Assignment 3: Data Mining in Action

31250 Introduction to Data Analytics

Alexander Justin Dealson | 24650196

Data Mining Problem:

In this report, there are numerous attributes in the bank marketing dataset presented to solve a business problem. The business problem in this report is about predicting whether a bank customer will subscribe to a term deposit or not based on the given attributes.

Input:

For this task we are given several bank marketing datasets. The first dataset is used to build the model. The first dataset contains all the features and their target attribute (whether a client subscribed or not). Whereas the second datasets, are the dataset that we will use to predict whether a client will subscribe or not. This dataset only contains the attribute but no target attribute. The attribute that we used are for instance, age, job, marital, etc. However, out of all the attributes, the attributes that play significant roles in affecting the prediction is duration and nr.employed.

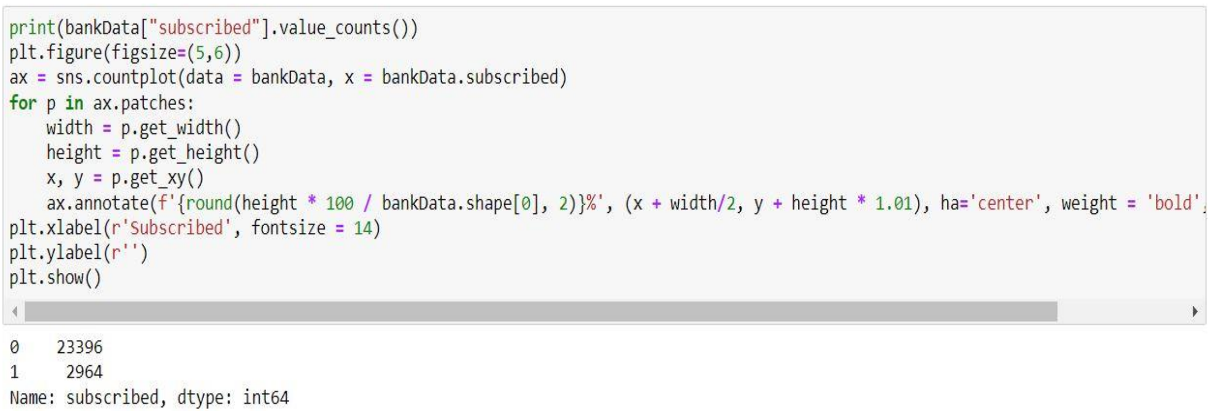
Output:

The outcome of the problems is the prediction of whether a person will subscribe to a term deposit or not based on their information. These predictions are made from the second bank marketing datasets which is the datasets that has all the features but no target attribute.

Data Preparation (Data Pre-Processing and Data Exploration):

1.Data Exploration

Before we made the model, the first step we need to do is to explore the data and pre-process the dataset that has both the feature and target attribute. To explore the data, we can first see the target class distribution by using this python code:



```
print(bankData["subscribed"].value_counts())
plt.figure(figsize=(5,6))
ax = sns.countplot(data = bankData, x = bankData.subscribed)
for p in ax.patches:
    width = p.get_width()
    height = p.get_height()
    x, y = p.get_xy()
    ax.annotate(f'{round(height * 100 / bankData.shape[0], 2)}%', (x + width/2, y + height * 1.01), ha='center', weight = 'bold',
plt.xlabel(r'Subscribed', fontsize = 14)
plt.ylabel(r'')
plt.show()
```

```
0    23396
1     2964
Name: subscribed, dtype: int64
```

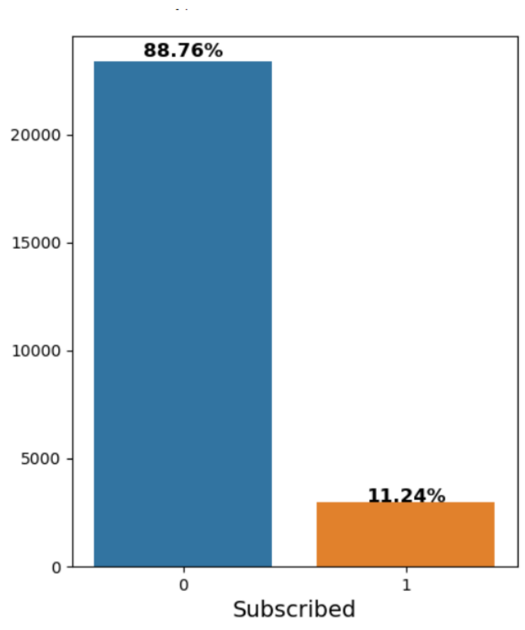
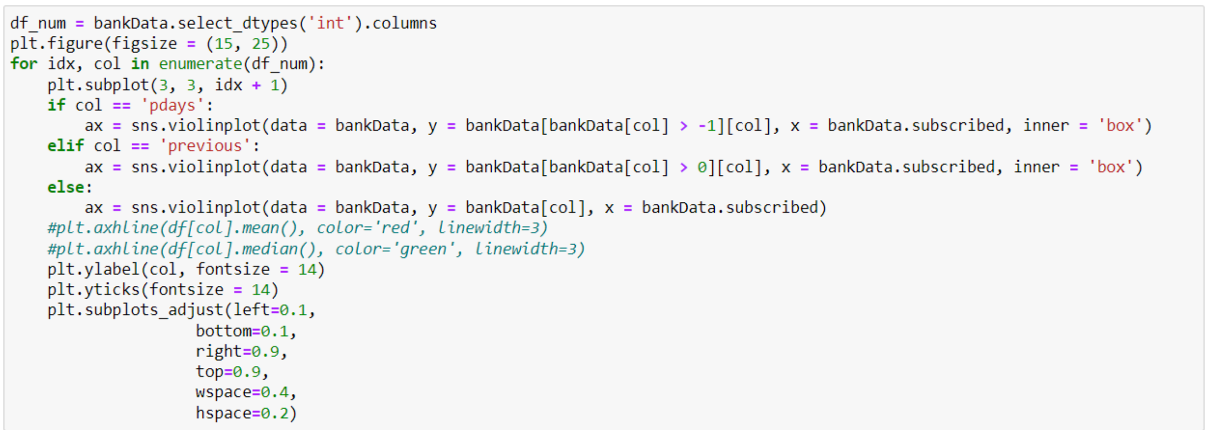


Figure 1. Target class Distribution

It can be seen that the target class which is the subscribed attribute is an unbalance class as the value is unequal. The bar chart is mostly comprised of the value 0 (no subscribed) whereas only 11.24% has the value 1 (yes subscribed). Furthermore, to explore our dataset further we can also visualize the distribution of numerical attribute against the target class by using violin plot. We can use this python code:



```python
df_num = bankData.select_dtypes('int').columns
plt.figure(figsize = (15, 25))
for idx, col in enumerate(df_num):
    plt.subplot(3, 3, idx + 1)
    if col == 'pdays':
        ax = sns.violinplot(data = bankData, y = bankData[bankData[col] > -1][col], x = bankData.subscribed, inner = 'box')
    elif col == 'previous':
        ax = sns.violinplot(data = bankData, y = bankData[bankData[col] > 0][col], x = bankData.subscribed, inner = 'box')
    else:
        ax = sns.violinplot(data = bankData, y = bankData[col], x = bankData.subscribed)
    #plt.axhline(df[col].mean(), color='red', linewidth=3)
    #plt.axhline(df[col].median(), color='green', linewidth=3)
    plt.ylabel(col, fontsize = 14)
    plt.yticks(fontsize = 14)
    plt.subplots_adjust(left=0.1,
                    bottom=0.1,
                    right=0.9,
                    top=0.9,
                    wspace=0.4,
                    hspace=0.2)
```

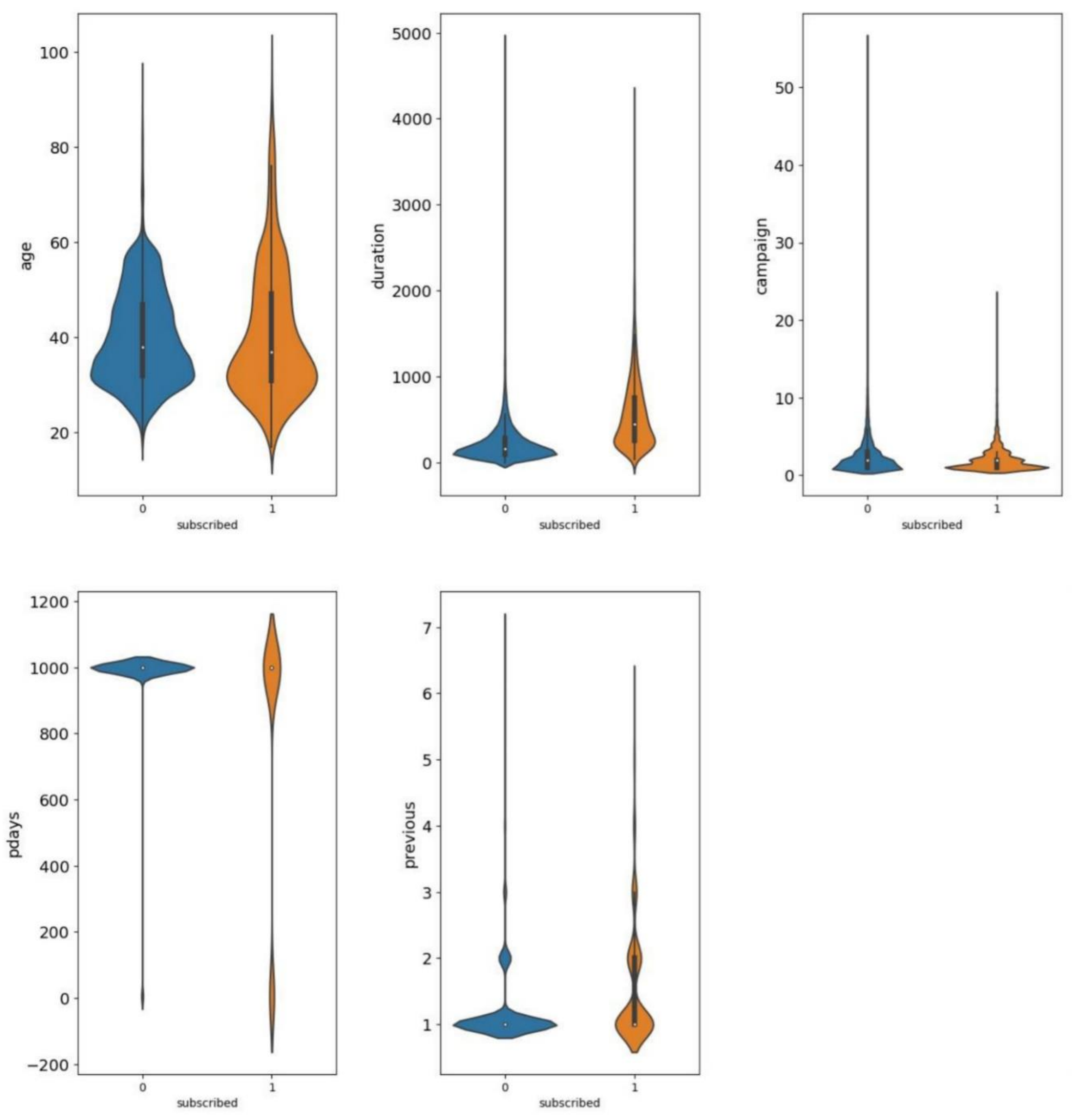
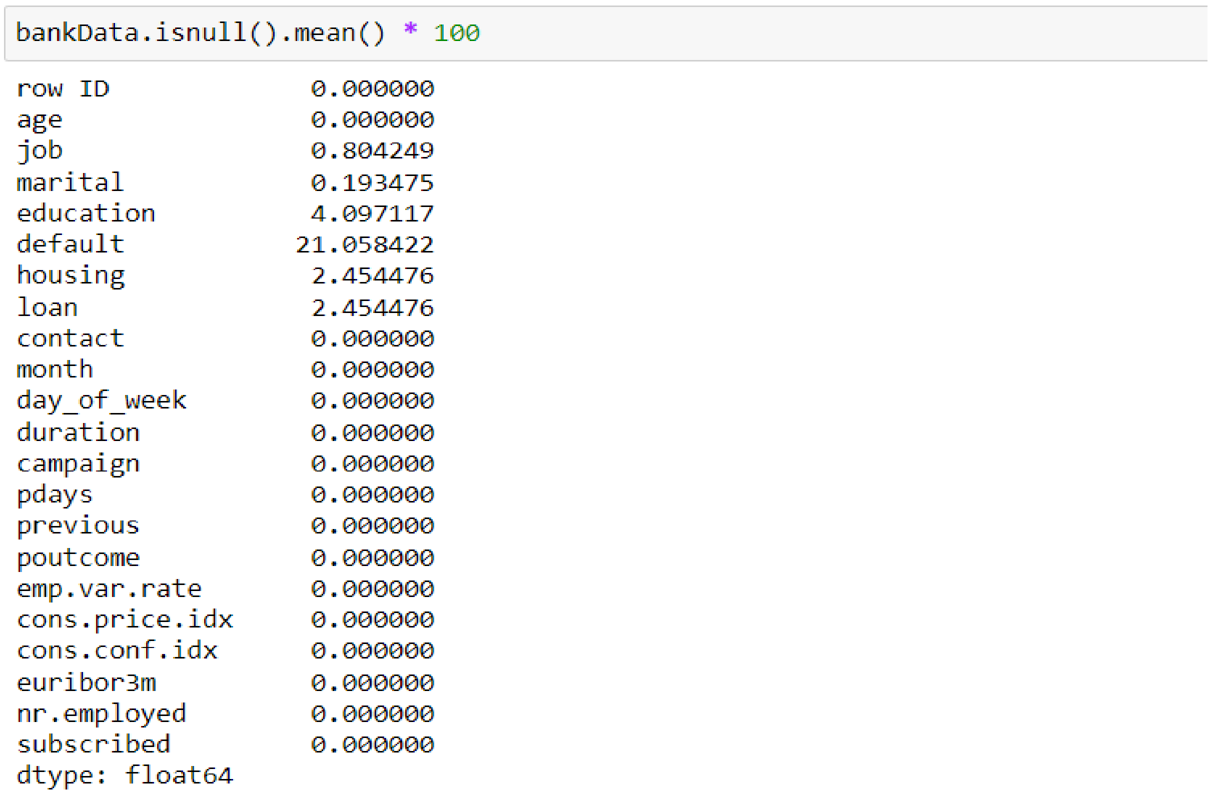


Figure 2. Violin plot Numerical attribute against subscribe attribute

Based on Figure 2, we can infer that those who subscribed to a term deposit tends to have a longer last contact duration than those who did not subscribed to a term deposit. This will be useful for gaining an information and assumption based on these attributes. In the violin plot above, we can have a general idea on the distribution of numerical attribute based on their decision to subscribe or no subscribed to a term deposit and we can derive an assumption from this chart.

2.Data Pre-processing

Before we get into our data pre-processing part, we must first solve the problem of missing values in our dataset. In the Diagram (Figure 3) below, we can see where the missing values lie in our dataset and their percentage. If the percentage of missing values exceeds certain threshold (10%) then, that specific attribute should be drop instead of imputing their missing values.



```
bankData.isnull().mean() * 100
```

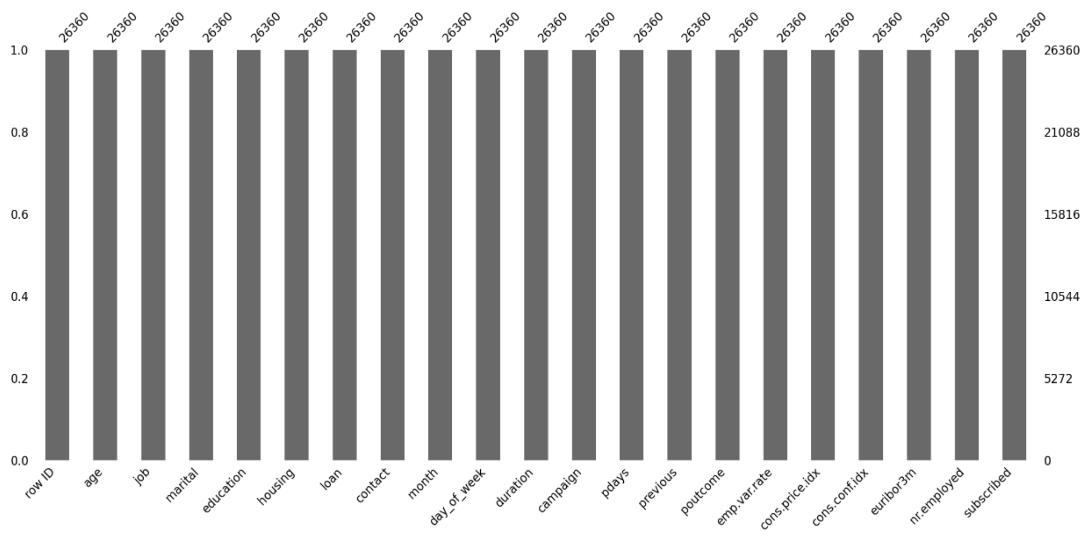
```
row ID             0.000000
age                0.000000
job                0.804249
marital            0.193475
education          4.097117
default           21.058422
housing            2.454476
loan               2.454476
contact            0.000000
month              0.000000
day_of_week        0.000000
duration           0.000000
campaign           0.000000
pdays              0.000000
previous           0.000000
poutcome           0.000000
emp.var.rate       0.000000
cons.price.idx     0.000000
cons.conf.idx      0.000000
euribor3m          0.000000
nr.employed        0.000000
subscribed         0.000000
dtype: float64
```

Figure 3. Missing values

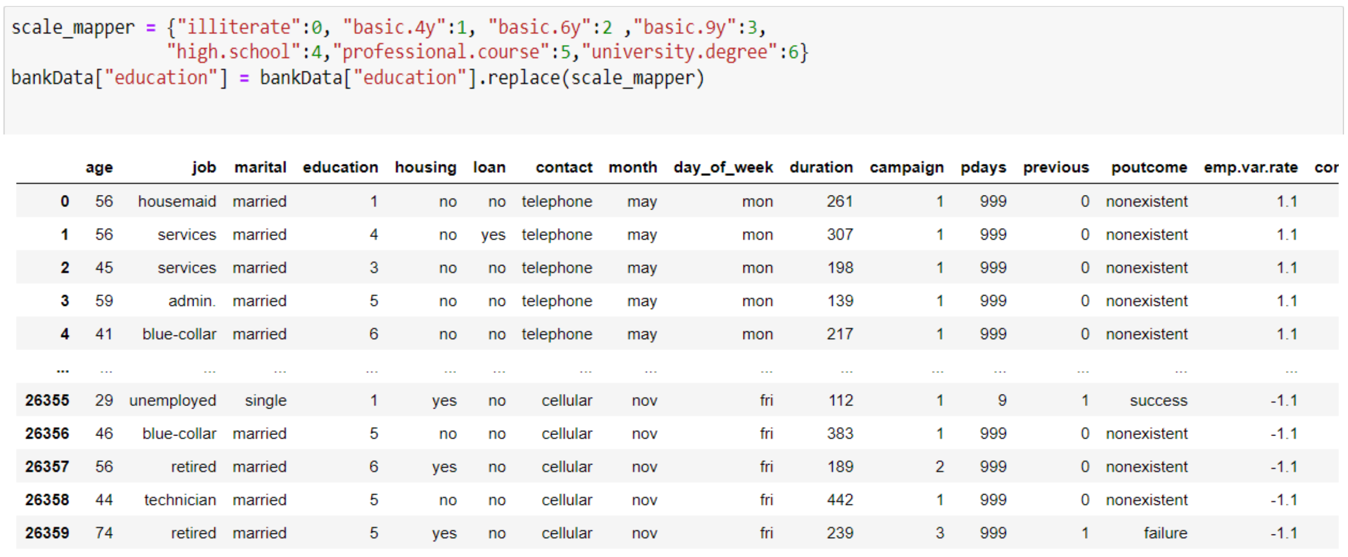
Based on Figure 3, the attribute default has a missing value of 21% which exceeds our threshold, thus it should be drop from our dataset as it may affect our data pre-processing and analysis. Whereas for the other missing values because they are categorical data, we can use a method called SimpleImputer from scikitlearn library to impute it with the most frequent values (mode). After this imputing process, we must recheck our dataset again.

```
msno.bar(bankData)
```

```
<AxesSubplot:>
```



Once our dataset has no missing value, then we can proceed to converting categorical attribute to numerical attribute because for scikit-learn or python, most of the algorithm can only accept data in numeric form. In converting categorical attribute, we must distinguish whether a certain categorical attribute is ordinal type or nominal type as the model we will build will impose a rank if the attribute is ordinal type. To convert ordinal categorical attribute to numerical attribute we can use this line of python code:

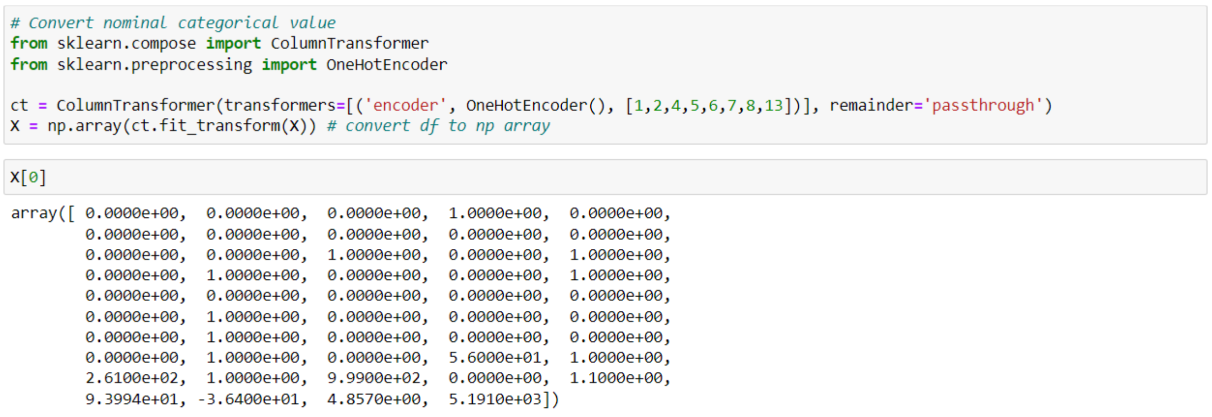


```
scale_mapper = {"illiterate":0, "basic.4y":1, "basic.6y":2 ,"basic.9y":3,
                "high.school":4,"professional.course":5,"university.degree":6}
bankData["education"] = bankData["education"].replace(scale_mapper)
```

	age	job	marital	education	housing	loan	contact	month	day_of_week	duration	campaign	pdays	previous	poutcome	emp.var.rate	cor
0	56	housemaid	married	1	no	no	telephone	may	mon	261	1	999	0	nonexistent	1.1	
1	56	services	married	4	no	yes	telephone	may	mon	307	1	999	0	nonexistent	1.1	
2	45	services	married	3	no	no	telephone	may	mon	198	1	999	0	nonexistent	1.1	
3	59	admin.	married	5	no	no	telephone	may	mon	139	1	999	0	nonexistent	1.1	
4	41	blue-collar	married	6	no	no	telephone	may	mon	217	1	999	0	nonexistent	1.1	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
26355	29	unemployed	single	1	yes	no	cellular	nov	fri	112	1	9	1	success	-1.1	
26356	46	blue-collar	married	5	no	no	cellular	nov	fri	383	1	999	0	nonexistent	-1.1	
26357	56	retired	married	6	yes	no	cellular	nov	fri	189	2	999	0	nonexistent	-1.1	
26358	44	technician	married	5	no	no	cellular	nov	fri	442	1	999	0	nonexistent	-1.1	
26359	74	retired	married	5	yes	no	cellular	nov	fri	239	3	999	1	failure	-1.1	

Figure 4. Converting ordinal categorical attribute to numeric

In Figure 4, the attribute education has been converted into a numerical attribute with a rank ordering in the value. Thus, the model will capture a relationship between each value. After converting the ordinal attribute, we then need to convert the nominal attribute and to do this we need to use a method called OneHotEncoder from scikitlearn library. The attributes that will be converted are job, marital, housing, loan, contact, month, day_of_week, month, and poutcome.



```
# Convert nominal categorical value
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder

ct = ColumnTransformer(transformers=[('encoder', OneHotEncoder(), [1,2,4,5,6,7,8,13])], remainder='passthrough')
X = np.array(ct.fit_transform(X)) # convert df to np array
```

```
X[0]
```

```
array([ 0.0000e+00,  0.0000e+00,  0.0000e+00,  1.0000e+00,  0.0000e+00,
        0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00,
        0.0000e+00,  0.0000e+00,  1.0000e+00,  0.0000e+00,  1.0000e+00,
        0.0000e+00,  1.0000e+00,  0.0000e+00,  0.0000e+00,  1.0000e+00,
        0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00,
        0.0000e+00,  1.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00,
        0.0000e+00,  1.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00,
        0.0000e+00,  1.0000e+00,  0.0000e+00,  5.6000e+01,  1.0000e+00,
        2.6100e+02,  1.0000e+00,  9.9900e+02,  0.0000e+00,  1.1000e+00,
        9.3994e+01, -3.6400e+01,  4.8570e+00,  5.1910e+03])
```

Figure 5. Converting nominal categorical attribute to numeric

After we convert all the categorical attribute to numeric, we can then see the correlation of all input attributes to the subscribe attribute (target class). To do this, we can use pandas.DataFrame.corr() and then map it to heatmap to see the correlation.

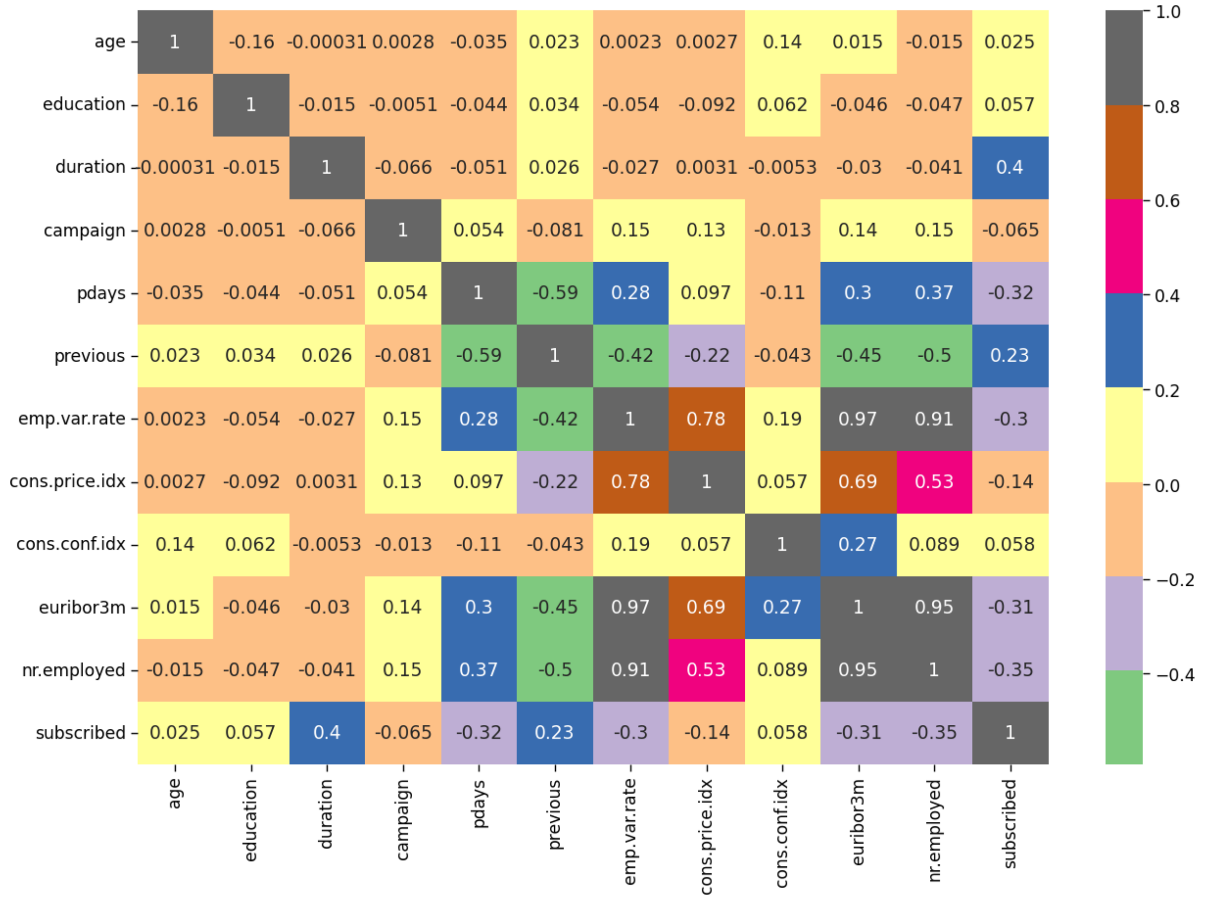


Figure 6. Heatmap of the dataset correlation columns

```
bank_corr['subscribed'].sort_values(ascending=True)
```

```
nr.employed            -0.351258
pdays                  -0.320820
euribor3m              -0.306341
emp.var.rate           -0.300278
poutcome_nonexistent   -0.193821
cons.price.idx         -0.143725
contact_telephone      -0.142108
month_may              -0.107764
job_blue-collar        -0.075260
campaign               -0.065436
marital_married        -0.043691
month_jul              -0.033678
job_services           -0.029101
day_of_week_mon        -0.024767
job_entrepreneur       -0.016961
month_aug              -0.016487
marital_divorced       -0.016435
job_housemaid          -0.010019
month_nov              -0.008758
day_of_week_fri        -0.006262
housing_no             -0.006018
job_technician         -0.004787
job_self-employed      -0.003484
month_jun              -0.003293
loan_yes               -0.002567
day_of_week_wed         0.000191
job_management          0.001061
loan_no                 0.002567
housing_yes             0.006018
day_of_week_tue         0.011728
job_unemployed          0.018921
day_of_week_thu         0.018995
```

```
age                    0.024988
job_admin.             0.030362
poutcome_failure       0.031838
education              0.057166
cons.conf.idx          0.057937
marital_single         0.059071
month_apr              0.075074
job_retired            0.082011
month_dec              0.087179
job_student            0.095646
month_sep              0.124644
month_oct              0.138406
contact_cellular       0.142108
month_mar              0.146155
previous               0.229887
poutcome_success       0.313669
duration               0.399974
subscribed             1.000000
Name: subscribed, dtype: float64
```

Figure 7. Correlation of Attribute to Target Attribute in Ascending Order

Based on Figure 7, we can see that the attribute that has a strong relationship with subscribed attribute are duration, poutcome_success, and nr.employed. But to investigate further, we can use feature importance from decision tree and other tree-based algorithm to verify this later in other section so that we can do feature selection. Moreover, later in other section, we will test how feature selection affect the results of our prediction. But before we build the model, the step we need to do is to separate input attribute and target class from the datasets using this python code:

```
X = bankData.iloc[:, :-1]
y = bankData.iloc[:,-1]
```

After we have separated the input attribute and the target class, we then need to normalize the input attribute because some machine learning tends to capture features that has high range of value to be more important. However, there are also some machine learning algorithms that do not need any normalization. So later in the section of building models, there will not be any normalization techniques done in some of the models. Furthermore, there are several normalization techniques such as Min-Max normalization and Z-score distribution which will be used depend on the model we build as some models tend to work better with certain normalization technique. In order to do normalization in python we can use both of this line of code:

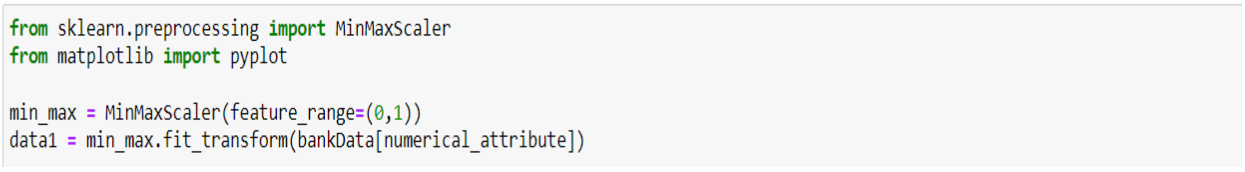

```
from sklearn.preprocessing import MinMaxScaler
from matplotlib import pyplot

min_max = MinMaxScaler(feature_range=(0,1))
data1 = min_max.fit_transform(bankData[numerical_attribute])
```

Figure 8. Min-Max Normalization

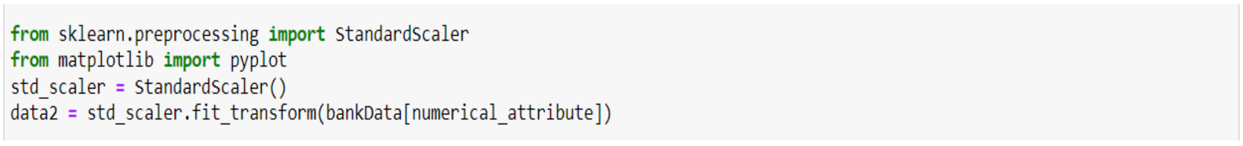
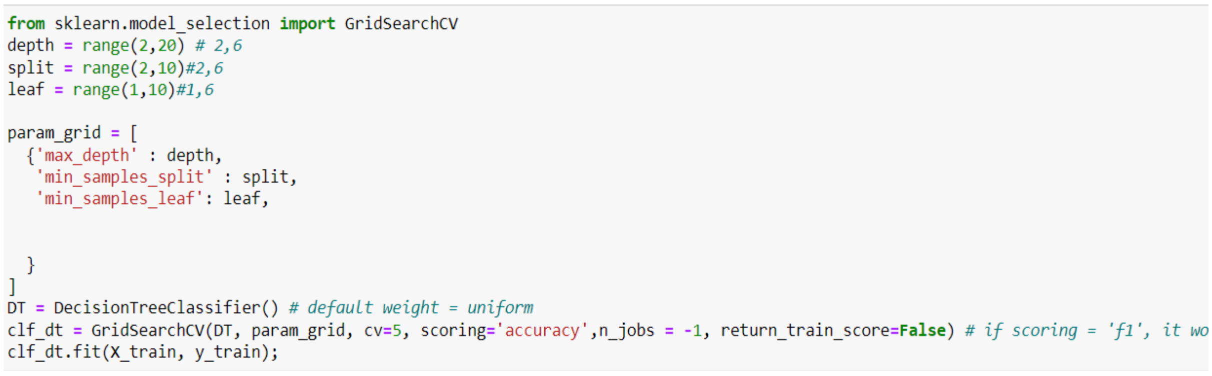

```
from sklearn.preprocessing import StandardScaler
from matplotlib import pyplot
std_scaler = StandardScaler()
data2 = std_scaler.fit_transform(bankData[numerical_attribute])
```

Figure 9. Z-Score Normalization

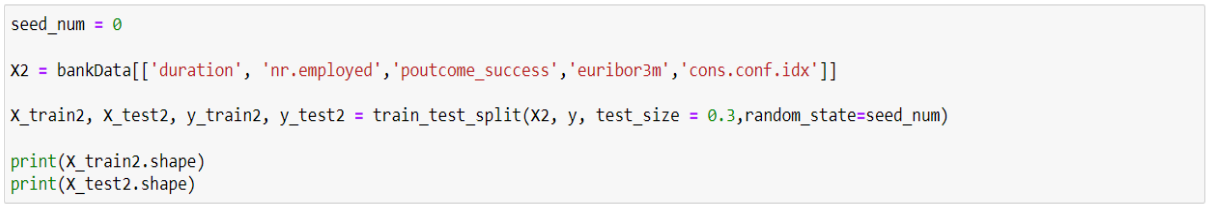
Steps to Solving The Problem:

In solving the problem, various methods are implemented and tested to see whether the model performance improve when we use the method. Each algorithm we build are enhanced with GridSearch (Parameter Optimization) from scikit-learn library in python so that we can tune the parameter to give the optimal result. The parameter we tune for each algorithm are different depending on the model itself. Regarding feature selection, it will be applied and shown in the building model section so that we can experiment whether it actually reduce overfitting and thus increasing the model performance. When we got into to the experiment, we will also be able to see the trade off between time complexity and performance. Therefore, feature selection will be discussed more in later section. Besides feature selection, some models also perform better when the dataset is normalized first. So, depending on the model, normalization will be applied. Another method that we will used to solve the problem is called SMOTE. SMOTE is used to solve the problem of unbalance class where there is a majority and minority class. When we have this kind of problem, the model we built would have been more biased towards the majority class and we do not want this in our model as it may affect our model. So later, we will investigate further on whether SMOTE method will improve or deteriorate our model. Finally, about partitioning strategy, because in python we already use GridSearch for parameter optimization, we do not need to split our data to training set, test set, and validation set. Instead, we split our data into training set and test set because GridSearch function will use training set and divide it into several fold which is similar to validation set. Finally, after we have all the models ready, we will then use cross-validation method in python called cross_validate from scikit-learn library. This method is used to validate our model and for performance comparison between each model.



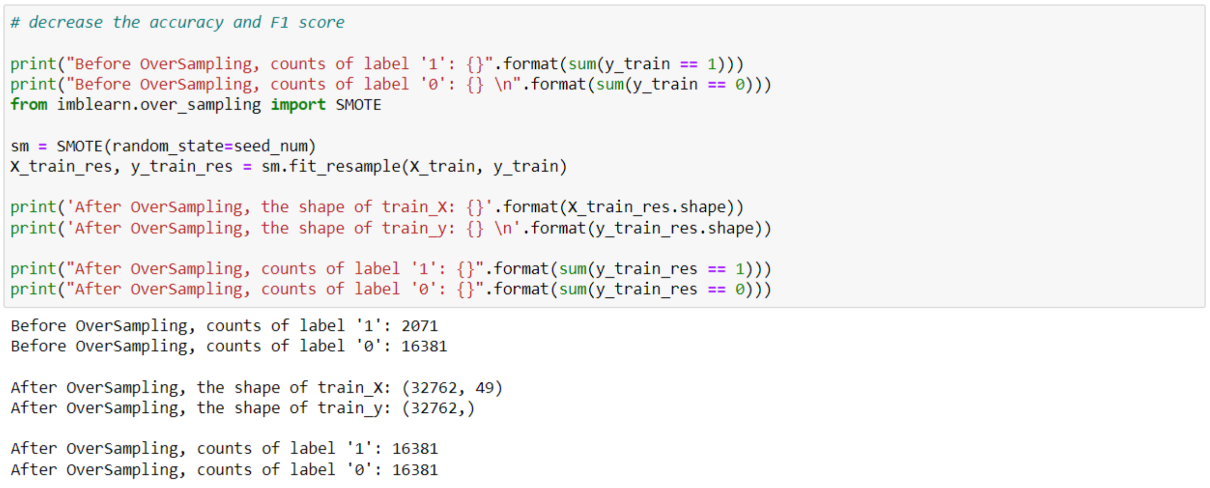
```
from sklearn.model_selection import GridSearchCV
depth = range(2,20) # 2,6
split = range(2,10)#2,6
leaf = range(1,10)#1,6

param_grid = [
  {'max_depth' : depth,
   'min_samples_split' : split,
   'min_samples_leaf': leaf,


  }
]
DT = DecisionTreeClassifier() # default weight = uniform
clf_dt = GridSearchCV(DT, param_grid, cv=5, scoring='accuracy',n_jobs = -1, return_train_score=False) # if scoring = 'f1', it wo
clf_dt.fit(X_train, y_train);
```

Figure 10. GridSearch Parameter Optimization on one of the models (Decision Tree)



```
seed_num = 0

X2 = bankData[['duration', 'nr.employed','poutcome_success','euribor3m','cons.conf.idx']]

X_train2, X_test2, y_train2, y_test2 = train_test_split(X2, y, test_size = 0.3,random_state=seed_num)

print(X_train2.shape)
print(X_test2.shape)
```

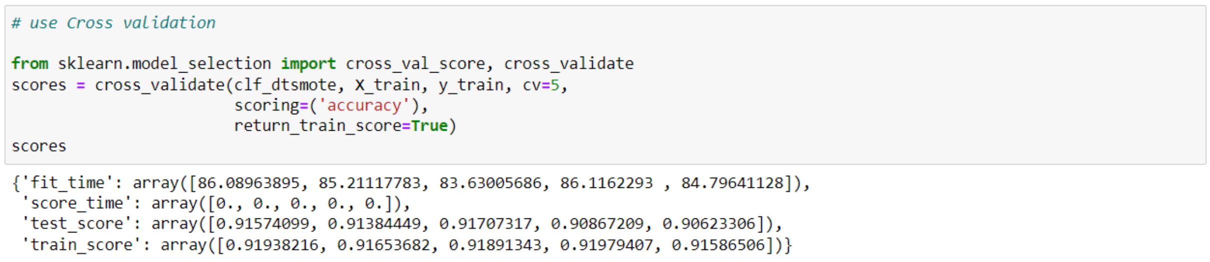
Figure 11. Feature Selection on one of the models (Decision Tree)



```
# decrease the accuracy and F1 score

print("Before OverSampling, counts of label '1': {}".format(sum(y_train == 1)))
print("Before OverSampling, counts of label '0': {} \n".format(sum(y_train == 0)))
from imblearn.over_sampling import SMOTE

sm = SMOTE(random_state=seed_num)
X_train_res, y_train_res = sm.fit_resample(X_train, y_train)

print('After OverSampling, the shape of train_X: {}'.format(X_train_res.shape))
print('After OverSampling, the shape of train_y: {} \n'.format(y_train_res.shape))

print("After OverSampling, counts of label '1': {}".format(sum(y_train_res == 1)))
print("After OverSampling, counts of label '0': {}".format(sum(y_train_res == 0)))
```

```
Before OverSampling, counts of label '1': 2071
Before OverSampling, counts of label '0': 16381

After OverSampling, the shape of train_X: (32762, 49)
After OverSampling, the shape of train_y: (32762,)

After OverSampling, counts of label '1': 16381
After OverSampling, counts of label '0': 16381
```

Figure 12. SMOTE in Python



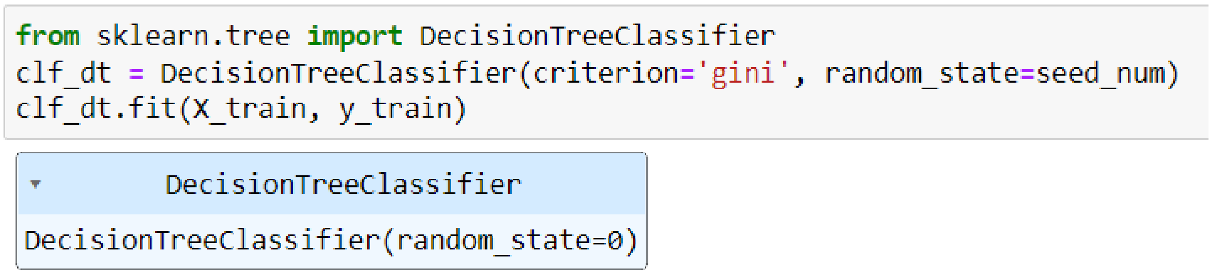
```
# use Cross validation

from sklearn.model_selection import cross_val_score, cross_validate
scores = cross_validate(clf_dtsmote, X_train, y_train, cv=5,
                        scoring=('accuracy'),
                        return_train_score=True)
scores
```

```
{'fit_time': array([86.08963895, 85.21117783, 83.63005686, 86.1162293 , 84.79641128]),
 'score_time': array([0., 0., 0., 0., 0.]),
 'test_score': array([0.91574099, 0.91384449, 0.91707317, 0.90867209, 0.90623306]),
 'train_score': array([0.91938216, 0.91653682, 0.91891343, 0.91979407, 0.91586506])}
```

Figure 13, Cross-Validation in Python

Decision Tree:

Decision Tree in python is built using a library from scikit-learn. To develop a Decision tree model in python will only need these few lines of code:

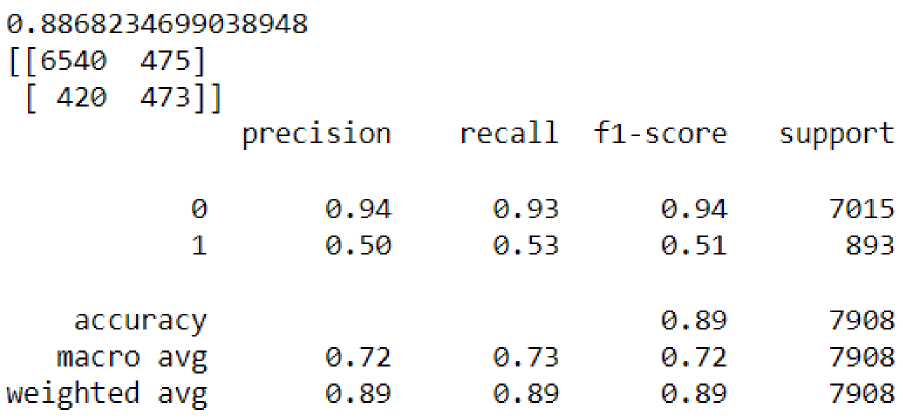


```
from sklearn.tree import DecisionTreeClassifier
clf_dt = DecisionTreeClassifier(criterion='gini', random_state=seed_num)
clf_dt.fit(X_train, y_train)
```

```
▾ DecisionTreeClassifier
DecisionTreeClassifier(random_state=0)
```

Figure 14. Decision Tree

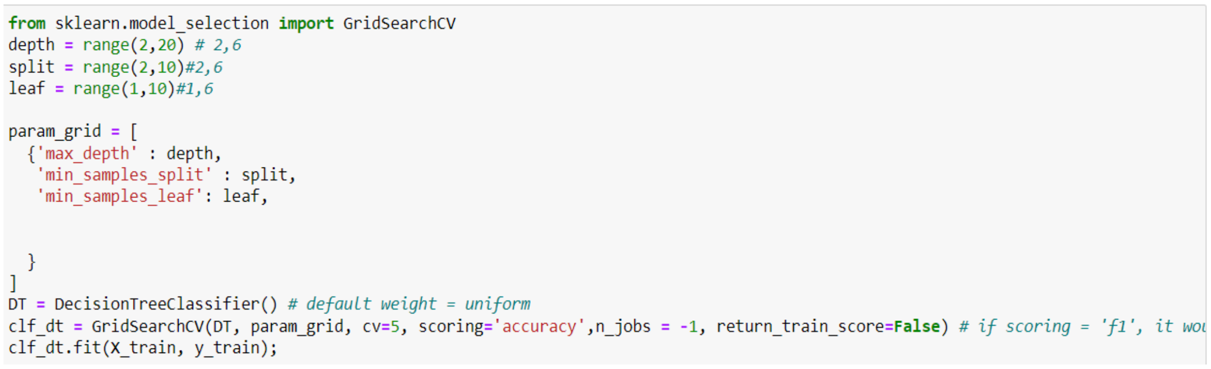
However, when we build a decision tree or any other model without any parameter optimization, the performance would not be optimal. The result on the test set is shown below.



```
0.8868234699038948
[[6540  475]
 [ 420  473]]
              precision    recall  f1-score   support

           0       0.94      0.93      0.94      7015
           1       0.50      0.53      0.51       893

    accuracy                           0.89      7908
   macro avg       0.72      0.73      0.72      7908
weighted avg       0.89      0.89      0.89      7908
```

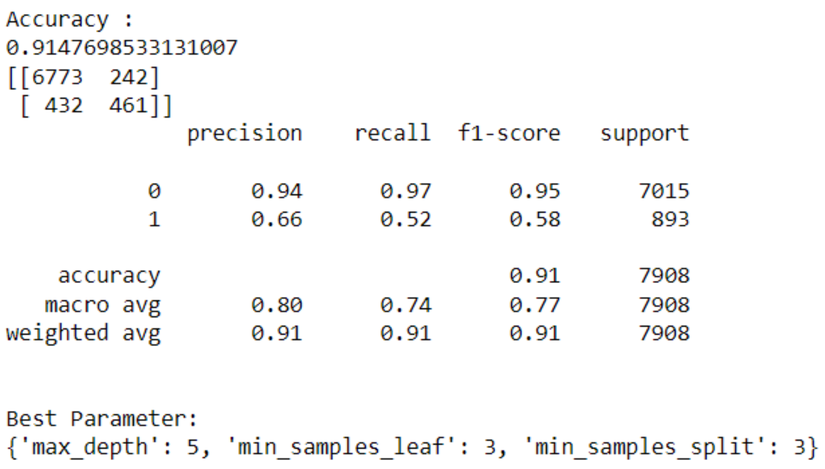
Figure 15. Decision Tree (without any parameter tuning) Performance on Test set

This result can be improved further when we use a parameter optimization technique in python called GridSearch. GridSearch will help us in finding the optimal parameter for the model by dividing the training set into several folds of training set and validation set.



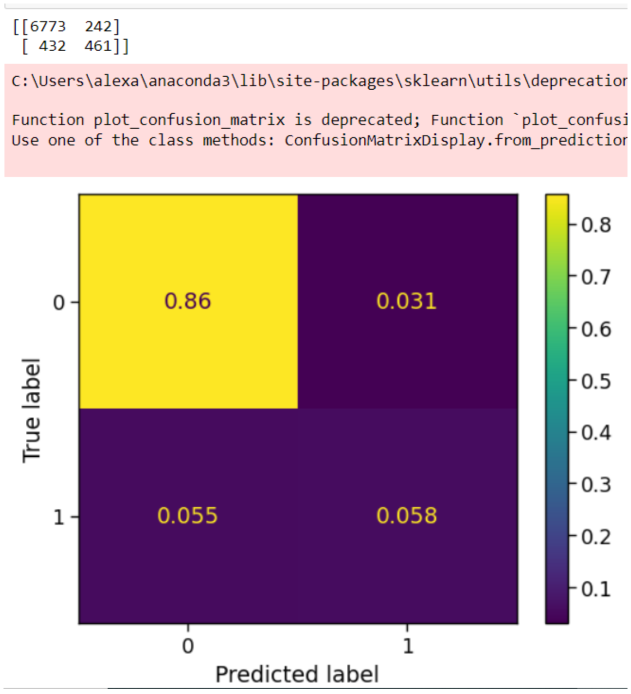
```
from sklearn.model_selection import GridSearchCV
depth = range(2,20) # 2,6
split = range(2,10)#2,6
leaf = range(1,10)#1,6

param_grid = [
  {'max_depth' : depth,
   'min_samples_split' : split,
   'min_samples_leaf': leaf,


  }
]
DT = DecisionTreeClassifier() # default weight = uniform
clf_dt = GridSearchCV(DT, param_grid, cv=5, scoring='accuracy',n_jobs = -1, return_train_score=False) # if scoring = 'f1', it wou
clf_dt.fit(X_train, y_train);
```

Figure 16. GridSearch on Decision Tree



```
Accuracy :
0.9147698533131007
[[6773  242]
 [ 432  461]]
              precision    recall  f1-score   support

           0       0.94      0.97      0.95      7015
           1       0.66      0.52      0.58       893

    accuracy                           0.91      7908
   macro avg       0.80      0.74      0.77      7908
weighted avg       0.91      0.91      0.91      7908


Best Parameter:
{'max_depth': 5, 'min_samples_leaf': 3, 'min_samples_split': 3}
```

Figure 17. Decision Tree (Parameter tuning) Performance



```
[[6773  242]
 [ 432  461]]
```

```
C:\Users\alexa\anaconda3\lib\site-packages\sklearn\utils\deprecation

Function plot_confusion_matrix is deprecated; Function `plot_confusi
Use one of the class methods: ConfusionMatrixDisplay.from_prediction
```

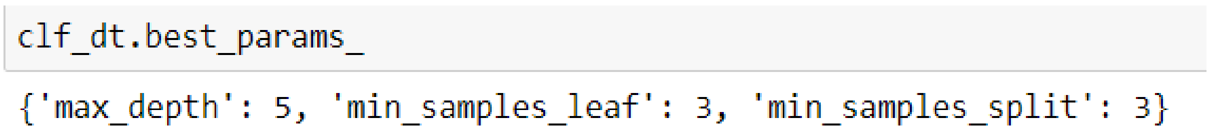
Figure 18. Confusion Matrix

The parameter that we will optimize in Decision Tree are:

- max_depth: the maximum depth of the tree
- min_samples_split: The minimum number of samples required to split an internal node
- min_samples_leaf: The minimum number of samples required to be at a leaf node

The value of parameter that we will experiment:

- max_depth: 2 to 19
- min_samples_split: 2 to 9
- min_samples_leaf: 2 to 9



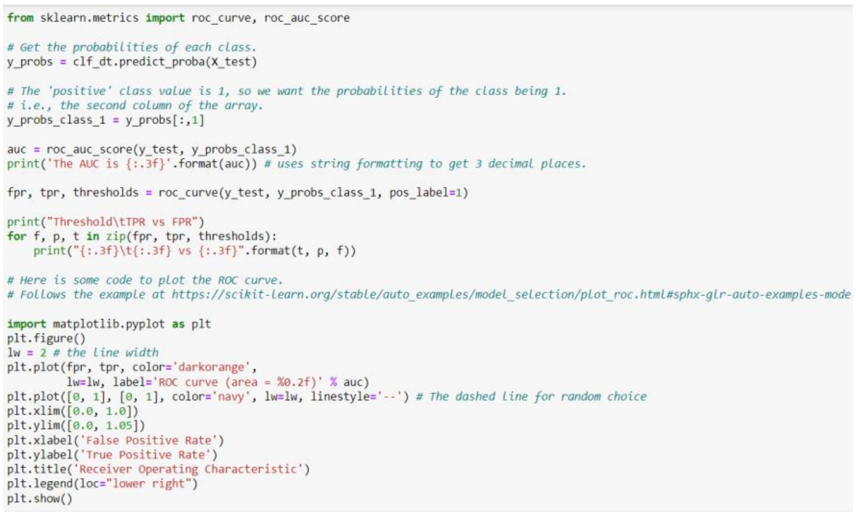
```
clf_dt.best_params_
```

```
{'max_depth': 5, 'min_samples_leaf': 3, 'min_samples_split': 3}
```

Figure 19. Best Parameter

Based on Figure 17, we can see that by applying GridSearch method, our model accuracy improves from 88.68% to 91.47%. In addition, when we have an unbalance dataset, the metrics of performance that we want to use is F1-score instead of accuracy for evaluating the model. As seen in the diagram above, the F1 score of the minority class increase from 0.51 to 0.58 and the F1-score of the majority class also has a slight increase from 0.94 to 0.95. Regarding the best parameter, it can also be found out using GridSearch method. Based on that, we can see that the best parameter is 5 for max_depth, min_samples_split is 3, and min_samples_leaf is 3.

However, to be more precise and detail, we will test our model using cross_validate method so that it can be compared with other decision tree experiment model as well as another algorithm. Another method that we will use to further evaluate our model is by using ROC curve. The ROC curve and AUC value for the tuned Decision Tree is shown in below diagram.



```python
from sklearn.metrics import roc_curve, roc_auc_score

# Get the probabilities of each class.
y_probs = clf_dt.predict_proba(X_test)

# The 'positive' class value is 1, so we want the probabilities of the class being 1.
# i.e., the second column of the array.
y_probs_class_1 = y_probs[:,1]

auc = roc_auc_score(y_test, y_probs_class_1)
print('The AUC is {:.3f}'.format(auc)) # uses string formatting to get 3 decimal places.

fpr, tpr, thresholds = roc_curve(y_test, y_probs_class_1, pos_label=1)

print("Threshold\tTPR vs FPR")
for f, p, t in zip(fpr, tpr, thresholds):
    print("{:.3f}\t{:.3f} vs {:.3f}".format(t, p, f))

# Here is some code to plot the ROC curve.
# Follows the example at https://scikit-learn.org/stable/auto_examples/model_selection/plot_roc.html#sphx-glr-auto-examples-mode

import matplotlib.pyplot as plt
plt.figure()
lw = 2 # the line width
plt.plot(fpr, tpr, color='darkorange',
         lw=lw, label='ROC curve (area = %0.2f)' % auc)
plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--') # The dashed line for random choice
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic')
plt.legend(loc="lower right")
plt.show()
```

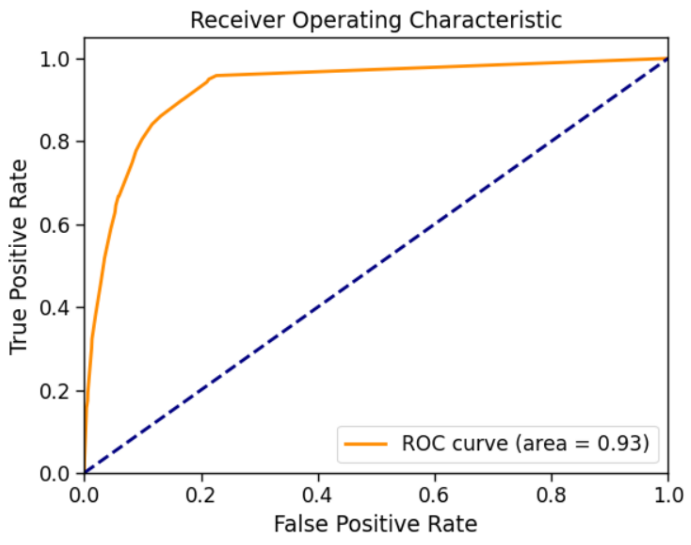


Figure 20. ROC curve for tuned Decision Tree

In diagram above (Figure 20), the AUC value is 0.93 which mean 93% predict 1 it is actually 1 or in layman's terms how often it can see a True Positive as True Positive. We are also able to visualize the tree from the model.

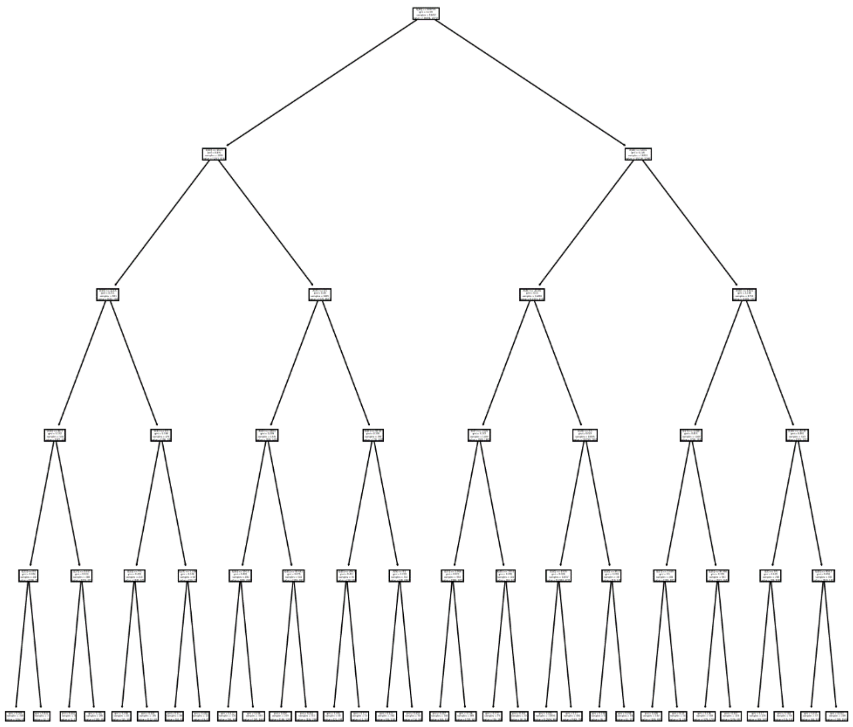
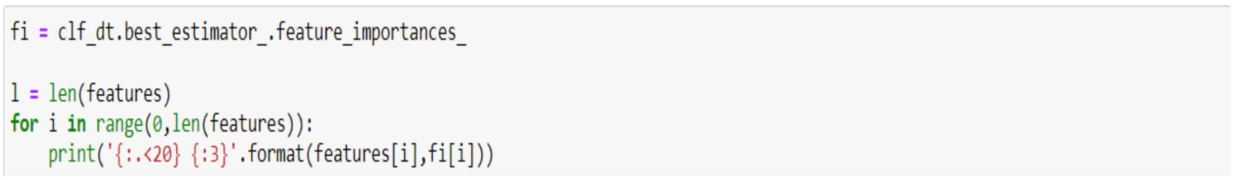


Figure 21. Tree Visualization

Another approach that we will try to combine with parameter optimization to improve our model performance is by using SMOTE and feature selection. For SMOTE, we can use the python code from figure 12. Whereas for feature selection, we can refer to figure 7 and then verify it again from a method called feature_importances_ that we gain from building the Decision Tree before. The result and the code are shown below:



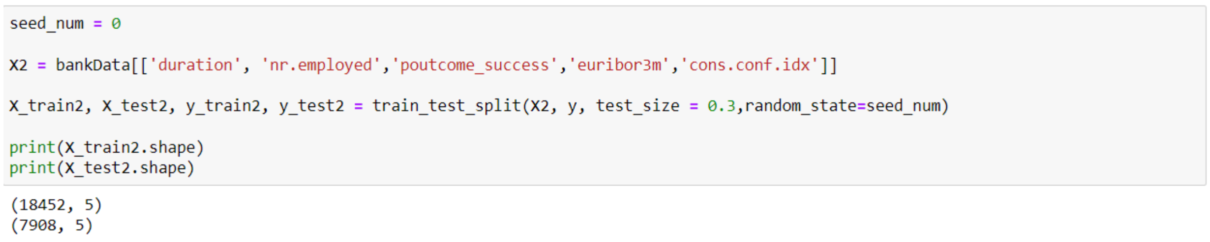
```
fi = clf_dt.best_estimator_.feature_importances_

l = len(features)
for i in range(0,len(features)):
    print('{:.<20} {:3}'.format(features[i],fi[i]))
```

```
job_admin........... 0.0016034642524747433
job_blue-collar..... 0.0
job_entrepreneur.... 0.0
job_housemaid....... 0.0
job_management...... 0.0
job_retired......... 0.0
job_self-employed... 0.0
job_services........ 0.0
job_student......... 0.0
job_technician...... 0.0012086084977587803
job_unemployed...... 0.0
```

```
marital_divorced.... 0.0
marital_married..... 0.0
marital_single...... 0.0
housing_no.......... 0.0
housing_yes......... 0.0
loan_no............. 0.0
loan_yes............ 0.0
contact_cellular.... 0.0
contact_telephone... 0.0
month_apr........... 0.0
month_aug........... 0.0
month_dec........... 0.00035418921079457524
month_jul........... 0.0
month_jun........... 0.0
month_mar........... 0.0
month_may........... 0.0
month_nov........... 0.0
month_oct........... 0.013926884812167284
month_sep........... 0.0
day_of_week_fri..... 0.0
day_of_week_mon..... 0.002403334387135196
day_of_week_thu..... 0.0
day_of_week_tue..... 0.0
day_of_week_wed..... 0.0
poutcome_failure.... 0.0
poutcome_nonexistent 0.0
poutcome_success.... 0.032830308131018233
age................. 0.0
education........... 0.0
duration............ 0.5270491753882955
campaign............ 0.0
pdays............... 0.003306758291704743
previous............ 0.0
emp.var.rate........ 0.0
cons.price.idx...... 0.008826330977243392
cons.conf.idx....... 0.030747730558177293
euribor............. 0.04181702521027498
nr.employed......... 0.3359261902829553
```

Figure 22. Decision Tree Feature Importance

Based on Figure 7 and Figure 22, we can then take several attributes which are Duration, nr.employed, poutcome_success, euribor3m, and cons.conf.idx that has the highest correlation to be selected for training the model. The line of python code:



```
seed_num = 0

X2 = bankData[['duration', 'nr.employed','poutcome_success','euribor3m','cons.conf.idx']]

X_train2, X_test2, y_train2, y_test2 = train_test_split(X2, y, test_size = 0.3,random_state=seed_num)

print(X_train2.shape)
print(X_test2.shape)
```

```
(18452, 5)
(7908, 5)
```

Figure 23. Decision Tree with Feature selection

Result table

Model	Accuracy	Recall class 0	Recall class 1	F1 class 0	F1 class 1	Parameter	Kaggle Score
Decision Tree (normal)	0.8868	0.93	0.53	0.94	0.51	`{'max_depth': none, 'min_samples_leaf': 1, 'min_samples_split': 2}`	0.89135
Decision Tree (Optimized)	0.91476	0.97	0.52	0.95	0.58	`{'max_depth': 5, 'min_samples_leaf': 3, 'min_samples_split': 3}`	0.92261
Decision Tree (Optimized SMOTE)	0.8837	0.91	0.70	0.93	0.58	`{'max_depth': 9, 'min_samples_leaf': 3, 'min_samples_split': 6}`	0.88315
Decision Tree (Optimized Feature Selection)	0.9135	0.97	0.50	0.95	0.57	`{'max_depth': 5, 'min_samples_leaf': 5, 'min_samples_split': 2}`	0.92169

Hence, by seeing the result in the result table, we can then see what approach we should use in building Decision Tree model. Amongst all the Decision Tree version, the version which perform the best is the model with all features, No SMOTE, and has been optimized, gives the best result in predicting the unknown dataset and the test set in the dataset where feature and target class are given. It turns out using SMOTE actually decrease our performance both in test set and Kaggle score.

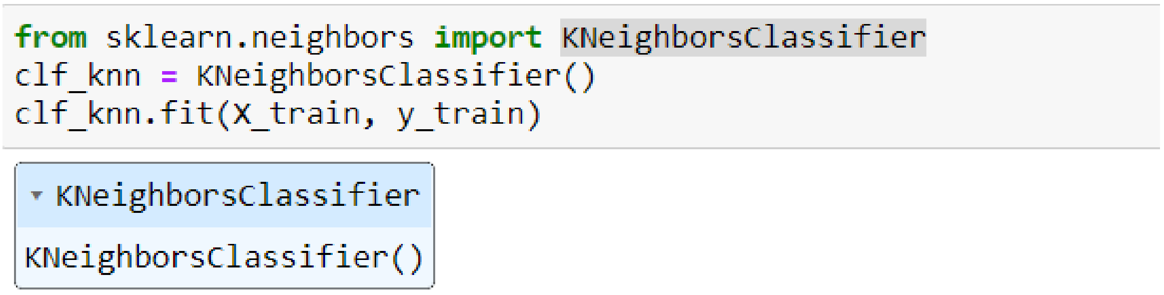
Time Complexity vs Performance Table

Model	Accuracy	F1 class 0	F1 class 1	Kaggle score	Avg Fit_time
Decision Tree (Optimized)	0.91476	0.95	0.58	0.92261	125.99
Decision Tree (Optimized Feature selection)	0.9135	0.95	0.57	0.92169	44.87

Although as seen in the table above that by using feature selection decrease the performance of the model, the time to fit and train the model is faster which is a trade-off. So, depending on the case, we might want to sacrifice performance over time complexity.

K-Nearest Neighbors:

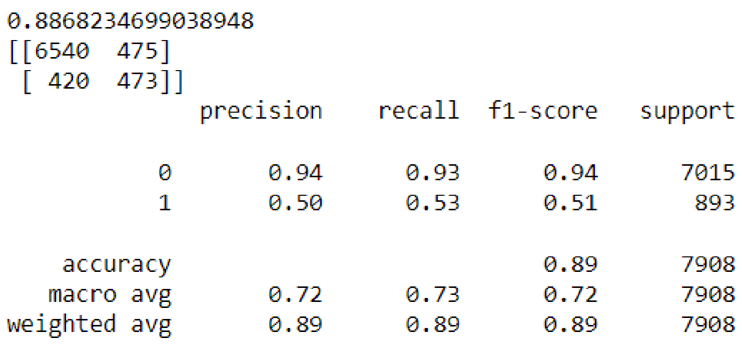
K-Nearest Neighbors in python is built using a library from scikit-learn called KNeighborsClassifier. To develop a KNN model in python will only need these few lines of code:



```python
from sklearn.neighbors import KNeighborsClassifier
clf_knn = KNeighborsClassifier()
clf_knn.fit(X_train, y_train)
```

```
▾ KNeighborsClassifier
KNeighborsClassifier()
```

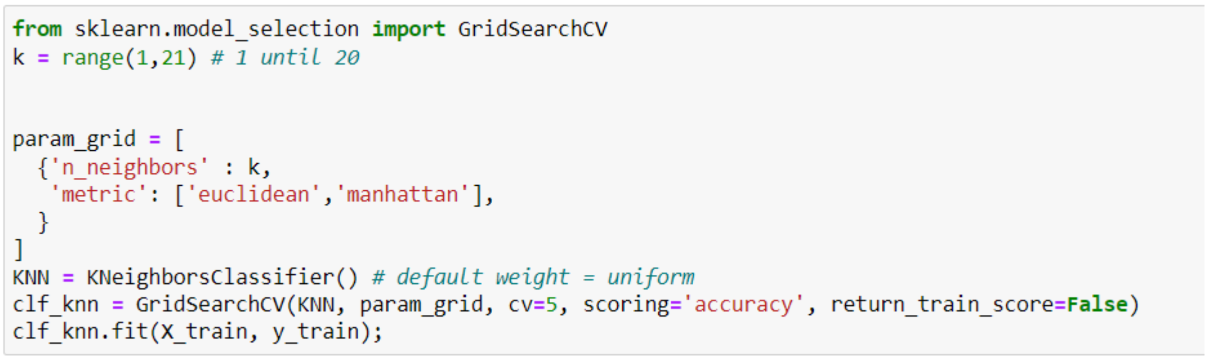
Figure 24. KNN algorithm



```
0.8868234699038948
[[6540  475]
 [ 420  473]]
              precision    recall  f1-score   support

           0       0.94      0.93      0.94      7015
           1       0.50      0.53      0.51       893

    accuracy                           0.89      7908
   macro avg       0.72      0.73      0.72      7908
weighted avg       0.89      0.89      0.89      7908
```

Figure 25. KNN Performance (Without any parameter optimization

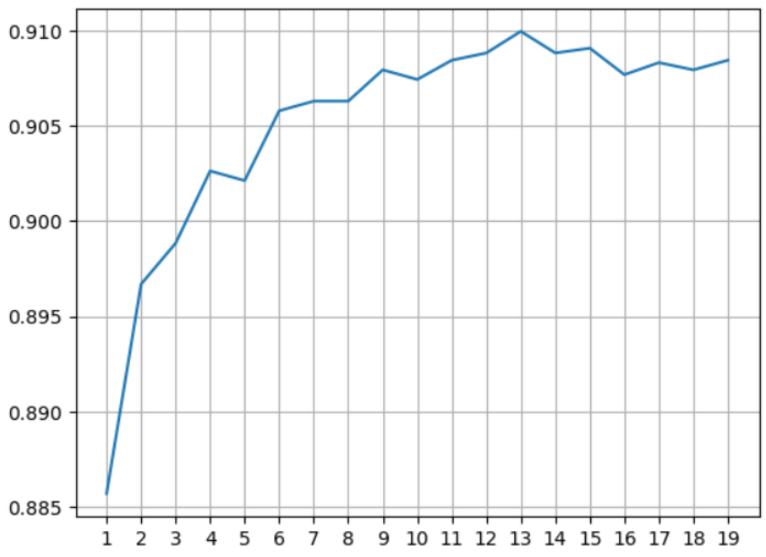
The accuracy can be further improved when we use GridSearch. Doing GridSearch is the same for every other algorithm, the only difference lies in the list of parameters that we want to optimize.

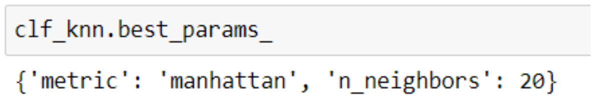


```python
from sklearn.model_selection import GridSearchCV
k = range(1,21) # 1 until 20


param_grid = [
  {'n_neighbors' : k,
   'metric': ['euclidean','manhattan'],
  }
]
KNN = KNeighborsClassifier() # default weight = uniform
clf_knn = GridSearchCV(KNN, param_grid, cv=5, scoring='accuracy', return_train_score=False)
clf_knn.fit(X_train, y_train);
```

Figure 26. GridSearch on KNN classifier

In addition, we are also able to plot a graph that illustrate a visualization of the accuracy when value of K increase.



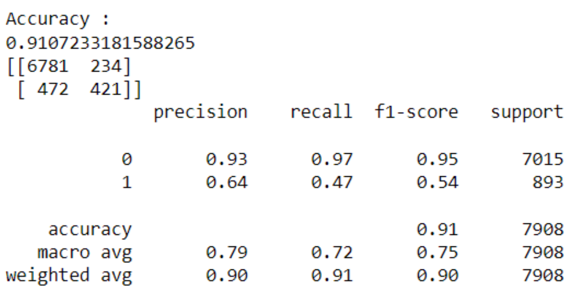


```
clf_knn.best_params_
```

```
{'metric': 'manhattan', 'n_neighbors': 20}
```

Figure 27. Graph of accuracy vs K value and best parameter

Based on Figure 27, we can infer that increase K until certain point would increase the performance of a model. But we also have to put something in mind, when increasing the performance of the model there's a trade-off which is time. The KNN model we develop using GridSearch parameter optimization would give us this result:



```
Accuracy :
0.9107233181588265
[[6781  234]
 [ 472  421]]
              precision    recall  f1-score   support

           0       0.93      0.97      0.95      7015
           1       0.64      0.47      0.54       893

    accuracy                           0.91      7908
   macro avg       0.79      0.72      0.75      7908
weighted avg       0.90      0.91      0.90      7908
```

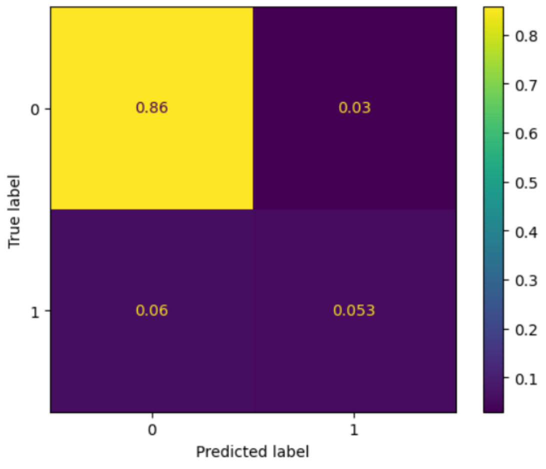


Figure 28. KNN (optimized) Performance

Figure 29. KNN Confusion Matrix

For further analysis on whether the type of metrics (parameter) should we opt for in this given problem, we can visualize the comparison using a python code shown below

```
score = clf_knn.cv_results_['mean_test_score']

plt.plot(np.arange(1,20), score[0:19])
plt.xticks(np.arange(1,20))
plt.grid()
plt.show
plt.xlabel('Value of K (Euclidean)')
plt.ylabel('Cross-Validated Accuracy')
print(max(score))
```

```
0.9129632542058286
```

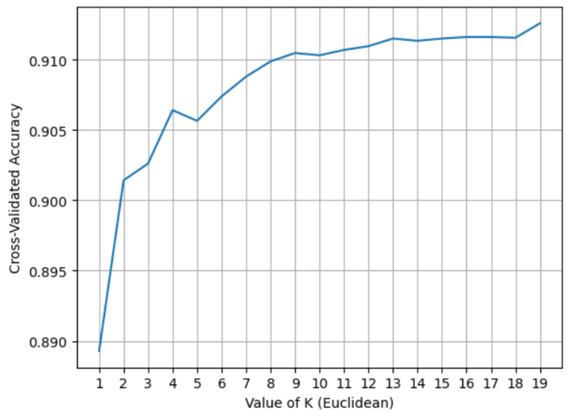
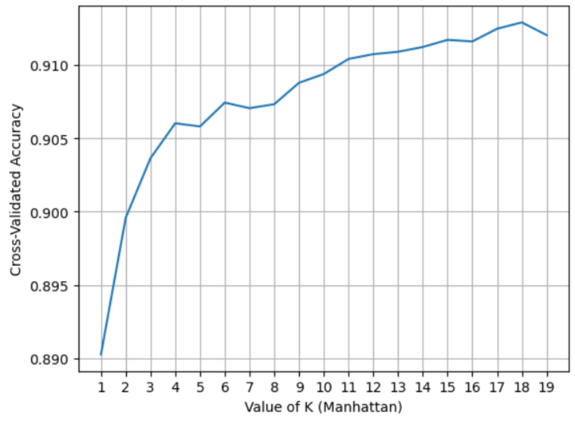


Figure 30. Accuracy when using Euclidean

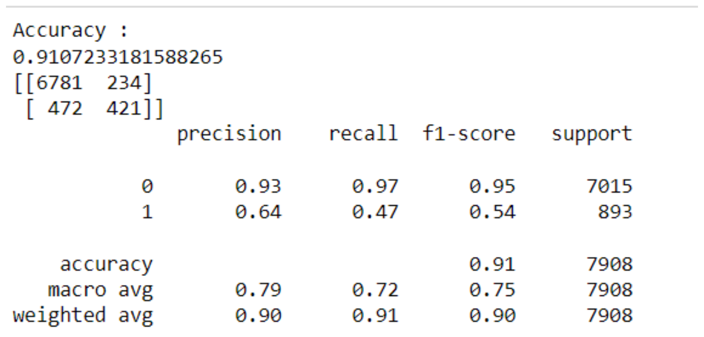
```
score = clf_knn.cv_results_['mean_test_score']

plt.plot(np.arange(1,20), score[20:39])
plt.xticks(np.arange(1,20))
plt.xlabel('Value of K (Manhattan)')
plt.ylabel('Cross-Validated Accuracy')
plt.grid()
plt.show()
print(max(score))
```



```
0.9129632542058286
```

Figure 31. Accuracy when using Manhattan

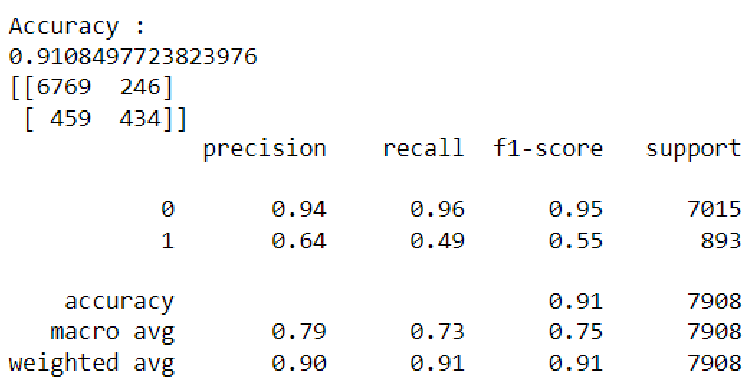
Another approach that we will experiment on to increase the performance is using Normalization. Normalization as mentioned in section 2 (Data Pre-processing) are required only for some algorithm. KNN needs normalization because it is an algorithm based on distance. So, the data will be normalized and then the model will be tuned with the normalized data. However, the result shown to be the same.



```
Accuracy :
0.9107233181588265
[[6781  234]
 [ 472  421]]
              precision    recall  f1-score   support

           0       0.93      0.97      0.95      7015
           1       0.64      0.47      0.54       893

    accuracy                           0.91      7908
   macro avg       0.79      0.72      0.75      7908
weighted avg       0.90      0.91      0.90      7908
```

Figure 32. KNN Normalization result

Another method that we can try on is using Feature Selection because the algorithm might have taken all things into consideration and thus causing it to be overfitted. The result of normalize + Feature selection is shown below.

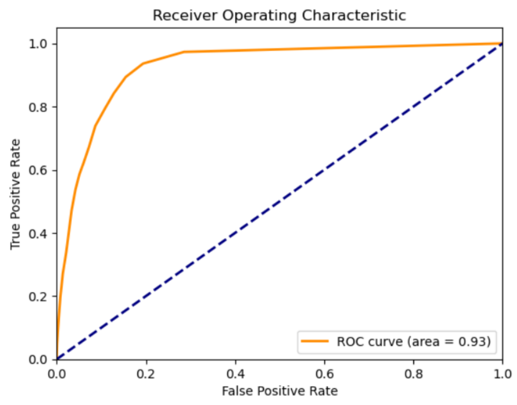


```
Accuracy :
0.9108497723823976
[[6769  246]
 [ 459  434]]
              precision    recall  f1-score   support

           0       0.94      0.96      0.95      7015
           1       0.64      0.49      0.55       893

    accuracy                           0.91      7908
   macro avg       0.79      0.73      0.75      7908
weighted avg       0.90      0.91      0.91      7908
```

Figure 33. Normalize + FS KNN Confusion Matrix

*FS = Feature Selection

Moreover, the ROC of simple KNN without any normalization and feature selection can be compared with the KNN with Normalization and Feature selection. The AUC value for Normalize and Feature selection KNN is higher by 0.01 than the simple KNN. The closer the AUC value to 1, the better the model in predicting.



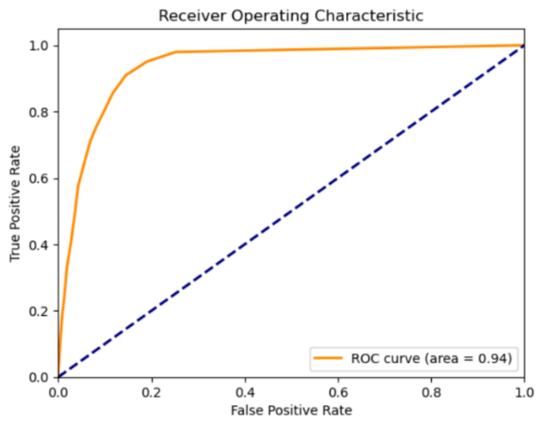


Figure 34. ROC (optimized KNN)

Figure 35. ROC (Normalize + FS KNN)

To summarize all the experiment, the table of result and time complexity is shown in below diagrams.

Result table

Model	Accuracy	Recall class 0	Recall class 1	F1 class 0	F1 class 1	Parameter	Kaggle Score
KNN (normal)	0.9021	0.96	0.47	0.95	0.52	`{'metric': 'minkowski', 'n_neighbors': 5}`	0.90015
KNN (Optimized)	0.9107	0.97	0.47	0.95	0.54	`{'metric': 'manhattan', 'n_neighbors': 20}`	0.91107

KNN (Optimized Normalize FS)	0.9108	0.96	0.49	0.95	0.55	{'metric': 'manhattan', 'n_neighbors': 18}	0.90288

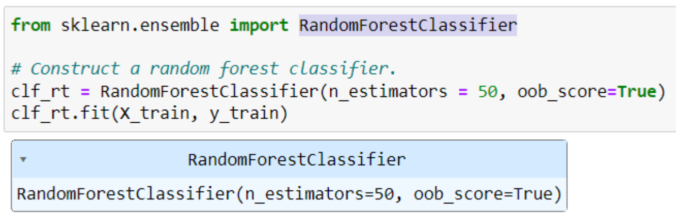
Although when optimized KNN (with normalization and Feature selection) are tested better in our first dataset, when the model is deployed into the second dataset (unknown) the model performs worse than the KNN without any normalization and Feature Selection.

Time Complexity vs Performance Table

Model	Accuracy	F1 class 0	F1 class 1	Kaggle score	Avg Fit_time
KNN (Optimized)	0.9107	0.95	0.54	0.91107	81.06
KNN (Normalized + FS)	0.9108	0.95	0.55	0.90288	19.67

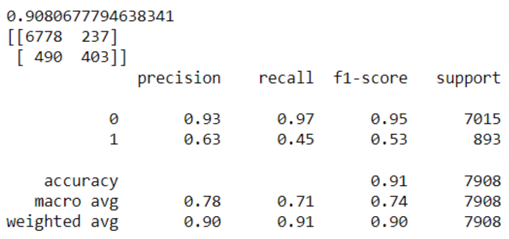
Random Forest:

Random Forest in python is built using a library from scikit-learn called RandomForestClassifier. To develop a RF model in python will only need these few lines of code:



```
from sklearn.ensemble import RandomForestClassifier

# Construct a random forest classifier.
clf_rt = RandomForestClassifier(n_estimators = 50, oob_score=True)
clf_rt.fit(X_train, y_train)
```

```
RandomForestClassifier
RandomForestClassifier(n_estimators=50, oob_score=True)
```

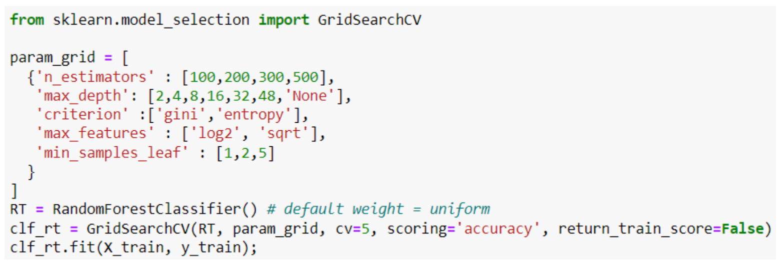


```
0.9080677794638341
[[6778  237]
 [ 490  403]]
              precision    recall  f1-score   support

           0       0.93      0.97      0.95      7015
           1       0.63      0.45      0.53       893

    accuracy                           0.91      7908
   macro avg       0.78      0.71      0.74      7908
weighted avg       0.90      0.91      0.90      7908
```

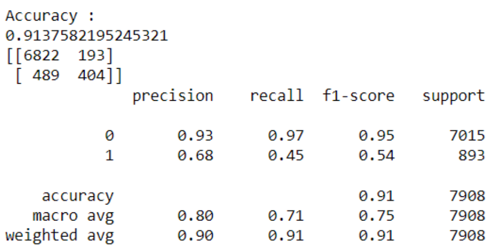
Figure 36. Python code Random Forest

Figure 37. Confusion Matrix

The accuracy can be further improved when we use GridSearch. Doing GridSearch is the same for every other algorithm, the only difference lies in the list of parameters that we want to optimize.



```
from sklearn.model_selection import GridSearchCV

param_grid = [
  {'n_estimators' : [100,200,300,500],
   'max_depth': [2,4,8,16,32,48,'None'],
   'criterion' :['gini','entropy'],
   'max_features' : ['log2', 'sqrt'],
   'min_samples_leaf' : [1,2,5]
  }
]
RT = RandomForestClassifier() # default weight = uniform
clf_rt = GridSearchCV(RT, param_grid, cv=5, scoring='accuracy', return_train_score=False)
clf_rt.fit(X_train, y_train);
```

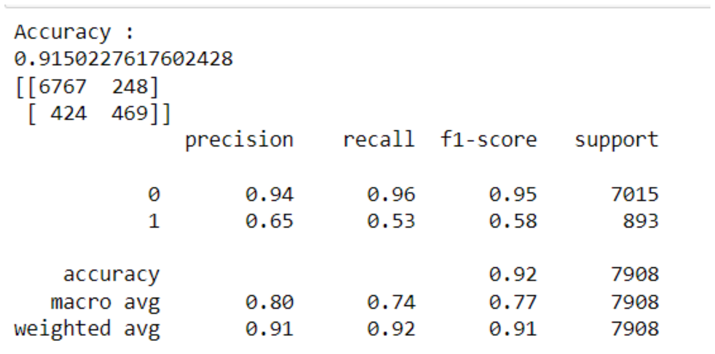


```
Accuracy :
0.9137582195245321
[[6822  193]
 [ 489  404]]
              precision    recall  f1-score   support

           0       0.93      0.97      0.95      7015
           1       0.68      0.45      0.54       893

    accuracy                           0.91      7908
   macro avg       0.80      0.71      0.75      7908
weighted avg       0.90      0.91      0.91      7908
```

Figure 38. GridSearch on Random Forest

Figure 39. Optimized RF Performance

The next approach we will try for Random Forest classifier are Feature Selection. The result is shown below.



```
Accuracy :
0.9150227617602428
[[6767  248]
 [ 424  469]]
              precision    recall  f1-score   support

           0       0.94      0.96      0.95      7015
           1       0.65      0.53      0.58       893

    accuracy                           0.92      7908
   macro avg       0.80      0.74      0.77      7908
weighted avg       0.91      0.92      0.91      7908
```

Figure 40. Random Forest FS Confusion matrix

Another performance measure we can test is using ROC curve. We will compare the optimized Random Forest with the Optimized + Feature Selection Random Forest.

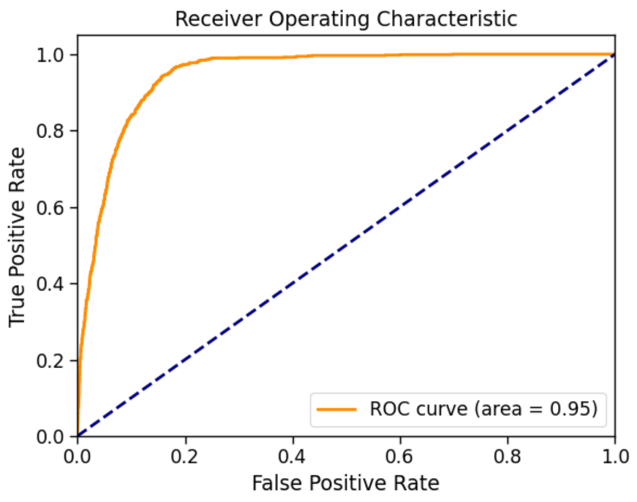


Figure 41. ROC Optimized RF

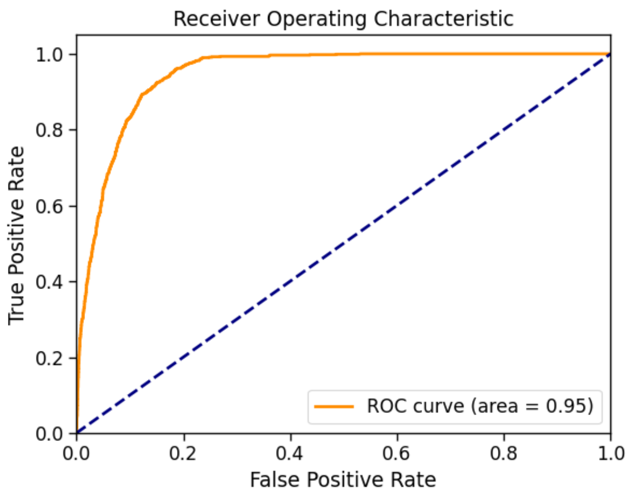


Figure 42. ROC Optimized FS RF

Result Table

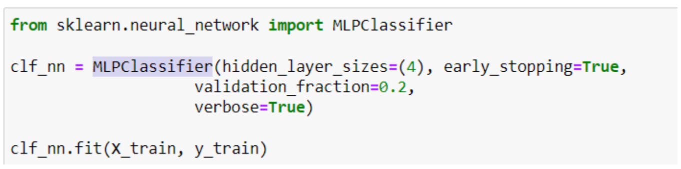
Model	Accuracy	Recall class 0	Recall class 1	F1 class 0	F1 class 1	Parameter	Kaggle Score
Random Forest (normal)	0.91135	0.97	0.47	0.95	0.54	`{'criterion': 'gini', 'max_depth': none, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'n_estimators': 100}`	Not tested

Random Forest (Optimized)	0.91236	0.97	0.46	0.95	0.54	{'criterion': 'gini', 'max_depth': 48, 'max_features': 'sqrt', 'min_samples_leaf': 2, 'n_estimators': 500}	0.92139
Random Forest (Optimized FS)	0.91426	0.96	0.52	0.95	0.58	{'criterion': 'gini', 'max_depth': 8, 'max_features': 'sqrt', 'min_samples_leaf': 5, 'n_estimators': 200}	0.91836

For Random Forest, the cross_validate method was not applied here due to the time it takes to run the method is more than an hour. However, the performance can already be evaluated based on the Kaggle score, but for the time complexity, based on the previous model such as Decision Tree and KNN, we know that using feature selection will decrease the time it takes to run the model.

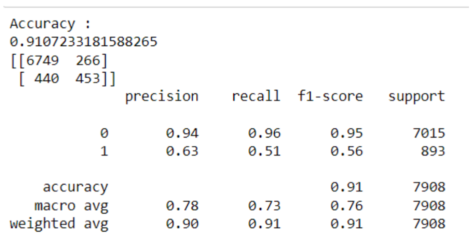
Neural Network:

Neural Network in python is built using a library from scikit-learn called MLPClassifier. To develop a NN model in python will only need these few lines of code:



```
from sklearn.neural_network import MLPClassifier

clf_nn = MLPClassifier(hidden_layer_sizes=(4), early_stopping=True,
                       validation_fraction=0.2,
                       verbose=True)

clf_nn.fit(X_train, y_train)
```

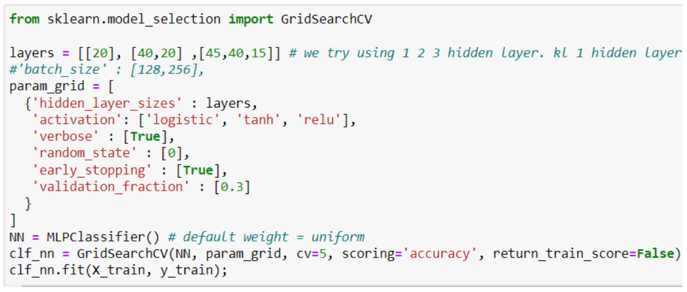
Figure 43. Python code NN



```
Accuracy :
0.9107233181588265
[[6749  266]
 [ 440  453]]
              precision    recall  f1-score   support

           0       0.94      0.96      0.95      7015
           1       0.63      0.51      0.56       893

    accuracy                           0.91      7908
   macro avg       0.78      0.73      0.76      7908
weighted avg       0.90      0.91      0.91      7908
```

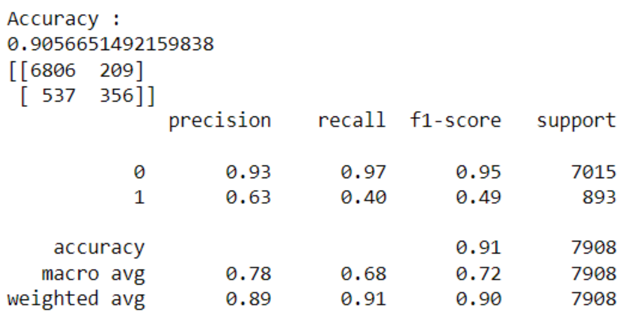
Figure 44. Confusion Matrix

The accuracy can be further improved when we use GridSearch. Doing GridSearch is the same for every other algorithm, the only difference lies in the list of parameters that we want to optimize.



```
from sklearn.model_selection import GridSearchCV

layers = [[20], [40,20] ,[45,40,15]] # we try using 1 2 3 hidden layer. kl 1 hidden layer
#'batch_size' : [128,256],
param_grid = [
  {'hidden_layer_sizes' : layers,
   'activation': ['logistic', 'tanh', 'relu'],
   'verbose' : [True],
   'random_state' : [0],
   'early_stopping' : [True],
   'validation_fraction' : [0.3]
  }
]
NN = MLPClassifier() # default weight = uniform
clf_nn = GridSearchCV(NN, param_grid, cv=5, scoring='accuracy', return_train_score=False)
clf_nn.fit(X_train, y_train);
```

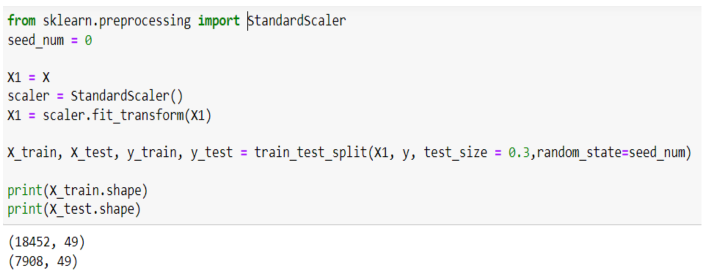
Figure 45. GridSearch on NN



```
Accuracy :
0.9056651492159838
[[6806  209]
 [ 537  356]]
              precision    recall  f1-score   support

           0       0.93      0.97      0.95      7015
           1       0.63      0.40      0.49       893

    accuracy                           0.91      7908
   macro avg       0.78      0.68      0.72      7908
weighted avg       0.89      0.91      0.90      7908
```

Figure 46. Optimized NN Performance

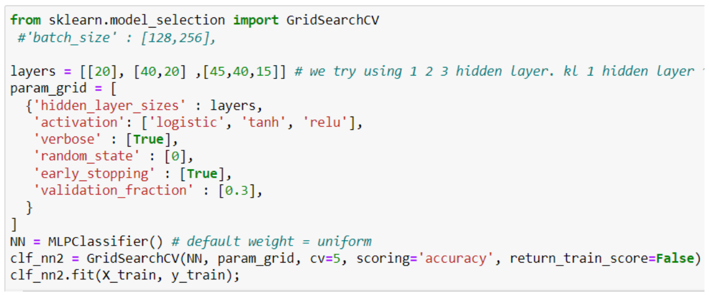
The next approach we will try for Neural Network is normalization. Neural Network need normalization because it helps to stabilize the gradient descent step, allowing us to use larger learning rates or help models converge faster for a given learning rate.



```
from sklearn.preprocessing import StandardScaler
seed_num = 0

X1 = X
scaler = StandardScaler()
X1 = scaler.fit_transform(X1)

X_train, X_test, y_train, y_test = train_test_split(X1, y, test_size = 0.3,random_state=seed_num)

print(X_train.shape)
print(X_test.shape)
```

```
(18452, 49)
(7908, 49)
```

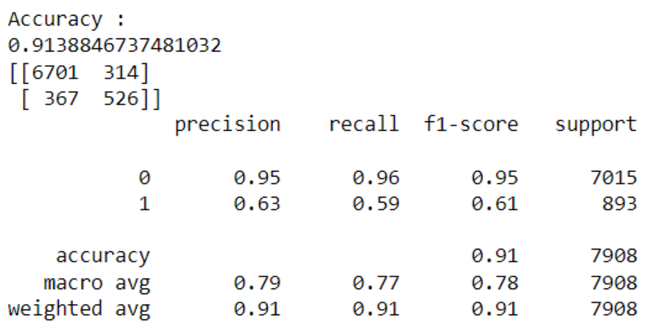
Figure 47. Standard Scaler



```
from sklearn.model_selection import GridSearchCV
 #'batch_size' : [128,256],

layers = [[20], [40,20] ,[45,40,15]] # we try using 1 2 3 hidden layer. kl 1 hidden layer
param_grid = [
  {'hidden_layer_sizes' : layers,
   'activation': ['logistic', 'tanh', 'relu'],
   'verbose' : [True],
   'random_state' : [0],
   'early_stopping' : [True],
   'validation_fraction' : [0.3],
  }
]
NN = MLPClassifier() # default weight = uniform
clf_nn2 = GridSearchCV(NN, param_grid, cv=5, scoring='accuracy', return_train_score=False)
clf_nn2.fit(X_train, y_train);
```

Figure 48. GridSearch on the normalized data

After we have normalized our data (Figure 47), we can then proceed to optimizing the parameter. This step is like the other steps we have done before with our previous model. The result of the Neural Network performance with data that has been normalized is shown below:



```
Accuracy :
0.9138846737481032
[[6701  314]
 [ 367  526]]
              precision    recall  f1-score   support

           0       0.95      0.96      0.95      7015
           1       0.63      0.59      0.61       893

    accuracy                           0.91      7908
   macro avg       0.79      0.77      0.78      7908
weighted avg       0.91      0.91      0.91      7908
```

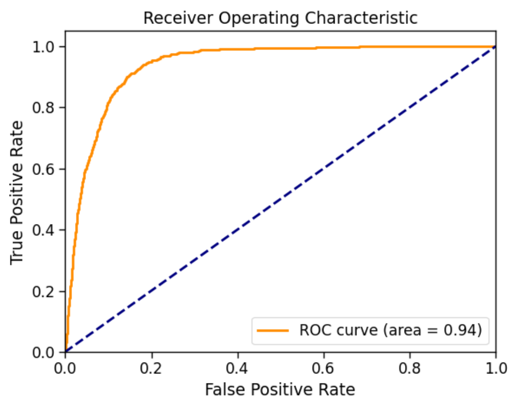
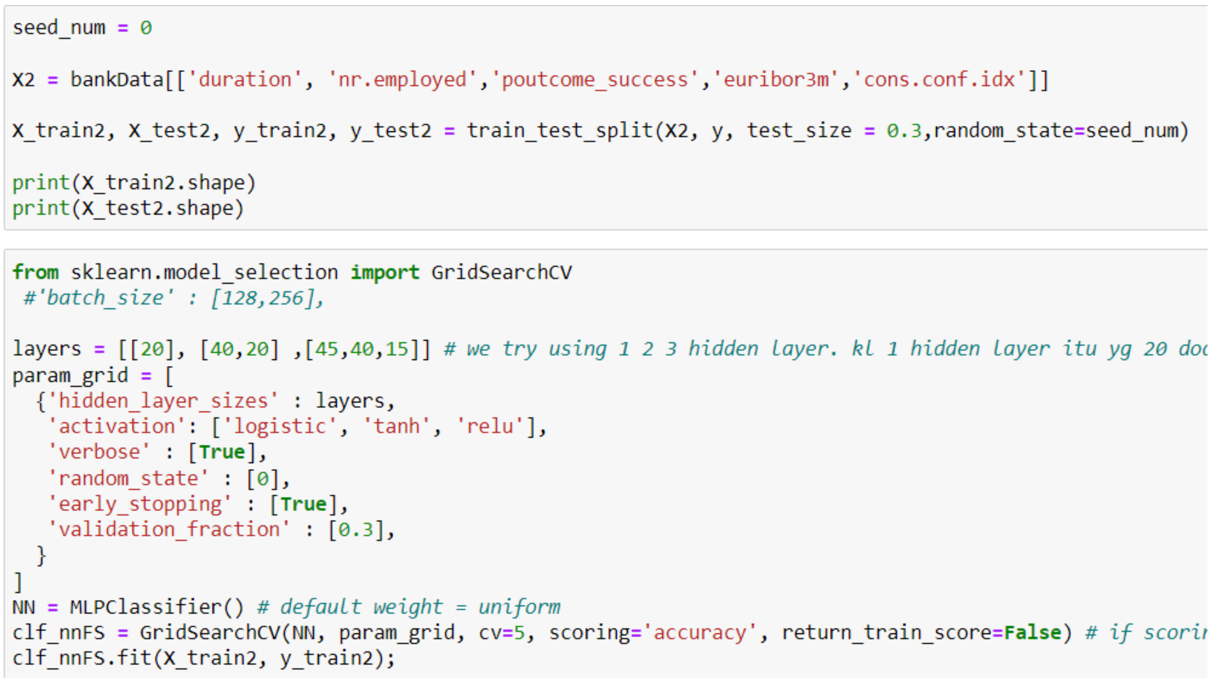


Figure 49. NN with Normalization

Figure 50. ROC

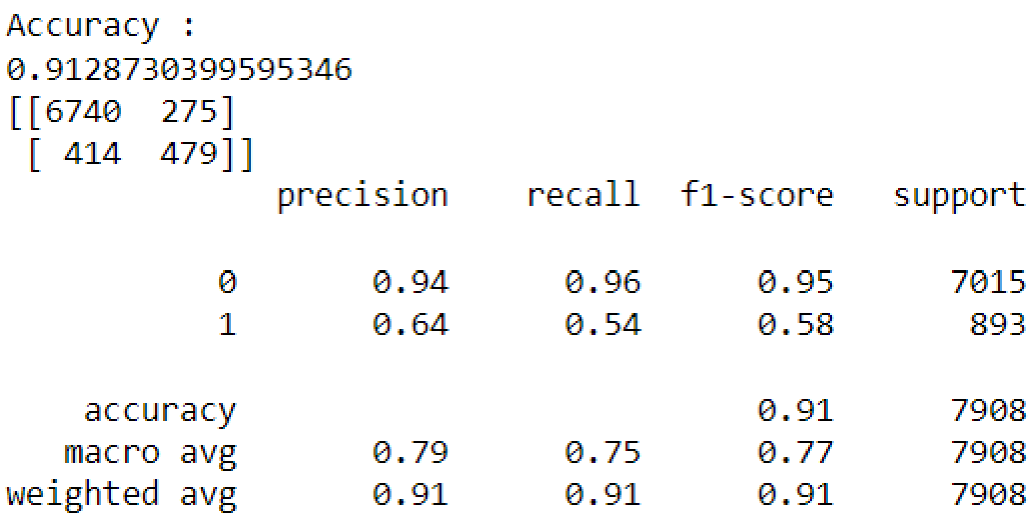
Based on Figure 49, it can be seen that the F1-score for the minority class improve significantly from 0.40 to 0.61. So, taking this into account, we will experiment on using normalization and feature selection now. The code is show here,



```
seed_num = 0

X2 = bankData[['duration', 'nr.employed','poutcome_success','euribor3m','cons.conf.idx']]

X_train2, X_test2, y_train2, y_test2 = train_test_split(X2, y, test_size = 0.3,random_state=seed_num)

print(X_train2.shape)
print(X_test2.shape)
```

```
from sklearn.model_selection import GridSearchCV
 #'batch_size' : [128,256],

layers = [[20], [40,20] ,[45,40,15]] # we try using 1 2 3 hidden layer. kl 1 hidden layer itu yg 20 doc
param_grid = [
  {'hidden_layer_sizes' : layers,
   'activation': ['logistic', 'tanh', 'relu'],
   'verbose' : [True],
   'random_state' : [0],
   'early_stopping' : [True],
   'validation_fraction' : [0.3],
  }
]
NN = MLPClassifier() # default weight = uniform
clf_nnFS = GridSearchCV(NN, param_grid, cv=5, scoring='accuracy', return_train_score=False) # if scori
clf_nnFS.fit(X_train2, y_train2);
```

Figure 51. Feature Selection (FS) on Neural Network



```
Accuracy :
0.9128730399595346
[[6740  275]
 [ 414  479]]
              precision    recall  f1-score   support

           0       0.94      0.96      0.95      7015
           1       0.64      0.54      0.58       893

    accuracy                           0.91      7908
   macro avg       0.79      0.75      0.77      7908
weighted avg       0.91      0.91      0.91      7908
```

Figure 52. FS Neural Network Confusion Matrix

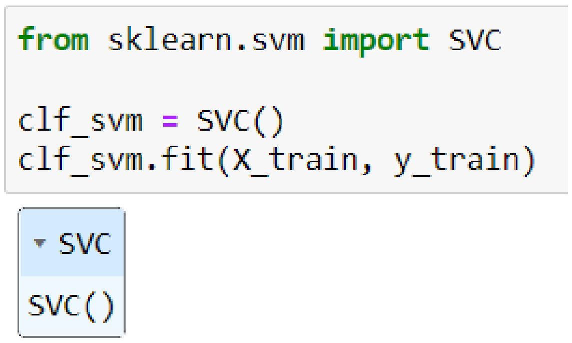
Result Table

Model	Accuracy	Recall class 0	Recall class 1	F1 class 0	F1 class 1	Parameter	Kaggle Score
Neural Network(normal)	0.9002	0.98	0.28	0.95	0.39	default	Not tested
Neural Network(Optimized)	0.91236	0.97	0.40	0.95	0.49	{'activation': 'logistic',	0.90834

						'early_stopping': True, 'hidden_layer_sizes': [40, 20], 'random_state': 0, 'validation_fraction': 0.3, 'verbose': True}	
Neural Network (Optimized Normalized)	0.91426	0.96	0.59	0.95	0.61	{'activation': 'logistic', 'early_stopping': True, 'hidden_layer_sizes': [40, 20], 'random_state': 0, 'validation_fraction': 0.3, 'verbose': True}	0.91745

Support Vector Machine:

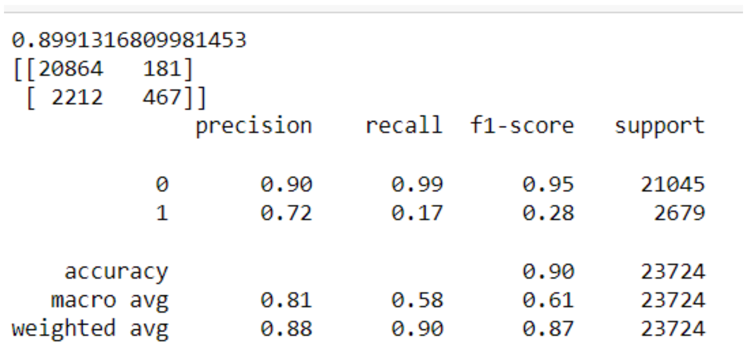
SVM in python is built using a library from scikit-learn called SVC. To develop a SVM model in python will only need these few lines of codes but the time it took to built one is very time consuming. So, in this report, we only use 5% of the dataset to train our model and the rest of 95% will be used to test the model. Moreover, we also perform Feature Selection Due to all this reason, we will not focus on getting the best performance amongst other algorithm like Decision Tree. Instead, we will focus on optimizing the model performance based on the experiment we got from other models.



```
from sklearn.svm import SVC

clf_svm = SVC()
clf_svm.fit(X_train, y_train)
```

```
▾ SVC
SVC()
```

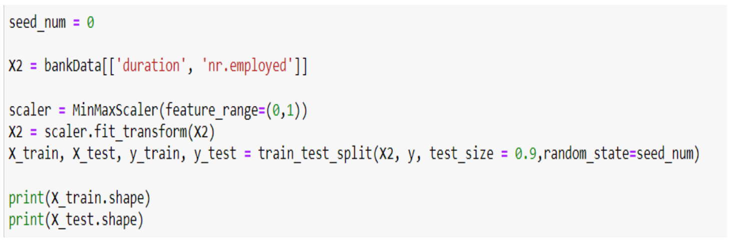
Figure 53. SVM



```
0.8991316809981453
[[20864   181]
 [ 2212   467]]
              precision    recall  f1-score   support

           0       0.90      0.99      0.95     21045
           1       0.72      0.17      0.28      2679

    accuracy                           0.90     23724
   macro avg       0.81      0.58      0.61     23724
weighted avg       0.88      0.90      0.87     23724
```

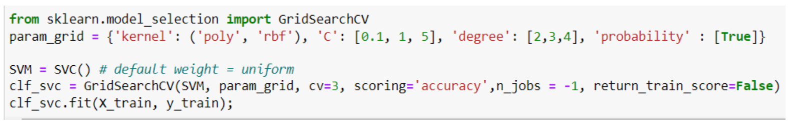
Figure 54. SVM Performance

So, now we will apply all methods such as normalization and feature selection to SVC to improve the model performance.



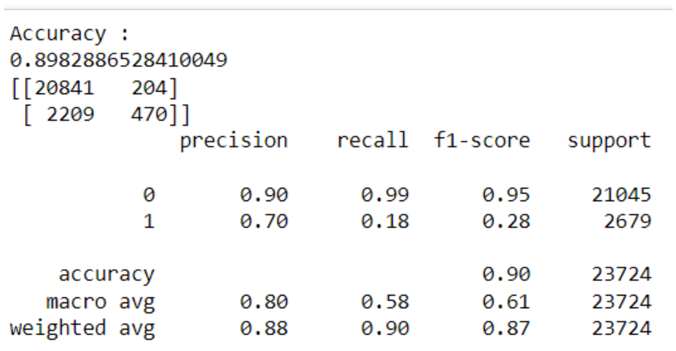
```
seed_num = 0

X2 = bankData[['duration', 'nr.employed']]

scaler = MinMaxScaler(feature_range=(0,1))
X2 = scaler.fit_transform(X2)
X_train, X_test, y_train, y_test = train_test_split(X2, y, test_size = 0.9,random_state=seed_num)

print(X_train.shape)
print(X_test.shape)
```

Figure 55. Feature Selection and Normalization



```
from sklearn.model_selection import GridSearchCV
param_grid = {'kernel': ('poly', 'rbf'), 'C': [0.1, 1, 5], 'degree': [2,3,4], 'probability' : [True]}

SVM = SVC() # default weight = uniform
clf_svc = GridSearchCV(SVM, param_grid, cv=3, scoring='accuracy',n_jobs = -1, return_train_score=False)
clf_svc.fit(X_train, y_train);
```

Figure 56. GridSearch on SVM



```
Accuracy :
0.8982886528410049
[[20841   204]
 [ 2209   470]]
              precision    recall  f1-score   support

           0       0.90      0.99      0.95     21045
           1       0.70      0.18      0.28      2679

    accuracy                           0.90     23724
   macro avg       0.80      0.58      0.61     23724
weighted avg       0.88      0.90      0.87     23724
```

Figure 57. SVM Performance

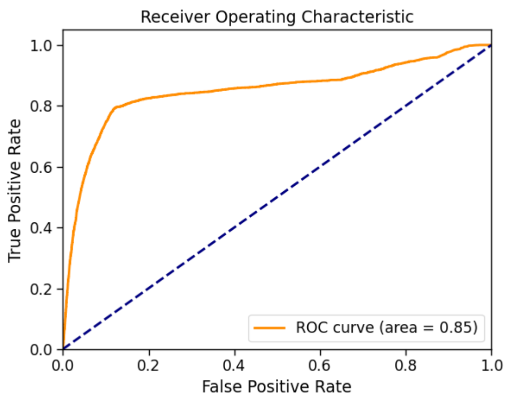


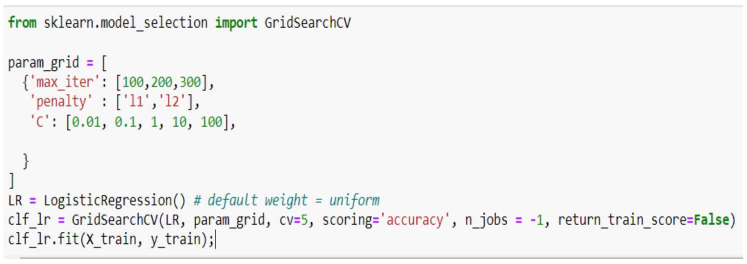
Figure 58. SVM ROC

Result Table

Model	Accuracy	Recall class 0	Recall class 1	F1 class 0	F1 class 1	Parameter	Kaggle Score
SVM (Optimization + Normalization + Feature Selection)	0.8982	0.99	0.18	0.95	0.28	`{'kernel': ('poly', 'rbf'), 'C': [0.1, 1, 5], 'degree': [2,3,4], 'probability' : [True]}`	0.91289

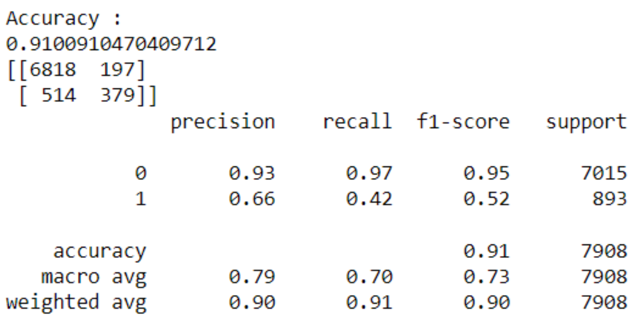
Logistic Regression:

Logistic regression is an algorithm used for solving classification problems although its name is regression. To build logistic regression in python we will need to import a class from scikit-learn called LogisticRegression(). In order for Logistic Regression to work well, we need to normalize our data. The code and the performance are shown below.



```python
from sklearn.model_selection import GridSearchCV

param_grid = [
  {'max_iter': [100,200,300],
   'penalty' : ['l1','l2'],
   'C': [0.01, 0.1, 1, 10, 100],

  }
]
LR = LogisticRegression() # default weight = uniform
clf_lr = GridSearchCV(LR, param_grid, cv=5, scoring='accuracy', n_jobs = -1, return_train_score=False)
clf_lr.fit(X_train, y_train);
```

Figure 59. Logistic Regression



```
Accuracy :
0.9100910470409712
[[6818  197]
 [ 514  379]]
              precision    recall  f1-score   support

           0       0.93      0.97      0.95      7015
           1       0.66      0.42      0.52       893

    accuracy                           0.91      7908
   macro avg       0.79      0.70      0.73      7908
weighted avg       0.90      0.91      0.90      7908
```

Figure 60. LR Performance

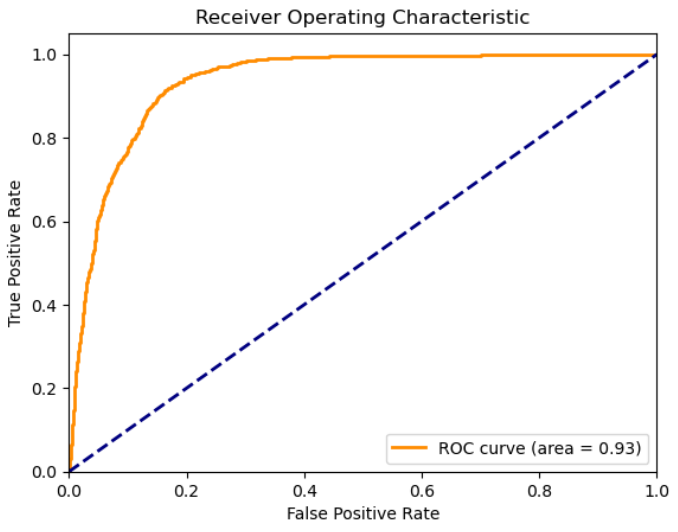


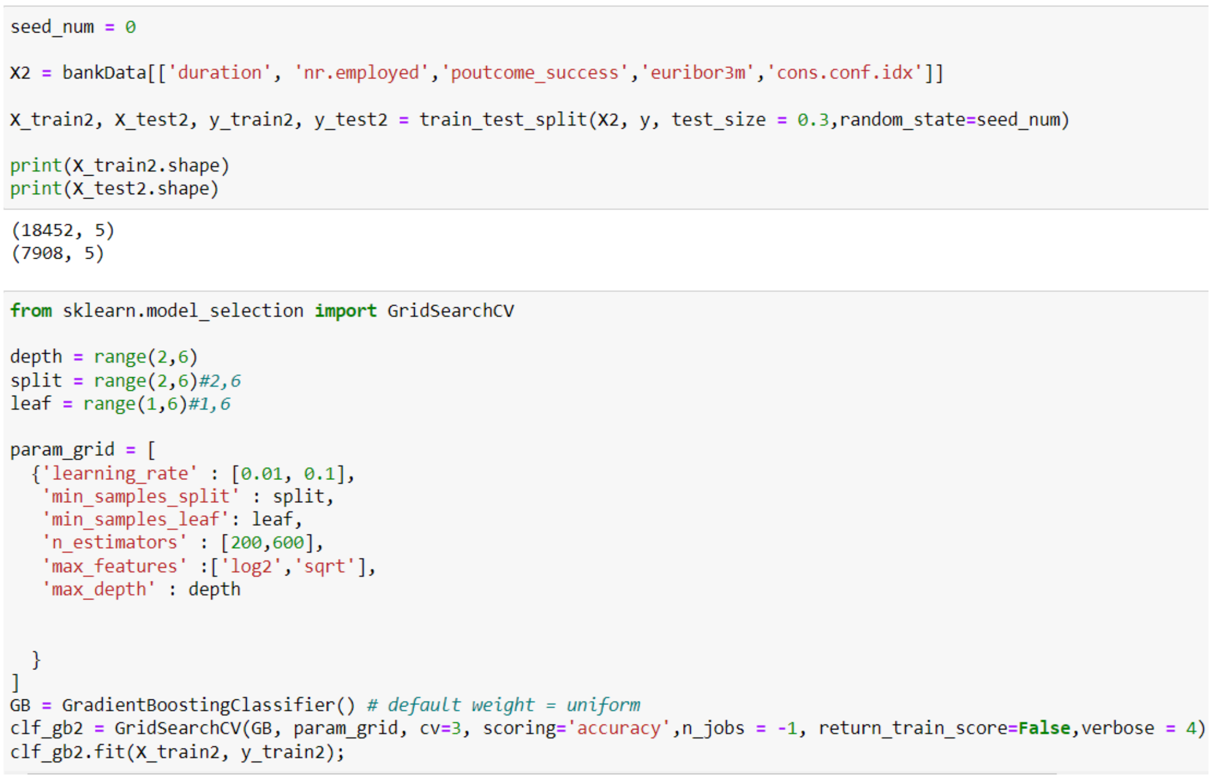
Figure 61. LR ROC

Result Table

Model	Accuracy	Recall class 0	Recall class 1	F1 class 0	F1 class 1	Parameter	Kaggle Score
Logistic Regression	0.9100	0.97	0.42	0.95	0.52	`{'C': 100, 'max_iter': 100, 'penalty': 'l2'}`	0.90682

Gradient Boosting:

Gradient Boosting in python is built using a library from scikit-learn called GradientBoostingClassifier. Gradient Boosting Algorithm is similar to Decision Tree and Random Forest. It does not need any normalization for it to work well. To develop Gradient Boosting algorithm in python, this code is needed:



```
seed_num = 0

X2 = bankData[['duration', 'nr.employed','poutcome_success','euribor3m','cons.conf.idx']]

X_train2, X_test2, y_train2, y_test2 = train_test_split(X2, y, test_size = 0.3,random_state=seed_num)

print(X_train2.shape)
print(X_test2.shape)
```

```
(18452, 5)
(7908, 5)
```

```
from sklearn.model_selection import GridSearchCV

depth = range(2,6)
split = range(2,6)#2,6
leaf = range(1,6)#1,6

param_grid = [
  {'learning_rate' : [0.01, 0.1],
   'min_samples_split' : split,
   'min_samples_leaf': leaf,
   'n_estimators' : [200,600],
   'max_features' :['log2','sqrt'],
   'max_depth' : depth


  }
]
GB = GradientBoostingClassifier() # default weight = uniform
clf_gb2 = GridSearchCV(GB, param_grid, cv=3, scoring='accuracy',n_jobs = -1, return_train_score=False,verbose = 4)
clf_gb2.fit(X_train2, y_train2);
```

Figure 62. Gradient Boosting Classifier

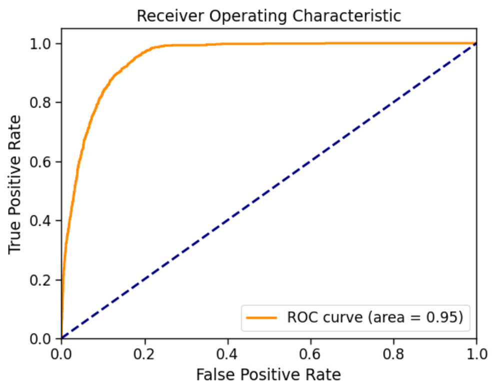
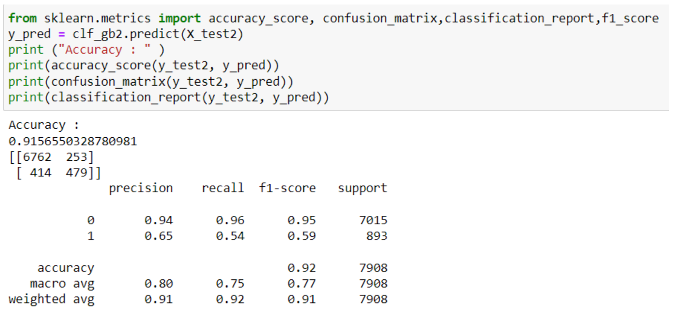


Figure 63. ROC



```
from sklearn.metrics import accuracy_score, confusion_matrix,classification_report,f1_score
y_pred = clf_gb2.predict(X_test2)
print ("Accuracy : " )
print(accuracy_score(y_test2, y_pred))
print(confusion_matrix(y_test2, y_pred))
print(classification_report(y_test2, y_pred))
```

```
Accuracy :
0.9156550328780981
[[6762  253]
 [ 414  479]]
              precision    recall  f1-score   support

           0       0.94      0.96      0.95      7015
           1       0.65      0.54      0.59       893

    accuracy                           0.92      7908
   macro avg       0.80      0.75      0.77      7908
weighted avg       0.91      0.92      0.91      7908
```

Figure 64. Confusion Matrix

Based on the 3 figures above, we can see that Gradient Boosting algorithm are able to perform well on binary classification problem. It can be seen and evaluated through the confusion matrix and ROC curve. Gradient Boosting is also known to be used for Machine Learning competition as it provides the best performance. However, we will test it out in the Kaggle. The result of the Kaggle and the overall summary is provided in the table below.

Result Table

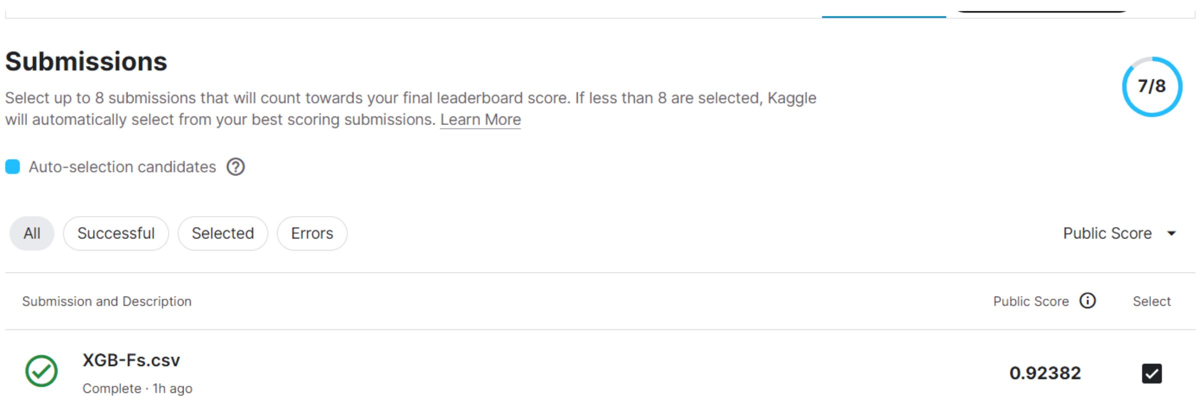
Model	Accuracy	Recall class 0	Recall class 1	F1 class 0	F1 class 1	Parameter	Kaggle Score
Gradient Boosting (Feature selection)	0.91565	0.96	0.54	0.95	0.59	`{'learning_rate': 0.01, 'max_depth': 5, 'max_features': 'log2', 'min_samples_leaf': 5, 'min_samples_split': 4, 'n_estimators': 600}`	0.92382

Best Classifier:

Model	Accuracy	Recall class 0	Recall class 1	F1 class 0	F1 class 1	Kaggle
Decision Tree	0.91476	0.97	0.52	0.95	0.58	0.92261
K-Nearest Neighbors	0.9107	0.97	0.47	0.95	0.54	0.91107
Random Forest	0.91236	0.97	0.46	0.95	0.54	0.92139
Neural Network	0.91426	0.96	0.59	0.95	0.61	0.91745
Support Vector Machine	0.8982	0.99	0.18	0.95	0.28	0.91289
Logistic Regression	0.9100	0.97	0.42	0.95	0.52	0.90682
Gradient Boosting	0.91565	0.96	0.54	0.95	0.59	0.92382

Out of the all model built, the best classifier for the bank marketing problem is Gradient Boosting. Gradient Boosting is able to achieve the highest performance in Kaggle with a score of 0.92382. Gradient Boosting is a tree-based algorithm that combines many models together to create a strong predictive model. This algorithm is widely used in many competitions if performance is the main target. Gradient Boosting works similar to Decision Tree and Random Forest because it is both a tree based and ensemble method classifier. Moreover, the reason Gradient Boosting outperform Random Forest and Decision Tree is because it trains itself to correct each other error (ensemble method) and Gradient Boosting are capable of capturing complex pattern in the data. Gradient Boosting relies on the intuition that the best possible next model, when combined with previous models, minimizes the overall prediction error. The key idea is to set the target outcomes for this next model in order to minimize the error. Hence, it all makes sense that Gradient Boosting is able to achieve the best score in this classification problems. The second-best model that also capture the pattern well is Decision Tree followed by Random Forest in third place. Thus, it can be deduced that tree-based algorithms are suitable for this bank marketing problems.

Kaggle Submission:



Submissions

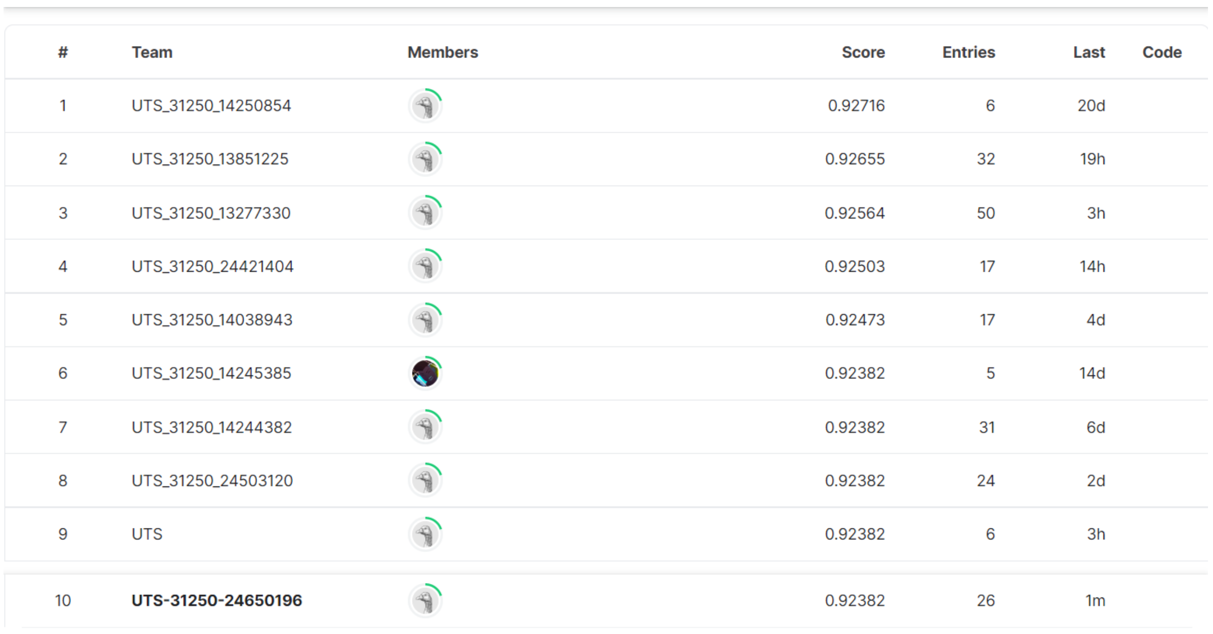
Select up to 8 submissions that will count towards your final leaderboard score. If less than 8 are selected, Kaggle will automatically select from your best scoring submissions. Learn More

7/8

Auto-selection candidates

Submission and Description	Public Score	Select
XGB-Fs.csv Complete · 1h ago	0.92382	✓

Kaggle Best Score



#	Team	Members	Score	Entries	Last	Code
1	UTS_31250_14250854		0.92716	6	20d	
2	UTS_31250_13851225		0.92655	32	19h	
3	UTS_31250_13277330		0.92564	50	3h	
4	UTS_31250_24421404		0.92503	17	14h	
5	UTS_31250_14038943		0.92473	17	4d	
6	UTS_31250_14245385		0.92382	5	14d	
7	UTS_31250_14244382		0.92382	31	6d	
8	UTS_31250_24503120		0.92382	24	2d	
9	UTS		0.92382	6	3h	
10	**UTS-31250-24650196**		0.92382	26	1m	

Kaggle Ranking Position

Reference:

Kumar, V. (2021, August 19). *KNN Classifier in Sklearn using GridSearchCV with Example*. MLK - Machine Learning Knowledge. https://machinelearningknowledge.ai/knn-classifier-in-sklearn-using-gridsearchcv-with-example/

G, E. (2018, October 27). *k-Neighbors Classifier with GridSearchCV Basics*. Medium. https://medium.com/@erikgreenj/k-neighbors-classifier-with-gridsearchcv-basics-3c445ddeb657

Martulandi, A. (2019, November 4). *Increase 10% Accuracy with Re-scaling Features in K-Nearest Neighbors + Python Code*. Medium. https://medium.datadriveninvestor.com/increase-10-accuracy-with-re-scaling-features-in-k-nearest-neighbors-python-code-677d28032a45

Raschka, S. (2022, November 3). *When should I apply data normalization/standardization?* Dr. Sebastian Raschka. https://sebastianraschka.com/faq/docs/when-to-standardize.html

How to choose a predictive model after k-fold cross-validation? (n.d.). Cross Validated. Retrieved November 3, 2022, from https://stats.stackexchange.com/questions/52274/how-to-choose-a-predictive-model-after-k-fold-cross-validation

Santos, G. (2021, August 18). *How to do Cross-Validation, KFold and Grid Search in Python*. Gustavorsantos. https://medium.com/gustavorsantos/how-to-do-cross-validation-kfold-and-grid-search-in-python-e570cdb20a28

Maheshwari, H. (2021, October 13). *How to decide the perfect distance metric for your machine learning model*. Medium. https://towardsdatascience.com/how-to-decide-the-perfect-distance-metric-for-your-machine-learning-model-2fa6e5810f11

What is Overfitting in Deep Learning [+10 Ways to Avoid It]. (n.d.). Www.v7labs.com. Retrieved November 3, 2022, from https://www.v7labs.com/blog/overfitting#h3