

# Assignment 1: Robinhood Analysis

*Stella L. Sob*

## An Overview of The Datasets

The two datasets that I have been given are: [equity\\_value\\_data.csv](#) and [features\\_data.csv](#). The equity\_value\_data is a 1000000 (rows) x 3 (columns) dataset. The input attributes are: 1 - timestamp, 2- close\_equity, and 3- user\_id.

The features\_data is a 5585 (rows) x 9 (columns) dataset. The input attributes are: 1- investment\_experience, 2 - liquidity\_needs, 3 - platform, 4 - time\_spent, 5 - instrument\_type\_first\_traded, 6 - first\_deposit\_amount, 7 - time\_horizon, 8 - user\_id

## Approach

First, I read in the equity\_value\_data.csv into DataFrame, equity\_df. I confirmed with equity\_df.shape that there are 1119158 rows x 3 columns. There is no NaN values in the timestamp, close\_equity and user\_id columns. I converted the equity\_df DataFrame to a pandas datetime object. I then found out the number of unique timestamps, close\_equity and user\_id:

```
>>> equity_df.nunique()
timestamp      255
close_equity   531483
user_id        5584
dtype: int64
```

I determined the missing days (gap\_days) from equity\_df, and displayed the first 10 rows of data:

```
>>> equity_df['gap_days'] = equity_df.timestamp.diff()
>>> equity_df.head(10)
```

	timestamp	close_equity	user_id	gap_days
0	2016-11-16 00:00:00+00:00	48.16	bcef4fa9b0bdf22bcf7deae708decf03	NaT
1	2016-11-17 00:00:00+00:00	48.16	bcef4fa9b0bdf22bcf7deae708decf03	1 days
2	2016-11-18 00:00:00+00:00	48.16	bcef4fa9b0bdf22bcf7deae708decf03	1 days
3	2016-11-21 00:00:00+00:00	48.16	bcef4fa9b0bdf22bcf7deae708decf03	3 days
4	2016-11-22 00:00:00+00:00	48.16	bcef4fa9b0bdf22bcf7deae708decf03	1 days

```

5 2016-11-23 00:00:00+00:00    48.16 bcef4fa9b0bdf22bcf7deae708decf03  1 days
6 2016-11-25 00:00:00+00:00    48.16 bcef4fa9b0bdf22bcf7deae708decf03  2 days
7 2016-11-28 00:00:00+00:00    48.16 bcef4fa9b0bdf22bcf7deae708decf03  3 days
8 2016-11-29 00:00:00+00:00    48.16 bcef4fa9b0bdf22bcf7deae708decf03  1 days
9 2016-11-30 00:00:00+00:00    48.16 bcef4fa9b0bdf22bcf7deae708decf03  1 days
>>>

```

To find out the number of users who have churned, I find out the number of unique users who have not logged on for more than 28 days:

```

num_churned_users = equity_df[equity_df['gap_days'] >= pd.Timedelta('28
days')]['user_id'].nunique()

>>> num_churned_users
287

```

and the total number of unique user\_ids:

```

total_unique_user_ids = equity_df['user_id'].unique()
>>> len(total_unique_user_ids)
5584

```

I obtained the percentage of churned users as follows:

```

percent_of_churned_users = np.round(
(num_churned_users)/len(total_unique_user_ids) * 100, decimals = 2)

```

```

>>> percent_of_churned_users = np.round((num_churned_users)/len(total_unique_user_ids) *
100, decimals = 2)
>>> print('The percentage of churned users = {0}%'.format(percent_of_churned_users))
The percentage of churned users = 5.14%
>>>

```

a) The percentage of users who have churned in the data provided = 5.14 %

## Part B - Building a Classifier

## Approach

I read in features\_data.csv into a features\_df DataFrame. features\_df has 5584 rows by 9 columns. From churned\_user\_ids array (.unique() returns churned\_user\_ids in an array), I create a churned\_df DataFrame, with a column 'churn' set to 1:

```
churned_df = pd.DataFrame(churned_user_ids, columns = ['user_id'])
churned_df['churn'] = 1
churned_df.head()
```

```
>>> churned_df.head()
```

	user_id	churn
0	270cda53a026bcf6c2b98492b23c1b99	1
1	319c069e77187c7e7c027eb00fd941	1
2	ec84f134c5b27c4f5702803e98eb3f40	1
3	6dbedf7f2972b4f6ea60a15d48cb292c	1
4	91837d41270b81b267fc205c3e03ee9b	1

I then merged features\_df and churned\_df into a DataFrame called model\_df, and fill the NaN values with 0. The first 5 data on model\_df is as follows:

```
>>> model_df.head()
```

	risk_tolerance	investment_experience	liquidity_needs	platform	...	first_deposit_amount
	time_horizon	user_id	churn			
0	high_risk_tolerance	limited_investment_exp	very_important_liq_need	Android	...	40.0
	med_time_horizon	895044c23edc821881e87da749c01034	0.0			
1	med_risk_tolerance	limited_investment_exp	very_important_liq_need	Android	...	200.0
	short_time_horizon	458b1d95441ced242949deefe8e4b638	0.0			
2	med_risk_tolerance	limited_investment_exp	very_important_liq_need	iOS	...	25.0
	long_time_horizon	c7936f653d293479e034865db9bb932f	0.0			
3	med_risk_tolerance	limited_investment_exp	very_important_liq_need	Android	...	100.0
	short_time_horizon	b255d4bd6c9ba194d3a350b3e76c6393	0.0			
4	high_risk_tolerance	limited_investment_exp	very_important_liq_need	Android	...	20.0
	long_time_horizon	4a168225e89375b8de605cbc0977ae91	0.0			

```
[5 rows x 10 columns]
```

A concise summary could be obtained as follows:

```
>>> model_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
```

```
Int64Index: 5584 entries, 0 to 5583
```

```
Data columns (total 10 columns):
```

#	Column	Non-Null Count	Dtype
0	risk_tolerance	5584 non-null	object
1	investment_experience	5584 non-null	object
2	liquidity_needs	5584 non-null	object
3	platform	5584 non-null	object
4	time_spent	5584 non-null	float64
5	instrument_type_first_traded	5584 non-null	object
6	first_deposit_amount	5584 non-null	float64
7	time_horizon	5584 non-null	object
8	user_id	5584 non-null	object
9	churn	5584 non-null	float64

```
dtypes: float64(3), object(7)
```

```
memory usage: 479.9+ KB
```

The number of unique values of each column:

```
>>> model_df.nunique()
```

risk_tolerance	3
investment_experience	4
liquidity_needs	3
platform	3
time_spent	4502
instrument_type_first_traded	11
first_deposit_amount	317
time_horizon	3
user_id	5584
churn	2

```
dtype: int64
```

and the unique column labels:

```
>>> model_df.columns
```

```
Index(['risk_tolerance', 'investment_experience', 'liquidity_needs',  
      'platform', 'time_spent', 'instrument_type_first_traded',  
      'first_deposit_amount', 'time_horizon', 'user_id', 'churn'],  
      dtype='object')
```

Apply [OrdinalEncoding](#) to categorical features such that they are converted to ordinal integers.

```
encode_cat_feat = OrdinalEncoder()
```

fit\_transform() performs standardization to bring down all the features to a common scale without distorting the differences in the range of the values:

```
new_df_transformed = encode_cat_feat.fit_transform(new_df)
```

I then transposed the index and columns i.e. reflect the new\_df DataFrame over the main diagonals such that the rows are written as columns and vice-versa:

```
for i, j in enumerate(new_df.columns):  
    new_df[j] = new_df_transformed.transpose()[i]
```

and added converted labels to model\_df:

```
for i in model_df.columns:  
    for i in new_df.columns:  
        model_df[i] = new_df[i]
```

```
>>> model_df.head()
```

	risk_tolerance	investment_experience	liquidity_needs	platform	time_spent	instrument_type_first_traded	first_deposit_amount	time_horizon	user_id	churn
0	0.0	2.0	2.0	0.0	33.129417	8.0	40.0	1.0	2970.0	0.0
1	2.0	2.0	2.0	0.0	16.573517	8.0	200.0	2.0	1512.0	0.0
2	2.0	2.0	2.0	2.0	10.008367	8.0	25.0	0.0	4326.0	0.0
3	2.0	2.0	2.0	0.0	1.031633	8.0	100.0	2.0	3854.0	0.0

```

4      0.0      2.0      2.0      0.0  8.187250      8.0      20.0      0.0 1615.0
0.0
>>>

```

Next, we will detect outliers using seaborn box-plots:

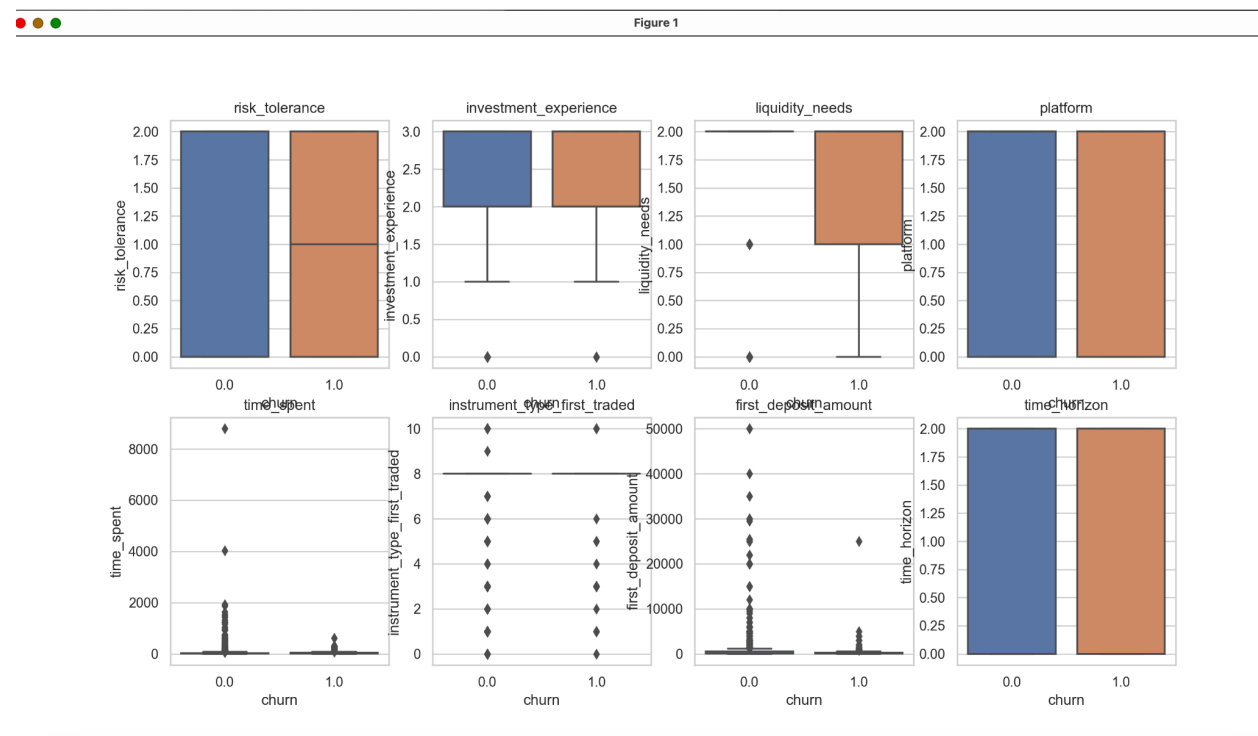
```

sns.set(style="whitegrid")
fig, axes = plt.subplots(2, 4, figsize = (20, 8))

count = 0
for i in range(2):
    for j in range(4):
        sns.boxplot(x = 'churn', y = model_df.columns[count], data = model_df,
ax = axes[i,j])
        count += 1

```

Then we proceed to detect the outliers using seaborn box-plots:



*Fig. (1) Seaborn box-plots to detect outliers*

From Fig. (1) above, we see that investment\_experience, liquidity\_needs, instrument\_type\_first\_traded, first\_deposit\_amount and time\_spent have much wider ranges than risk\_tolerance, platform and time\_horizon.

Without removing the imbalance of the classes (yet), we now get a peek into the class distribution of churned and non-churned users. We calculate the percentage of non-churned users:

```
num_non_churned_users = len(model_df[model_df['churn'] == 0])
percent_of_non_churned_users = (num_non_churned_users /
len(total_unique_user_ids)) * 100
print(f'Percentage of Non-churn events in the
data:{percent_of_non_churned_users}')
```

```
>>> num_non_churned_users
5297
>>> len(total_unique_user_ids)
5584
>>> percent_of_non_churned_users = (num_non_churned_users / len(total_unique_user_ids)) *
100
>>> print(f'Percentage of Non-churn events in the data:{percent_of_non_churned_users}')
Percentage of non-churn events in the data:94.86031518624641
```

And the percentage of churned users:

```
num_churned_users = len(model_df[model_df['churn'] == 1])
percent_of_churned_users = (num_churned_users / len(total_unique_user_ids)) *
100
print(f'Percentage of churned events in the data:{percent_of_churned_users}')
```

```
>>> num_churned_users
287
>>> percent_of_churned_users = (num_churned_users / len(total_unique_user_ids)) * 100
>>> percent_of_churned_users
5.139684813753582
```

We will now view the class distribution:

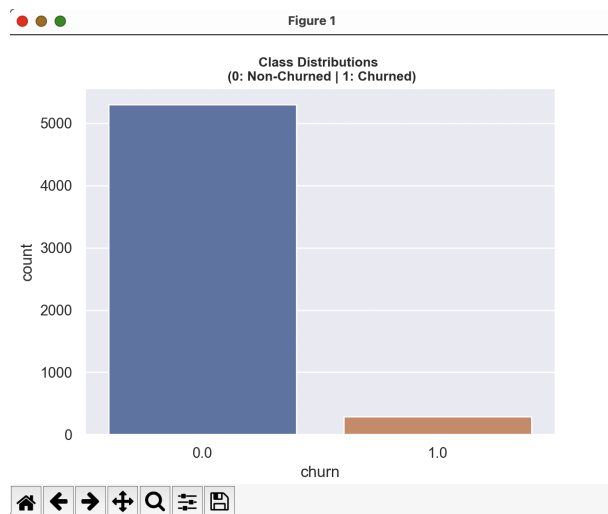


Fig. (2) Class Distribution of non-churned vs churned events

Clearly, it shows in Fig. (2) above that there is an imbalanced distribution between churned and non-churned events.

We will do feature selection using Chi-square Test. First we drop the columns of 'user\_id' and 'churn' in model\_df, and assign all the data to X. The column 'churn' would be assigned to y:

```
X = model_df.drop(columns = ['user_id', 'churn'])
y = model_df['churn']
```

I then splitted the data into 80% training set and 20% testing set:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,
random_state = 42)
```

```
print(X_train.shape)
print(y_train.shape)
```

```
>>> X = model_df.drop(columns = ['user_id', 'churn'])
```

```
>>> X
```

```
risk_tolerance investment_experience liquidity_needs platform time_spent
instrument_type_first_traded first_deposit_amount time_horizon
0      0.0      2.0      2.0      0.0 33.129417      8.0      40.0      1.0
1      2.0      2.0      2.0      0.0 16.573517      8.0     200.0      2.0
2      2.0      2.0      2.0      2.0 10.008367      8.0      25.0      0.0
3      2.0      2.0      2.0      0.0  1.031633      8.0     100.0      2.0
4      0.0      2.0      2.0      0.0  8.187250      8.0      20.0      0.0
```



...	...	...	...	...	...	...	...	...
5579	0.0	2.0	2.0	0.0	8.339283	8.0	300.0	0.0
5580	2.0	2.0	1.0	2.0	7.241383	8.0	100.0	2.0
5581	2.0	3.0	2.0	1.0	22.967167	8.0	50.0	2.0
5582	2.0	2.0	1.0	2.0	10.338417	8.0	100.0	0.0
5583	0.0	2.0	1.0	2.0	18.470950	8.0	50.0	0.0

[5584 rows x 8 columns]

```
>>> y = model_df['churn']
```

```
>>> y
```

```
0    0.0
```

```
1    0.0
```

```
2    0.0
```

```
3    0.0
```

```
4    0.0
```

...

```
5579  0.0
```

```
5580  1.0
```

```
5581  0.0
```

```
5582  0.0
```

```
5583  0.0
```

Name: churn, Length: 5584, dtype: float64

```
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 42)
```

```
>>> print(X_train.shape)
```

```
(4467, 8)
```

```
>>> print(y_train.shape)
```

```
(4467,)
```

Using SelectKBest, I selected features that contribute most to the target variable before moving on to modeling the data:

```
>>> X_test_fs.shape
```

```
(1117, 8)
```

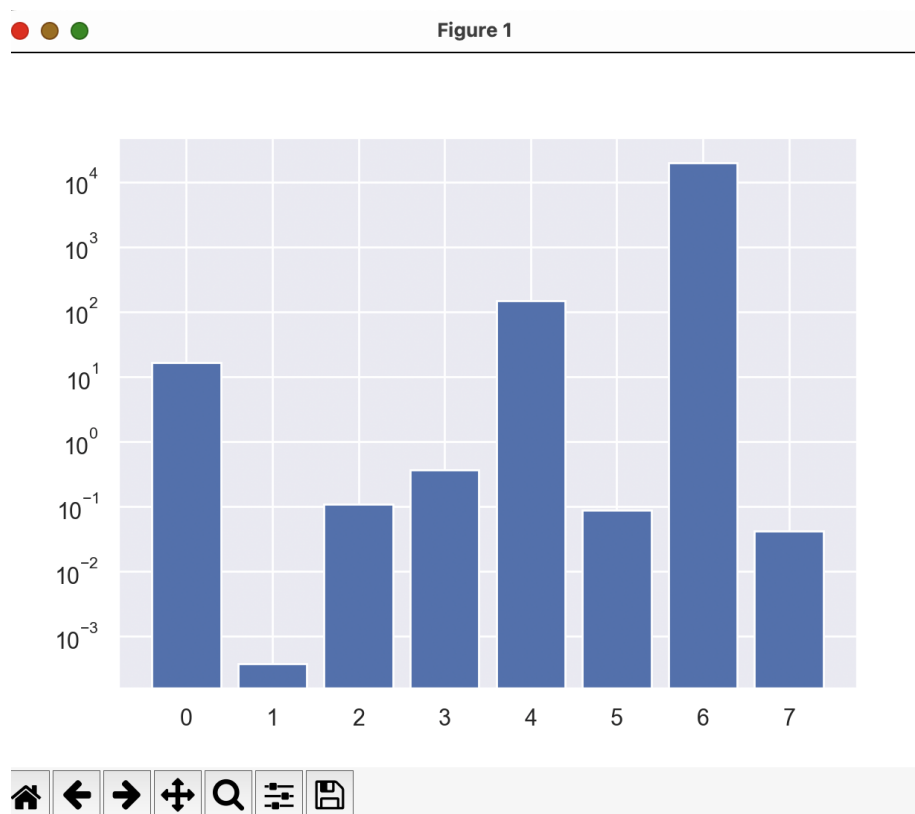
```
>>> X_train_fs.shape
```

```
(4467, 8)
```

I printed out the scores for each feature:

```
>>> best_features_scores = fs.scores_  
>>> for i in range(len(best_features_scores)):  
...     print('Feature %d: %f' % (i, best_features_scores[i]))  
...  
Feature 0: 16.511612  
Feature 1: 0.000381  
Feature 2: 0.109252  
Feature 3: 0.363341  
Feature 4: 147.511208  
Feature 5: 0.088567  
Feature 6: 19774.184590  
Feature 7: 0.042232
```

and plotted the scores. The y-axis has been set to logarithmic scale:



*Fig. (3) Feature scores for the original, imbalanced dataset*

Next, I will examine the accuracy, F-1 score, and ROC\_AUC scores by training the data using Random Forest Classifier and running confusion matrix. The mean accuracy of X\_test is:

```
>>> rf_clf = RandomForestClassifier(n_estimators = 300)
>>> rf_clf = rf_clf.fit(X_train, y_train)
>>> mean_accuracy = rf_clf.score(X_test, y_test) * 100
>>> print('Accuracy of X_test (in %): ', mean_accuracy)
Accuracy of X_test (in %): 94.18084153983885
```

and the F-1 score is:

```
>>> y_pred = rf_clf.predict(X_test)
>>> f1 = f1_score(y_test, y_pred)
>>> print('F1 score: ', f1)
F1 score: 0.0
```

The confusion matrix is displayed as follows:

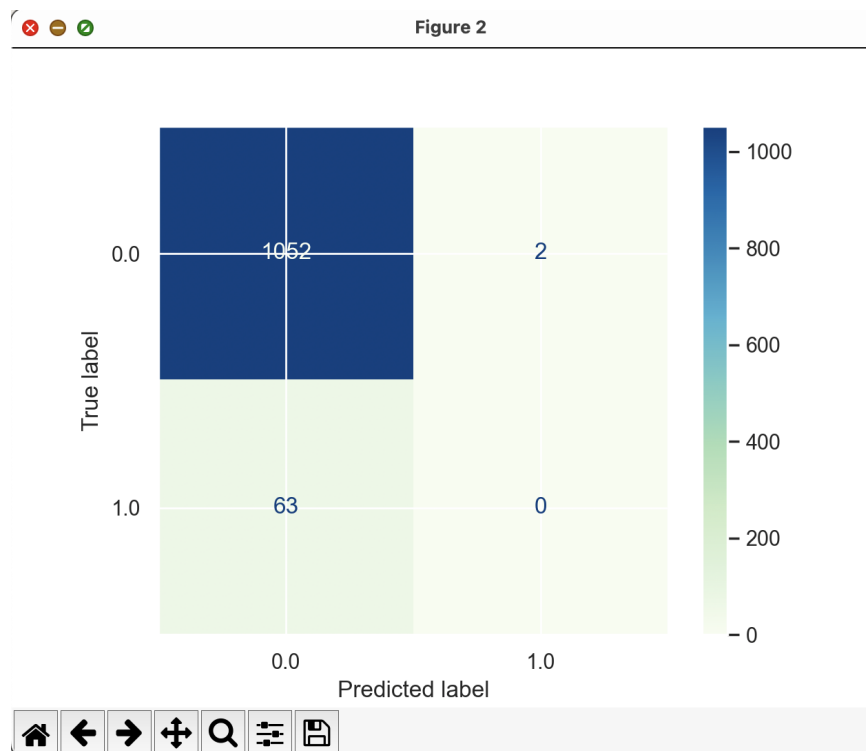


Fig. (4) confusion matrix for original, imbalanced dataset

```
>>> roc_auc = roc_auc_score(y_test, rf_clf.predict_proba(X_test)[: , 1])
```

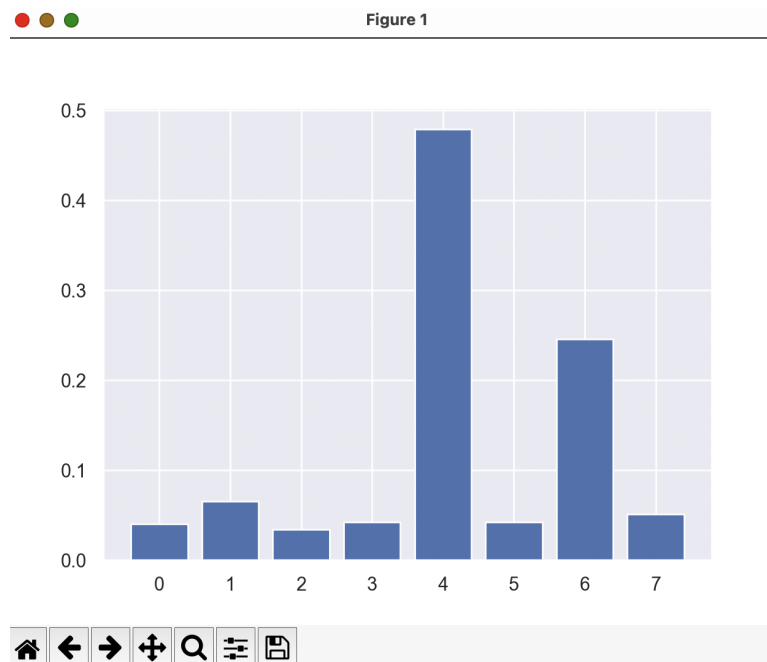
```
>>> print('AUC Score =', roc_auc)
AUC Score = 0.5859160868648534
```

We have a good accuracy at 94.18%, F-1 score at 0.0 and ROC\_AUC score at 0.5859. The ROC\_AUC score tells us how efficient the model is. The higher the AUC, the better is the model at distinguishing between the positive and negative classes. This AUC score of 0.5859 ( $< 1$ ) indicates that there's only a 58.6% chance that the classifier distinguishes between the positive ('churned' in our case) and negative ('non-churned' in our case) classes. A small F-1 score ( $< 0.5$ ) could mean the classifier has a high number of false positives [There are 2 false positives (?) indicated by the confusion matrix] which could be the outcome of imbalanced classes. In such an imbalanced class problem, we have to prepare the data with Under/Over-sampling in order to curb these issues of False Positives (FP)/ False Negatives (FN).

We'll check on the feature importance before moving on to under-sampling the data.

```
>>> importance = rf_clf.feature_importances_
>>> print(importance)
[0.0401698 0.06577056 0.03411774 0.04262653 0.47870777 0.04211701
 0.24544315 0.05104742]
```

The plot is displayed as follows:



*Fig. (5) Feature importance before under-sampling*

Let us now try to balance the classes by under-sampling. I created a churned\_df and a non\_churned\_df DataFrames:

```
churned_df = model_df[model_df['churn'] == 1]
churned_df_len = len(churned_df)
```

```
non_churned_df = model_df[model_df['churn'] == 0][:churned_df_len]
```

```
non_churned_df_len = len(non_churned_df)
```

I then created a balanced\_df DataFrame by concatenating the churned\_df and non\_churned\_df:

```
balanced_df = pd.concat([churned_df, non_churned_df])
```

100% of the random sample (essentially, I am shuffling) is returned from this balanced\_df as:

```
shuffled_from_balanced_df = balanced_df.sample(frac=1, random_state=None)
```

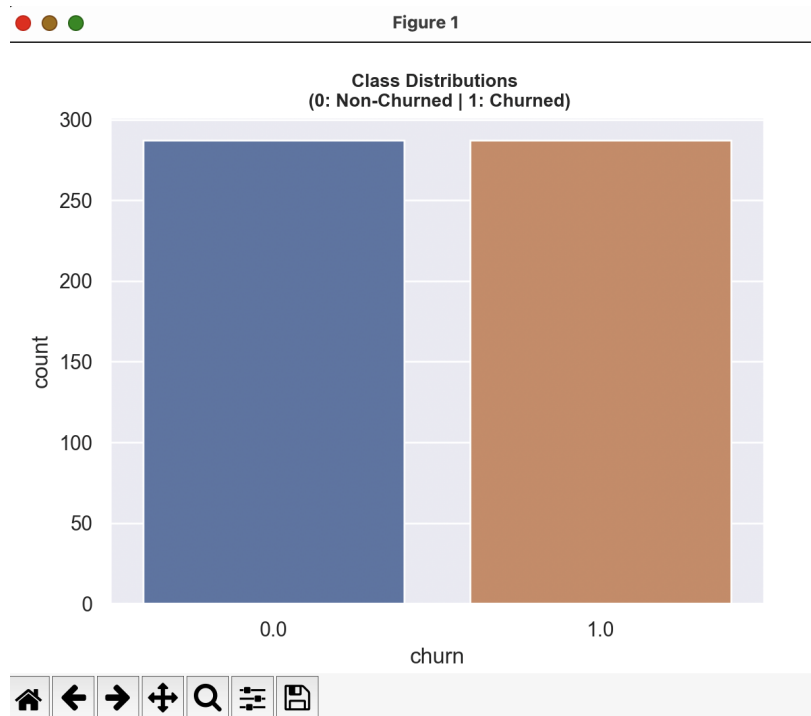
```
>>> shuffled_from_balanced_df.shape
```

```
(574, 10)
```

```
>>> shuffled_from_balanced_df.head()
```

	risk_tolerance	investment_experience	liquidity_needs	platform	time_spent	instrument_type	first_traded	first_deposit_amount	time_horizon
user_id	churn								
4136	0.0	2.0	2.0	0.0					
35.310950		8.0	30.00	2.0					
1087.0	1.0								
238	0.0	2.0	2.0	1.0					
0.000000		8.0	100.00	0.0					
5526.0	0.0								
96	0.0	1.0	1.0	0.0					
27.596250		8.0	1000.00	2.0					
4775.0	0.0								
7	0.0	1.0	1.0	2.0					
0.000000		8.0	100.00	2.0					
4107.0	0.0								
79	0.0	3.0	2.0	2.0					
21.737033		8.0	10.49	2.0					
3226.0	0.0								

Let us see how the class distribution looks like now:



*Fig. (6) Class Distribution of non-churned vs churned events after under-sampling*

Let us now run chi-squared feature selection, examine the scores for the features, plot the scores. Then we'll run Random Forest Classifier, and examine accuracy, F-1 score and plot the confusion matrix.

Our dataset would now be data from shuffled\_from\_balanced\_df. I removed columns 'user\_id' and 'churn' from shuffled\_from\_balanced\_df, and assigned to X. The label consists of values from the 'churn' column.

```
>>> X = shuffled_from_balanced_df.drop(columns = ['user_id', 'churn'], axis = 1)
```

```
>>> X.shape
```

```
(574, 8)
```

```
>>> y = shuffled_from_balanced_df['churn']
```

```
>>> y.shape
```

```
(574,)
```

Now, I splitted the dataset into 75% training set and 25% testing set:

```
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25, random_state = 1)
```

```
>>> X_train.shape
```

```
(430, 8)
```

```
>>> X_test.shape
```

```
(144, 8)
```

Feature selection from this shuffled\_from\_balanced\_df using chi-square test yields:

```
>>> best_features_scores
```

```
array([1.07572377e+01, 4.79956888e-03, 3.15284155e-03, 5.18802855e-01,  
       2.03736303e+02, 2.20653103e-02, 3.97627624e+04, 5.35703723e-01])
```

```
>>> for i in range(len(best_features_scores)):
```

```
...     print('Feature %d: %f' % (i, best_features_scores[i]))
```

```
...
```

```
Feature 0: 10.757238
```

```
Feature 1: 0.004800
```

```
Feature 2: 0.003153
```

```
Feature 3: 0.518803
```

```
Feature 4: 203.736303
```

```
Feature 5: 0.022065
```

```
Feature 6: 39762.762442
```

```
Feature 7: 0.535704
```

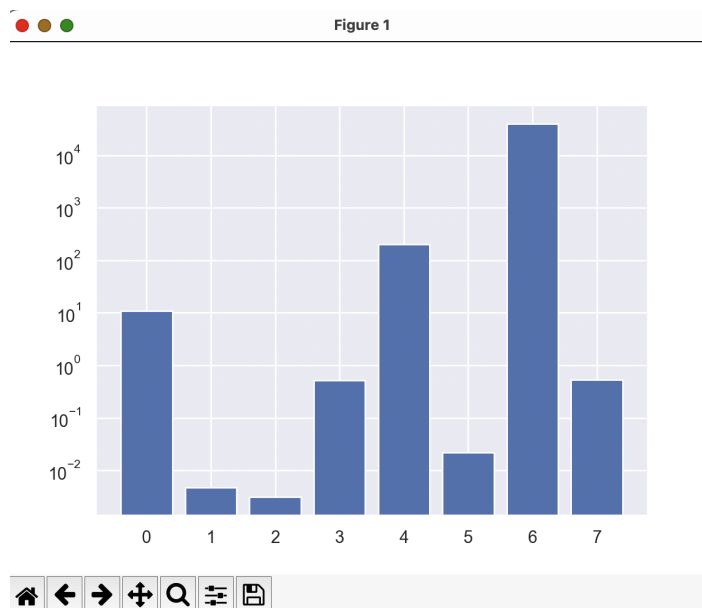


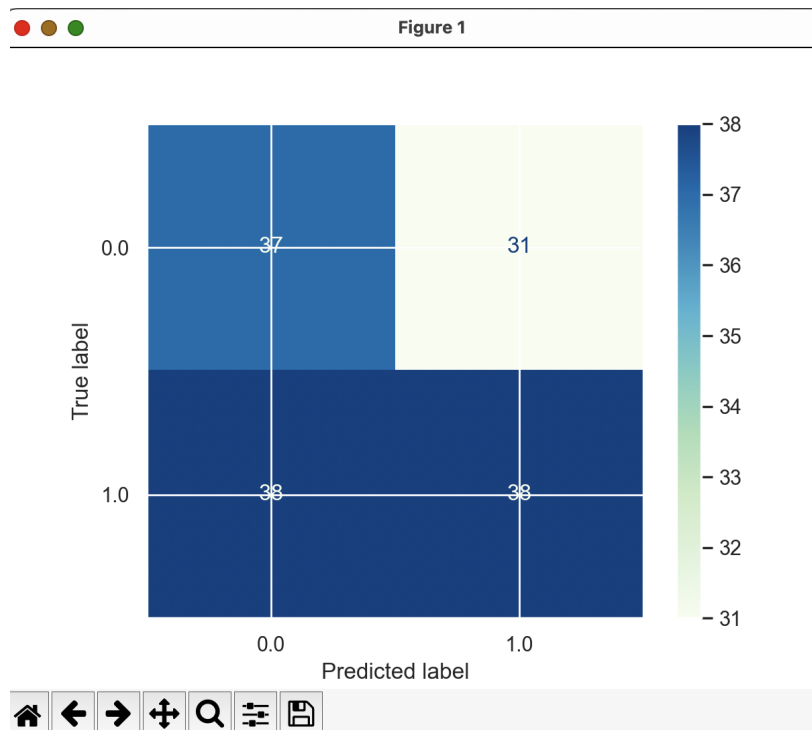
Fig. (7) Feature scores after under-sampling

Now, we'll run this new set of shuffled\_from\_balanced\_df dataset, and examine the accuracy, F-1 score and ROC\_AUC by running Random Forest Classifier. We find that the accuracy is now:

```
>>> mean_accuracy_2 = rf_clf.score(X_test, y_test) * 100
>>> print('Accuracy of X_test (in %): ', mean_accuracy_2)
Accuracy of X_test (in %): 52.083333333333336
```

```
>>> y_pred = rf_clf.predict(X_test)
>>> f1 = f1_score(y_test, y_pred)
>>> print('F1 score: ', f1)
F1 score: 0.5241379310344828
```

The confusion matrix looks as follows:



*Fig. (8) Confusion matrix after under-sampling*

The ROC\_AUC score is:

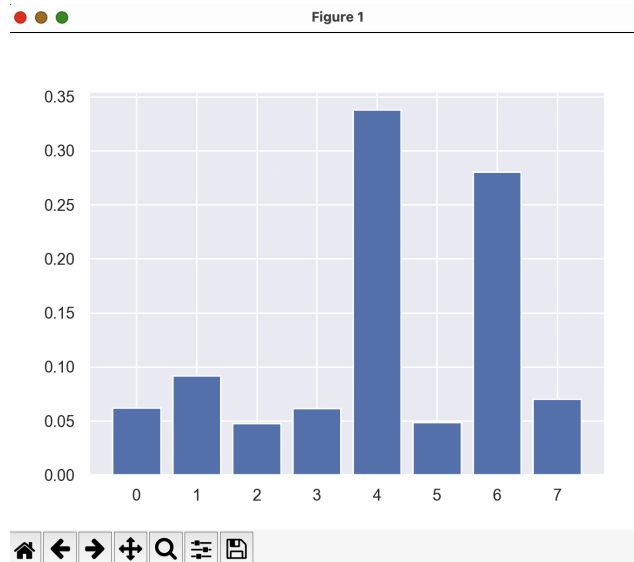
```
>>> roc_auc = roc_auc_score(y_test, rf_clf.predict_proba(X_test)[:, 1])
>>> print('AUC Score =', roc_auc)
AUC Score = 0.5602747678018575
```



The feature importance is:

```
>>> feat_importance = rf_clf.feature_importances_  
>>> print(feat_importance)  
[0.06182428 0.09185716 0.04756361 0.06180319 0.33764397 0.04873419  
 0.28051274 0.07006086]
```

and the plot looks like:



*Fig. (9) Feature importance after under-sampling*

The accuracy has now decreased to 52.08%, the ROC\_AUC score has gotten slightly worse, and False Positives (FP) has increased while the False Negatives (FN) has decreased. The F-1 score in fact yields a higher number of FP. Balancing the classes by under-sampling, and running Random Forest Classifier shows that we have not improved the performance by much.

In under-sampling, the 3 most important features are Feature 4 - time\_spent, Feature 6 - first\_deposit\_amount and Feature 1 - investment\_experience. But all these become moot-points since we see above there is not much of a performance improvement balancing the classes by under-sampling.

Let us now move over to do Over-sampling using SMOTE.

I reloaded the imbalanced DataFrame, model\_df for over-sampling:

```
x = model_df.drop(columns = ['user_id', 'churn'])  
y = model_df['churn']
```

I splitted the data into 80% training data and 20% testing data:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,
random_state = 42)
print(X_train.shape)
print(X_test.shape)
```

```
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 42)
>>> X_train.shape
(4467, 8)
>>> X_test.shape
(1117, 8)
```

After defining the over-sampling method, we transformed the dataset:

```
>>> oversample = SMOTE()
>>> X, y = oversample.fit_resample(X, y)
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 42)
>>> X_train.shape
(8475, 8)
>>> X_test.shape
(2119, 8)
```

Now, we need to run chi-square test to do feature selection and Random Forest Classifier again on this over-sampled dataset. We will then evaluate the feature importance, the F-1 score, and the ROC\_AUC score.

The best features scores obtained:

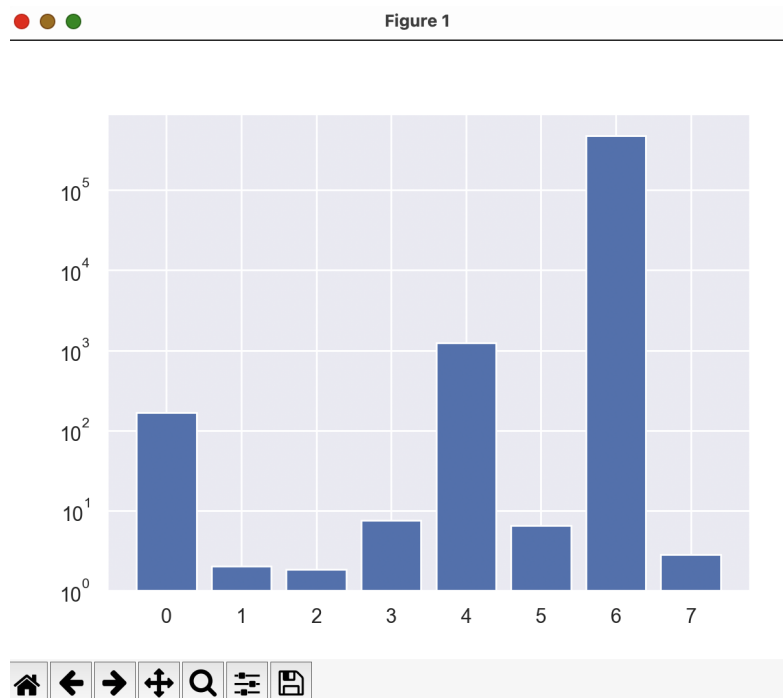
```
>>> best_features_scores = fs_oversampled.scores_
>>> for i in range(len(best_features_scores)):
...     print('Feature %d %f' %(i, best_features_scores[i]))
...
Feature 0 166.833971
Feature 1 2.033710
Feature 2 1.847368
Feature 3 7.486345
```

Feature 4 1237.778860

Feature 5 6.551760

Feature 6 480261.315685

Feature 7 2.796319



*Fig. (10) Features scores after over-sampling*

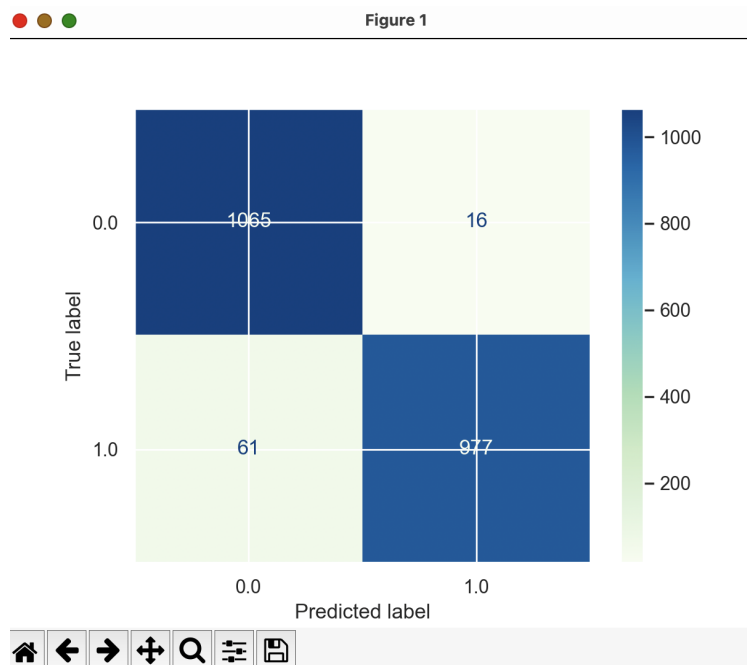
Running the Random Forest Classifier on this over-sampled dataset yields:

```
>>> rf_clf = RandomForestClassifier(n_estimators = 300)
>>> rf_clf = rf_clf.fit(X_train, y_train)
>>> mean_accuracy_3 = rf_clf.score(X_test, y_test) * 100
>>> print('Accuracy of X_test (in %): ', mean_accuracy_3)
Accuracy of X_test (in %): 96.36621047663992
>>>
>>> y_pred = rf_clf.predict(X_test)
>>> f1 = f1_score(y_test, y_pred)
>>> print('F1 score: ', f1)
F1 score: 0.9620876415558838
```

Running confusion matrix yields:

```
>>> confusion_matrix(y_test, y_pred)
array([[1065, 16],
       [ 61, 977]])
```

```
>>> plot_confusion_matrix(rf_clf, X_test, y_test, cmap='GnBu') shows:
```



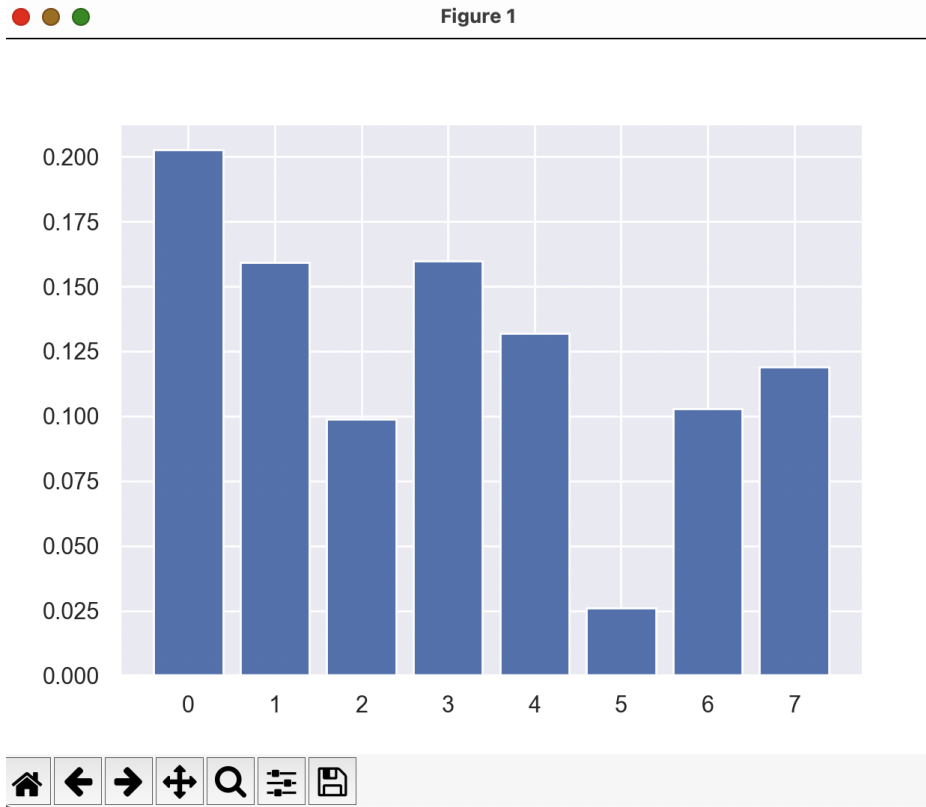
*Fig. (11) Confusion matrix after over-sampling*

and the ROC\_AUC score is:

```
>>> roc_auc = roc_auc_score(y_test, rf_clf.predict_proba(X_test)[:, 1])
>>> print('AUC Score =', roc_auc)
AUC Score = 0.9869670379421038
```

Finally, we check the feature importance of the Random Forest Classifier on this over-sampled dataset:

```
>>> importance = rf_clf.feature_importances_
>>> print(importance)
[0.20256228 0.15909356 0.09882322 0.15967465 0.13189048 0.02609289
 0.10295232 0.11891059]
```



*Fig. (12) Feature importance after over-sampling*

When we tabulate all the results, we would see:

	Original dataset	Under-sampled dataset	Over-sampled dataset
Accuracy	94.18 %	52.08%	96.37%
F-1 score	0.0	0.5241	0.9620
ROC_AUC score	0.5859	0.5603	0.9869

*Fig. (11) Accuracy, f-1 score, ROC\_AUC score vs datasets*

Clearly, the over-sampled dataset yields the greatest accuracy. The False positives (FP) has decreased from 31 (in the under-sampled dataset) to 16. F-1 score takes into account precision and recall. The closer the F-1 score is to 1, the better the model. We see that the over-sampled dataset yields the highest F-1 score out of the three in Fig. (11) above. This means the precision and recall is the highest in the over-sampled dataset. For the ROC\_AUC score, the higher it is, the better is the model's performance at distinguishing between the positive (churned users) and the negative (non-churned users) classes.

From the original features\_data.csv, the feature label names are as follows:

Feature 0 - risk\_tolerance

Feature 1 - investment\_experience

Feature 2 - liquidity\_needs

Feature 3 - platform

Feature 4 - time\_spent

Feature 5 - instrument\_type\_first\_traded

Feature 6 - first\_deposit\_amount

Feature 7 - time\_horizon

Feature 8 - user\_id

From Fig. (), we see that using Random Forest Classifier on the over-sampled dataset, the 3 most important features to predict if a customer will churn are: Feature 0 - risk\_tolerance, Feature 3 - platform, Feature 1 - investment\_experience. Also, Feature 4 - time\_spent is important in this case as well.