, kernel=rbf;, score=0.610 total time= 1.9min
[CV 2/3] END .................C=0.1, kernel=rbf;, score=0.606 total time= 2.0min
[CV 3/3] END .................C=0.1, kernel=rbf;, score=0.607 total time= 2.4min
[CV 1/3] END ...................C=1, kernel=rbf;, score=0.683 total time= 1.8min
[CV 2/3] END ...................C=1, kernel=rbf;, score=0.677 total time= 1.5min
[CV 3/3] END ...................C=1, kernel=rbf;, score=0.678 total time= 1.5min
[CV 1/3] END ..................C=10, kernel=rbf;, score=0.761 total time= 2.3min
[CV 2/3] END ..................C=10, kernel=rbf;, score=0.764 total time= 2.2min
[CV 3/3] END ..................C=10, kernel=rbf;, score=0.762 total time= 2.6min
[CV 1/3] END .................C=100, kernel=rbf;, score=0.815 total time= 4.8min
[CV 2/3] END .................C=100, kernel=rbf;, score=0.815 total time= 4.5min
[CV 3/3] END .................C=100, kernel=rbf;, score=0.814 total time= 4.4min
{'C': 100, 'kernel': 'rbf'}
```

(3) Anomaly Detection Using OneClassSVM

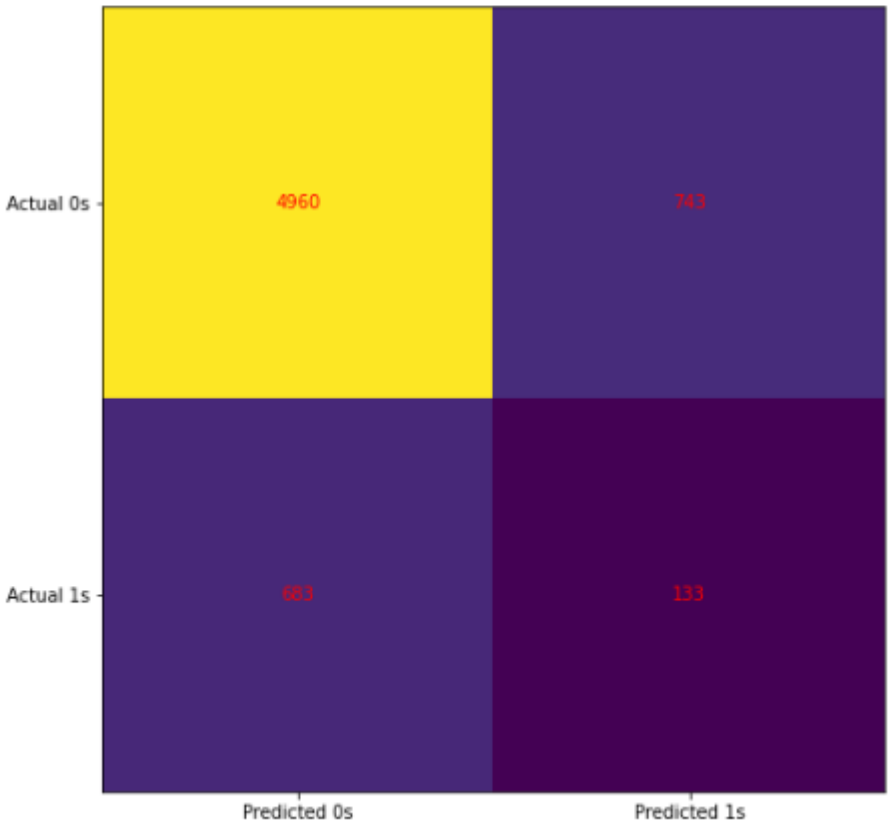
It is a method that is illustrated in the lecture (SVM lecture note slide #28). It is an unsupervised algorithm that takes only the features of the majority class as training data. After training, it will treat all data that seem not similar to its training data as “anomaly” and output -1 for them, otherwise, output 1. I use OneClassSVM for this method. For the nu parameter, I set it as the proportion of class 1 in training data. As minority class is not important in training now, I use train_x[train_y==0] as the training input without using the resampling method. The result is as follows:

```
accuracy: 0.7812547936800123
F1: 0.15721040189125296
              precision    recall  f1-score   support

           0       0.88      0.87      0.87      5703
           1       0.15      0.16      0.16       816

    accuracy                           0.78      6519
   macro avg       0.52      0.52      0.52      6519
weighted avg       0.79      0.78      0.78      6519

True Positive:133
True Negative:4960
False Positve:743
False Negative:683
```



Seems this method is not suitable for our dataset. That might indicate treating the minority class as an outlier is not suitable for this dataset.

(4) Light GBM Classifier

It is a variant of the decision tree (Decision Tree last slide) which is a state-of-art classifier. It has high efficiency and training speed. I first perform a random grid search on the following parameters with the same cv and scoring as previous, and also use the resampled training set:

```
n_estimators = [int(x) for x in np.linspace(start=200,stop=500,num=50)]
max_depth = [int(x) for x in np.linspace(10,100,num=10)]
```

```
min_samples_leaf = [int(x) for x in np.linspace(start=5,stop=25,num=5)]
learning_rate = [0.5,0.1,0.05,0.01]
min_child_weight = [int(x) for x in range(10)]
subsample = [x for x in np.linspace(start=0.2,stop=1,num=5)]

Best param = {'subsample': 0.8,
 'n_estimators': 420,
 'min_samples_leaf': 15,
 'min_child_weight': 0,
 'max_depth': 100,
 'learning_rate': 0.5}
```

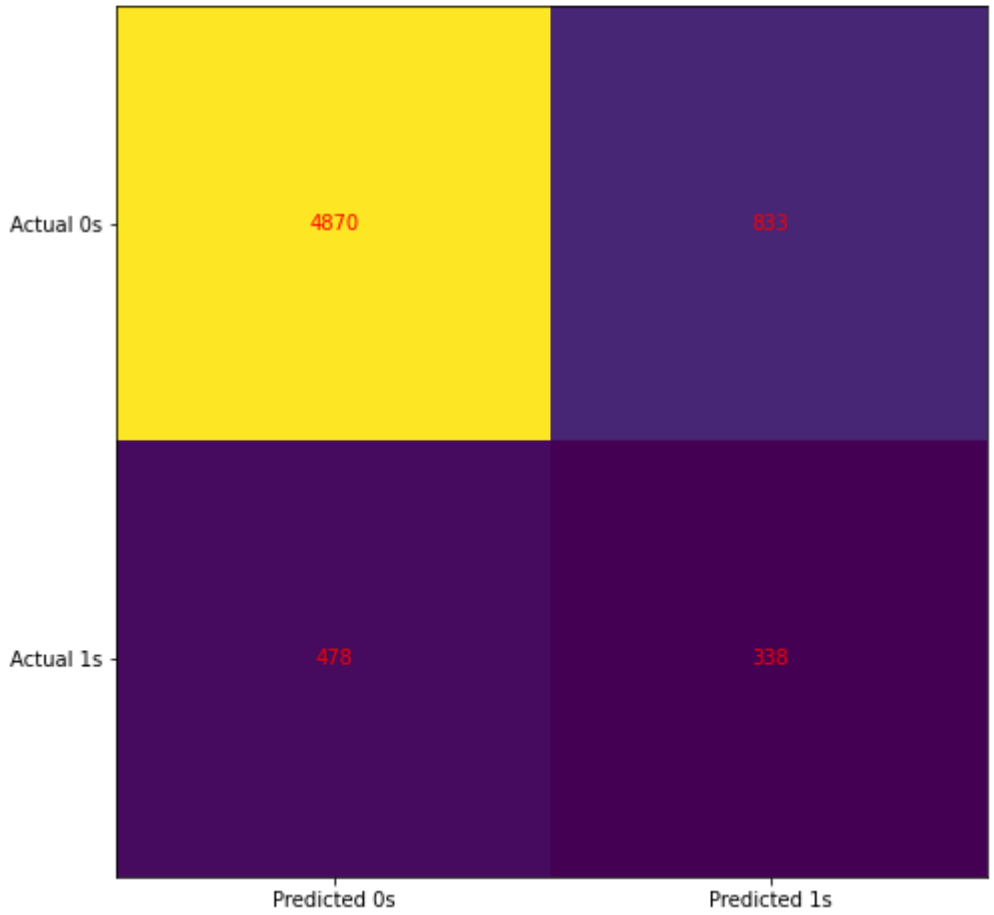
Using the best param,

```
accuracy: 0.7988955361251726
F1: 0.3402113739305486
              precision    recall  f1-score   support

           0       0.91      0.85      0.88      5703
           1       0.29      0.41      0.34       816

    accuracy                           0.80      6519
   macro avg       0.60      0.63      0.61      6519
weighted avg       0.83      0.80      0.81      6519

True Positive:338
True Negative:4870
False Positve:833
False Negative:478
```



Conclusion for (iv): RandomForestClassifier is the best one as it yields 0.39 F1 scores. The remaining is the LightGBM (0.34), SVM (0.33), MLP (0.32), and OneClassSVM (0.16). Even though the recall value for 1 of SVM is the largest (0.54), its precision is low. Put in real life, it means that the algorithm detects a lot of users as “high-risk”, but only a small portion of them is really high-risk which required more manpower to filter low-risk cases manually. So there is a trade-off here and I would consider the F1 score more important as it is more comprehensive. Moreover, even though in

my training LightGBM get only a 0.34 F1 score, its training speed is the fastest among all models (less than 5s). This can be a reason why it is popular in classification.

(v) Clustering

Step:

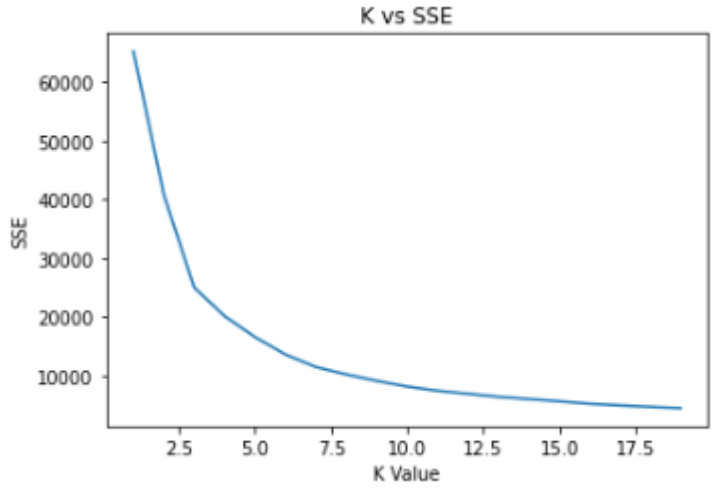
(1) data = dataset with column[“Age”,”Total_income”]

(2) data = StandardScaler().fit_transform(data)

(3) model = KMeans(n_clusters=k) (sklearn)

(4) model.fit(data)

Elbow Curve:



3 seems to be the “Elbow”.

Use k = 3,

```
cluster = [[-0.7917972  -0.24169707]
 [ 1.01680406 -0.37798192]
 [-0.03932569  1.78891388]]
```

The clusters are not well separated. But from the plot, we can notice that the data distribution is like a triangle shape, which means young and old people have less total income on average and people in middle age has higher total income on average. The three centroids also explained this phenomenon. To explain this, young people have low total income as they have just yet working for a job while old people have retired and have no more income source. People in middle age save a lot as they have to prepare for retirement.

Possibles Improvement

In this project, there are two possibles improvement for getting higher F1 scores. First, I believe there can be some slight improvement in training a RandomForestClassifier by using a normal grid search instead of the random one to search for more possible combinations. But there might not be a big improvement as I think small changes in parameter values would not have a big difference.
Second, I believe there can be a big improvement by increasing the size of the dataset. As I mentioned before, I originally used the original dataset with around 10k rows and no matter how I tuned the parameters it never reached F1 scores higher than 0.25 on the testing set. Meanwhile, after I changed the way the author cleaned the dataset and retrieved back the lost information (30k rows), the F1 score significantly improved to 0.39 F1 score. Therefore, I think if we can find more data, especially the minority cases, then the model will have a better prediction.

Appendix: Previous Best Result Using Original Dataset

Using the original dataset (the one with 9709 rows), using the same preprocessing, and using the best param after grid search on randomForestClassifier and r = 0.25:

```
RandomForestClassifier(n_estimators=434,max_features="sqrt",max_depth=70,min_samples_leaf=5,bootstrap=False,random_state=0)
```

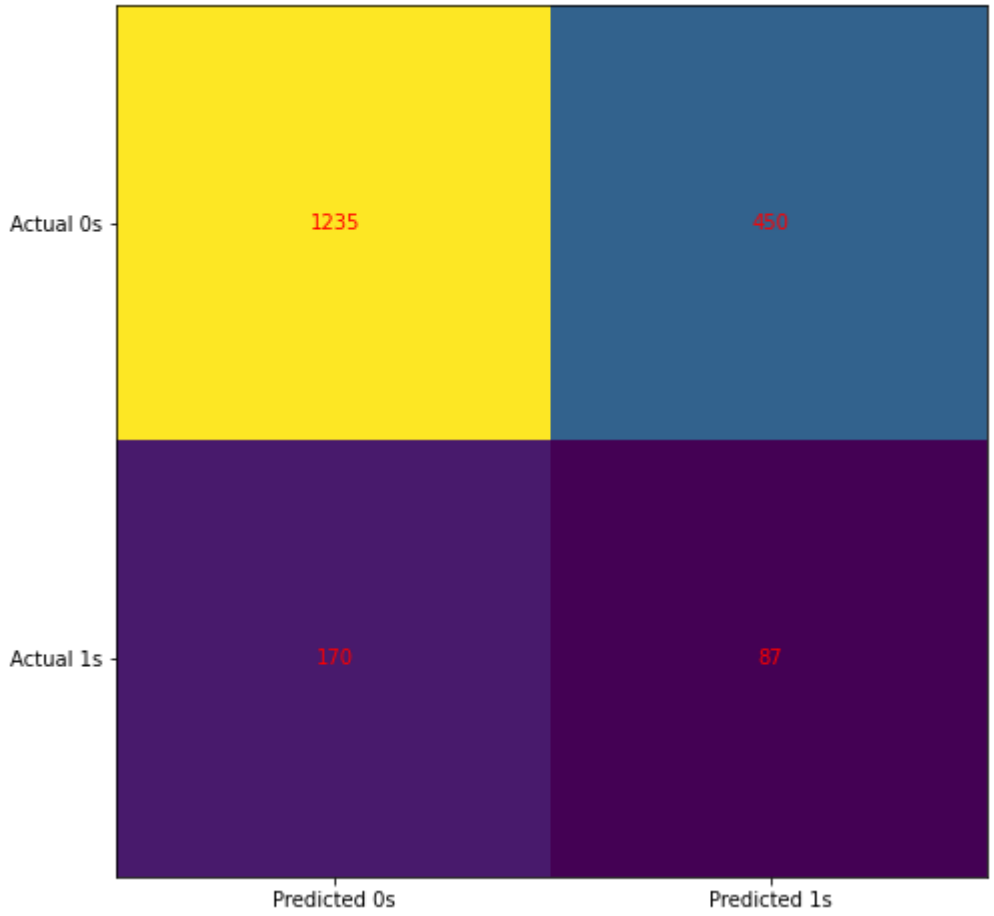
I can only get

```
accuracy: 0.6807415036045315
F1: 0.21914357682619648
              precision    recall  f1-score   support

           0       0.88      0.73      0.80      1685
           1       0.16      0.34      0.22       257

    accuracy                           0.68      1942
   macro avg       0.52      0.54      0.51      1942
weighted avg       0.78      0.68      0.72      1942

True Positive:87
True Negative:1235
False Positve:450
False Negative:170
```



It shows that there is big improvement in F1 by using a larger dataset (0.21 vs 0.39).