National College of Ireland

Configuration Manual

MSc Research Project
Data Analytics

Thi Thanh Thuy Nguyen
Student ID: 22107720

School of Computing
National College of Ireland

Supervisor: Jorge Basilio

National College of Ireland
Project Submission Sheet
School of Computing

National College of Ireland

Student Name:	Thi Thanh Thuy Nguyen
Student ID:	22107720
Programme:	Data Analytics
Year:	2018
Module:	MSc Research Project
Supervisor:	Jorge Basilio
Submission Due Date:	20/12/2018
Project Title:	Configuration Manual
Word Count:	XXX
Page Count:	31

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	
Date:	30th January 2024

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Thi Thanh Thuy Nguyen
22107720

1 Introduction

This project has multiple phases: Business Understanding and Data Collection is the first, then Data Preparation, Data Exploration, Model Development, Model Evaluation, and Model Deployment. Model development, data handling, and analysis are tasks that are necessary for a good project outcome. Google Colab notebook will be used as the main tool throughout the entire process stage in order to fulfill the project's objectives. Every step of the process will include the use of Python libraries, such as 'numpy' and 'pandas' for data preprocessing, 'matplotlib' and 'seaborn' for data visualization, 'sklearn' for model creation, valuation, and optimization, and 'shap' for model interpretation.

2 Environment

2.1 Hardware Requirements

Depending on the volume of data and the type of data to be processed, the required configuration may be different. In this project, the tabular data processed is about 3.7Mb, the key configuration is shown as Table 1. In this study, the software used is mainly Google with Google Colab.

Table 1: System Configuration

Processor	2.7 GHz Dual-Core Intel Core i5
Memory	8 GB 1867 MHz DDR3
System type	64-bit operating system, x64-based processor

2.2 Software Requirements

In this study, the key software used is Google to access Google Colab. Google Colab serves as an outstanding platform for writing and executing arbitrary python code through the browser, it offer free GPU & CPU computational resources with some specification as Table 2 below:

Table 2: Colab Specification

CPU	Intel(R) Xeon(R) CPU @ 2.20GHz
Memory	12 GB GDDR5 VRAM
Runtimes	Up to 12 hours

3 Data Collection

The purpose of the dataset is to assist the bank in forecasting which customers are most likely to leave based on account activity and demographic data gathered from Kaggle (Kaggle; 2019).

The following process flow for completing data mining projects adheres to the CRISP-DM (Cross Industry Standard Process for Data Mining) Schröer et al. (2021), which will be tailored to the specific needs of the bank.

4 Data Preparation

4.1 Importing Libraries

The Figure 1 displays python libraries need to be installed to Colab environment including catboost, xgboost, tensorlow, scikreras and keras. More specifically, catboost and xgboost are for catboost and xgboost classifier development, tensorlow, scikreras and keras are for building and optimize the ANN.

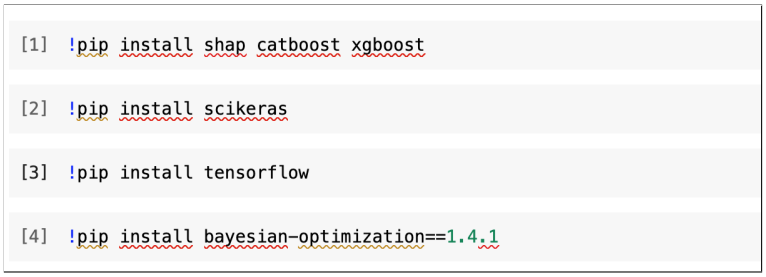

Figure 1: Installed libraries

The library needs to be available before starting the project illustrated in Fugure 2. In addition to libraries used for model development, libraries supporting data preprocessing, visualization, data exploratory and model evaluation were also called.

Figure 2: Imported libraries

4.2 Data import

The code to reading the dataset and data shape is shown in Figure 3. The output displayed that the input data has 23,382 rows and 21 columns.

```
df = pd.read_csv('churn_prediction.csv', delimiter=',')
```

```
df.head()
```

	customer_id	vintage	age	gender	dependents	occupation	city	customer_nw_category	branch_code
0	1	2101	66	Male	0.0	self_employed	187.0	2	755
1	2	2348	35	Male	0.0	self_employed	NaN	2	3214
2	4	2194	31	Male	0.0	salaried	146.0	2	41
3	5	2329	90	NaN	NaN	self_employed	1020.0	2	582
4	6	1579	42	Male	2.0	self_employed	1494.0	3	388

5 rows × 21 columns

```
df.shape
```

```
(28382, 21)
```

Figure 3: Import dataset and display first five rows

4.3 Data pre-processing

Duplication check is shown in the Figure 4. Duplication check should be conducted so that other stages of pre-processing such as missing value imputation will not impacted by duplicated values.

```
# Checking if there are duplicates in the dataset
print("Number of duplicated records in dataset: ", df.duplicated().sum())
```

```
Number of duplicated records in dataset:  0
```

Figure 4: Check duplidation

Figure 5 displays the code to count the number of unique value of all columns. This stage is to support for categorize feature into numerical and categorical properly.

```
# Count Unique value of all columns
df.nunique()
```

```
customer_id                     28382
vintage                          1459
age                                90
gender                              2
dependents                         15
occupation                          5
city                             1604
customer_nw_category                3
branch_code                      3185
current_balance                 27903
previous_month_end_balance      27922
average_monthly_balance_prevQ   27801
average_monthly_balance_prevQ2  27940
current_month_credit            10411
previous_month_credit           10711
current_month_debit             13704
previous_month_debit            14010
current_month_balance           27944
previous_month_balance          27913
churn                               2
last_transaction                  361
dtype: int64
```

Figure 5: Check duplidation

Some columns unnecessary or inadequate information would be removed out of the dataset, illustrated in Figure 6. Specifically, the 'customer_id' is the unique key of the data, the 'last_transaction' actually displayed the date, but the dataset's author did not provide any more information about this column so that other feature could be acquired by using it. Similarly, the 'city' and 'branch_code' have also been removed because of inadequate information.

```
df.drop(columns= ['customer_id',"last_transaction","city","branch_code"], inplace=True)
df.head()
```

	vintage	age	gender	dependents	occupation	customer_nw_category	current_balance	previous_m
0	2101	66	Male	0.0	self_employed	2	1458.71	
1	2348	35	Male	0.0	self_employed	2	5390.37	
2	2194	31	Male	0.0	salaried	2	3913.16	
3	2329	90	NaN	NaN	self_employed	2	2291.91	
4	1579	42	Male	2.0	self_employed	3	927.72	

Figure 6: Dropping unnecessary columns

Figure 7 displays some basic information of continuous variables such as mean, max and median to knowledge about the range value of every single feature before conducting next stages.

```
# Describe
df.describe()
```

	vintage	age	dependents	customer_nw_category	current_balance	previous_month_end_l
count	28382.000000	28382.000000	25919.000000	28382.000000	2.838200e+04	2.838
mean	2091.144105	48.208336	0.347236	2.225530	7.380552e+03	7.495
std	272.676775	17.807163	0.997661	0.660443	4.259871e+04	4.252
min	73.000000	1.000000	0.000000	1.000000	-5.503960e+03	-3.149
25%	1958.000000	36.000000	0.000000	2.000000	1.784470e+03	1.906
50%	2154.000000	46.000000	0.000000	2.000000	3.281255e+03	3.379
75%	2292.000000	60.000000	0.000000	3.000000	6.635820e+03	6.656
max	2476.000000	90.000000	52.000000	3.000000	5.905904e+06	5.740

Figure 7: Descriptive analysis

Figure 8 is illustrating the stage of categorizing features based on the number of unique values of every single features. In this project, the features with over 5 unique values would be defined as numeric variables, the others are categorical variables.

```
# Define numeric variables and Categorical Variables
def return_col_match(expr, column_list):

    select_col=[]
    for item in column_list:
        z = re.match(expr, item)

    if z:
        select_col.append(item)
    return select_col

numeric_var = []
cate_var = []

Vartype_threshold = 5

for cols in df.columns.values:
    if (df[cols].nunique() > Vartype_threshold):
        numeric_var.append(cols)
    else:
        cate_var.append(cols)

select_col=[]
select_col.extend(return_col_match('[Tt]+(otal)+',cate_var))
select_col.extend(return_col_match('^[\w]+(_min_)+',cate_var))
select_col.extend(return_col_match('^[\w]+(_max_)+',cate_var))
select_col.extend(return_col_match('^[\w]+(delt_)+',cate_var))
numeric_var.extend(select_col)
[cate_var.remove(i) for i in select_col]

print('Numerical variables:           \n{}'. format(numeric_var))
print('Categorical variables:         \n{}' . format(cate_var))
```

```
Numerical variables:
['vintage', 'age', 'dependents', 'current_balance', 'previous_month_end_balance', 'average_monthly_balance_prevQ',
Categorical variables:
['gender', 'occupation', 'customer_nw_category', 'churn']
```

Figure 8: Categorizing features

The outcome of categorizing features is illustrating in the Figure 9. After removing 4 unnecessary columns, the sample data remains 14 numerical variables and 4 categorical

variables.

```
print('The number of numerical variable: ', len(numeric_var))
print('The number of categorical variable: ', len(cate_var))
```

```
The number of numerical variable:  13
The number of categorical variable:  4
```

```
# Count Unique value of numerical variables
df[[c for c in df.columns if c in numeric_var]].nunique().sort_values(ascending=False)
```

```
current_month_balance            27944
average_monthly_balance_prevQ2   27940
previous_month_end_balance       27922
previous_month_balance           27913
current_balance                  27903
average_monthly_balance_prevQ    27801
previous_month_debit             14010
current_month_debit              13704
previous_month_credit            10711
current_month_credit             10411
vintage                           1459
age                                 90
dependents                          15
dtype: int64
```

Figure 9: Categorized output

Figure 10 describe the code to execute which attributes containing missing values and values they had been imputed. In this case, 'gender' and 'occupation' are categorical variables that would be imputed by 'unknow' or 'other' value, meanwhile missed value of 'dependents' would be filled by itself median value.

Handle missing value

```
[22] # Find columns having missing value
     [col for col in df.columns if df[col].isnull().any()]
```

```
     ['gender', 'dependents', 'occupation']
```

Columns of 'gender', 'dependents' and 'occupation' has missing value

```
[23] # Fill NaN values with a specific value, for example,
     df['gender'].fillna(value='other', inplace=True)
     df['dependents'] = df['dependents'].fillna(df['dependents'].median())
     df['occupation'].fillna(value='unknown', inplace=True)
```

Figure 10: Missing value imputation

After imputing values missed, the output in Figure 11 displayed the imputation correctly.

```
# Find unique value of categorical variable
for col in cate_var:
    print(col, df[col].unique())
```

```
gender ['Male' 'other' 'Female']
occupation ['self_employed' 'salaried' 'retired' 'student' 'unknown' 'company']
customer_nw_category [2 3 1]
churn [0 1]
```

Figure 11: Missing value imputation check

The five features include 'balance_difference', 'credit_utilization_ratio', 'debit_credit_ratio', 'balance_change_percentage' and 'avg_monthly_balance_change' have been created and their column name have also added to the numeric variable list, as shown in the Figure 12.

```
# Create a new attribute 'balance_difference'
df['balance_difference'] = abs(df['current_month_balance'] - df['previous_month_balance'])

# Create a new attribute 'credit_utilization_ratio'
df['credit_utilization_ratio'] = df['current_month_credit'] / df['previous_month_end_balance']

# Create a new attribute 'debit_credit_ratio'
df['debit_credit_ratio'] = df['current_month_debit'] / df['current_month_credit']

# Create a new attribute 'balance_change_percentage'
df['balance_change_percentage'] = ((df['current_month_balance'] - df['previous_month_balance']) / df

# Create a new attribute 'avg_monthly_balance_change'
df['avg_monthly_balance_change'] = (df['average_monthly_balance_prevQ'] - df['average_monthly_balance
```

```
for i in ('balance_difference','credit_utilization_ratio','debit_credit_ratio','balance_change_percer
  numeric_var.append(i)
```

Figure 12: Creating new features

4.4 Univariate analysis

The Figure 13 is the code to plot the percentage of churners and non-churners. The output shows that the percentage of churner is 18.53% (5260 churn cases out of 28382) and the percentage of 'non-churner' is 81.47% (23122 NOT churn cases out of 28382).

```
# Extracting the target variable
target = df['churn']

# Displaying the percentage of churners and non-churners
print(f'Percentage of \033[1m"churner"\033[0m: % {round(target.value_counts(normalize=True)[1]*100,2)
({target.value_counts()[1]} churn cases out of {len(df)})\nPercentage of \033[1m"non-churner"\033[0m:

# Plotting the pie chart
plt.figure(figsize=(4, 4))
target.value_counts().plot(kind="pie", autopct='%1.1f%%', explode=[0, 0.1],shadow=True,colors=['red',

# Adding labels and title
plt.title('The proportion of Churners and Non-Churners')
plt.legend(['Non-Churners', 'Churners'], loc='upper right')
plt.show()
```

Figure 13: Plot the proportion of the target variable

Figure 14 displays some basic information of continuous variables under data-frame.

```
df_description = df.describe().loc[['min', '25%', 'mean', '50%',  '75%', 'max']]

# Put everything into a DataFrame
summary_data = pd.DataFrame({
    # 'Count': df_description.loc['count'],
    'Mean': df_description.loc['mean'],
    # 'Std Dev': df_description.loc['std'],
    'Min': df_description.loc['min'],
    '25%': df_description.loc['25%'],
    '50%': df_description.loc['50%'],
    '75%': df_description.loc['75%'],
    'Max': df_description.loc['max']
})

# Displaying the new DataFrame
summary_data_rounded = summary_data.round(1)
print(summary_data_rounded)
```

Figure 14: Displaying the descriptive analysis output under data-frame

Figure 15 illustrates the code to execute distribution of every single feature.

```
# Plot histogram of numerical variables
%matplotlib inline

df.hist(figsize=(60, 60))
plt.show()
```

Figure 15: Plotting distribution of every single feature

Box plots provide a quick visual summary of the variability of values towards the target variable 'churn'. Figure 16 is displaying the executing the importance of customer's age against churn decision.

```
# Boxplot by age
plt.figure(figsize= (6,6))
sns.boxplot(x = 'churn', y = 'age', data = df)
plt.title("Boxplot of churn and customers' age")
plt.show()
```

Figure 16: Executing boxplot of churn and customers' age

Figure 17 describes the code to execute the importance of customer's vintage against churn decision.

```
# Boxplot by customer's vintage
plt.figure(figsize= (6,6))
sns.boxplot(x = 'churn', y = 'vintage', data = df)
plt.title("Boxplot of churn and customer' vintage ")
plt.show()
```

Figure 17: Executing boxplot of churn and customers' vintage

Figure 18 is to displays the difference in the histogram of customer's age between the churn rate among different age groups.

```
def churn_prediction_plot(data, col):
    churn_data = data[data['churn'] == 1][col].dropna()
    non_churn_data = data[data['churn'] == 0][col].dropna()

    # Set up parameters
    sns.set(style="whitegrid")
    plt.figure(figsize=(6, 6))

    # Plot histograms
    sns.histplot(churn_data, kde=True, color='red', label='Churn', stat='density', common_norm=False)
    sns.histplot(non_churn_data, kde=True, color='blue', label='Non-Churn', stat='density', common_nc
    plt.title(f'Histogram of {col} for Churn and Non-Churn')
    plt.xlabel(col)
    plt.ylabel('Density')
    plt.legend()

    plt.show()

churn_prediction_plot(df, 'age')
```

Figure 18: Executing distribution of customer ages between two classes of churn

Figure 19 is to displays the distribution of two classes of churn in customer gender by using bar plot.

```
df.groupby(['gender'])['churn'].agg('sum')
plt.figure(figsize=(5,5))
sns.countplot(data=df, x='gender', hue='churn');
```

Figure 19: Executing distribution of two classes of churn in customer gender

Figure 20 is to illustrate the distribution of two classes of churn in occupation by using bar plot.

```
df.groupby(['occupation'])['churn'].agg('sum')
plt.figure(figsize=(6,6))
sns.countplot(data=df, x='occupation', hue='churn');
```

Figure 20: Executing distribution of churners and non-churners in customer occupation

Figure 21 is executing the bar plot of customer networth segment toward churners and non-chuners.

```
df.groupby(['customer_nw_category'])['churn'].agg('sum')
plt.figure(figsize=(6,6))
sns.countplot(data=df, x='customer_nw_category', hue='churn');
```

Figure 21: Bar plot of customers' networth and churn

4.5 Multivariate analysis

The Pearson correlation analysis to figure out the internal relationship between independent features is displayed in Figure 22.

```
#Correlation check for continuous variables
df.corr()
```

Figure 22: Pearson correlation executing

Figure 23 is to visualize the Pearson correlation analysis.

```
# Correlation check on variables
plt.figure(figsize=(10,5))
plt.title("Pearson corrlation analysis between banking customer's feature")
sns.heatmap(data=df[numeric_var].corr(),annot=False);
```

Figure 23: Pearson correlation matrix executing

4.6 Transformation

Figure 24 is to use One-hot coding to transform categorical features into binary variables.

```
cate_var
```

```
['gender', 'occupation', 'customer_nw_category', 'churn']
```

```
y = pd.get_dummies(df.occupation, prefix='occupation')
df = pd.concat([df, y], axis=1)

y1 = pd.get_dummies(df.gender, prefix='gender')
df = pd.concat([df, y1], axis=1)

y2 = pd.get_dummies(df.customer_nw_category, prefix='cus_nw_cate')
df = pd.concat([df, y2], axis=1)
```

Figure 24: One-hot coding

Dropping original categorical features after using One-hot coding to transform, as shown by Figure 25.

```
df.drop(columns = ['gender','occupation','customer_nw_category'], inplace = True)
```

Figure 25: Dropping original categorical features

Figure 26 is to do sampling by using Min max scaler and split the dataset into training set and testing set with A cutoff ratio of 70% and 30%. There are two set of training data generated, one with sampling and one without sampling. The purpose is to observe

the performance between using and non-using sampling. 'Stratify' is set to make sure the churn rate between training set and testing set equally.

```
X = df.drop(columns = 'churn', axis=1)
y = df['churn']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42
                                                    , stratify=y)

scaler =MinMaxScaler()
X_train_sampling= scaler.fit_transform(X_train)
X_test_sampling = scaler.transform(X_test)
```

Figure 26: Sampling and splitting data into training set and testing set

5 Modelling and Evaluation

5.1 Baseline training

Figure 27 is displaying the function setting up for baseline training. The purpose of baseline training is to get an overview how effective each model works in certain problem as well as the given dataset.

```
def fit_and_score(models, X_train, X_test, y_train, y_test):
    np.random.seed(42)

    model_scores = {}

    for name, model in models.items():
        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)

        accuracy = model.score(X_test, y_test)
        recall = recall_score(y_test, y_pred)
        f1 = f1_score(y_test, y_pred)

        # If the model has a 'predict_proba' method, calculate AUC
        if hasattr(model, 'predict_proba'):
            y_proba = model.predict_proba(X_test)[:, 1]
            auc = roc_auc_score(y_test, y_proba)
        else:
            auc = None

        model_scores[name] = {'Accuracy': accuracy, 'Recall': recall, 'F1 Score': f1, 'AUC': auc}

    model_scores = pd.DataFrame(model_scores).transpose()
    model_scores = model_scores.sort_values('Accuracy', ascending=False)

    return model_scores
```

Figure 27: Baseline training function

Figure 28 is illustrating hyperparameters setting for baseline training of machine learning model, including xgboost, catboost and random forest. Most of the parameters is default.

```
models = {
    'RandomForest': RandomForestClassifier(),
    'CatBoost': CatBoostClassifier(iterations=1000,  # Set the number of boosting iterations
                                   learning_rate=0.1,  # Set the learning rate
                                   depth=6,  # Set the depth of the trees
                                   loss_function='Logloss',  # Specify the loss function for classi
                                   random_seed=42),
    'XGBoost': xgb.XGBClassifier(learning_rate=0.1,  # Set the learning rate
                                 n_estimators=1000,  # Set the number of boosting rounds
                                 max_depth=6,  # Set the maximum depth of the trees
                                 objective='binary:logistic',  # Specify the objective function for
                                 random_state=42)
}
```

Figure 28: Hyperparameting setting for baseline training

Figure 29 displays to execute the baseline training in dataset without sampling, in this case, they are 'X_train' and 'X_test'

```
baseline_model_scores = fit_and_score(models, X_train, X_test, y_train, y_test)
```

Figure 29: Execute the baseline training

Figure 30 describe the result of baseline training base on measurements.

```
(baseline_model_scores*100).round(4)
```

	Accuracy	Recall	F1 Score	AUC
CatBoost	86.5766	48.0355	57.0139	83.6402
RandomForest	86.5297	44.2966	54.9312	84.7317
XGBoost	86.4474	47.0849	56.2879	82.4476

Figure 30: Baseline training results

Figure 31 compares visually the baseline training output by bar chart.

```
plt.figure(figsize=(10,5))
sns.barplot(data=baseline_model_scores.sort_values('F1 Score').T)
plt.title('Baseline Model Precision Score')
plt.xticks(rotation=90);
```

Figure 31: Visualize baseline training output

Figure 33 displays the code to execute the baseline training in dataset with applying sampling.

```
baseline_model_scores_sampling = fit_and_score(models, X_train_sampling, X_test_sampling, y_train, y_
```

Figure 32: Execute the baseline training

Figure 33 describe the result of baseline training to dataset with using sampling in terms of measurements.

```
(baseline_model_scores_sampling*100).round(4)
```

	Accuracy	Recall	F1 Score	AUC
XGBoost	86.6706	48.4157	57.3789	82.5123
RandomForest	86.4592	44.6134	54.9785	84.5287
CatBoost	86.2948	47.3384	56.1443	83.0610

Figure 33: Execute the baseline training output

5.2 Hyperparameter tuning

Figure 34, 35 and 36 are setting the hyperparameter tuning process to maximise the f1 score for each classifier. The expected output is the set of hyperparameter having the highest F1 score. Random Search CV has been using in this study.

```
def randomsearch_cv_scores(models, params, X_train, X_test, y_train, y_test):
    np.random.seed(42)

    model_rs_scores = {}
    model_rs_best_param = {}

    for name, model in models.items():
        rs_model = RandomizedSearchCV(model,
                                      param_distributions=params[name],
                                       scoring='f1',
                                       cv=5)
        rs_model.fit(X_train,y_train)
        model_rs_scores[name] = rs_model.score(X_test,y_test)
        model_rs_best_param[name] = rs_model.best_params_

    return model_rs_scores, model_rs_best_param
```

Figure 34: Setting the measurement

```
models = {
    'RandomForest': RandomForestClassifier(),
    'CatBoost': CatBoostClassifier(),
    'XGBoost': xgb.XGBClassifier()
}
```

Figure 35: Setting the classifier

```
params = {'XGBoost':{'learning_rate': [0.001, 0.01, 0.1, 0.2],
                     'n_estimators': [50, 100, 200, 500],
                     'max_depth': [3, 6, 9],
                     'objective': ['binary:logistic'],
                     'eval_metric': ['logloss']
                    },
          'CatBoost':{'iterations': [50, 100, 200, 500, 1000],
                      'learning_rate': [0.0001, 0.001, 0.01, 0.1],
                      'depth': [6, 8, 10],
                      'loss_function': ['Logloss', 'CrossEntropy']
                     },
          'RandomForest':{'n_estimators': [50, 100, 500, 1000],
                          'max_depth': [6, 8, 10],
                          'max_features': [2, 3, 4, 5],
                          'min_samples_split': [2, 5, 10],
                          'min_samples_leaf': [1, 2, 4]
                         }
          }
```

Figure 36: Setting hyperparameter for tuning

Executing to get the best set of hyperparameter, as shown in Figure 37. The output of this stage is going to used for model training.

```
model_rs_scores_1, model_rs_best_param_1 = randomsearch_cv_scores(models, params, X_train, X_test, y_
```

Figure 37: Executing the score

The highest F1 score and the best set of hyperparameter according to each classifier are shown in Figure 38 and 39.

```
model_rs_scores_1
```

```
{'RandomForest': 0.5309734513274337,
 'CatBoost': 0.5556849049282111,
 'XGBoost': 0.5688003066308931}
```

Figure 38: Best score of each model

Output from Figure 39 is going to be used in the model training stage.

```
# Best params output
model_rs_best_param_1
```

```
{'RandomForest': {'n_estimators': 500,
  'min_samples_split': 5,
  'min_samples_leaf': 2,
  'max_features': 5,
  'max_depth': 10},
 'CatBoost': {'loss_function': 'CrossEntropy',
  'learning_rate': 0.01,
  'iterations': 1000,
  'depth': 8},
 'XGBoost': {'objective': 'binary:logistic',
  'n_estimators': 500,
  'max_depth': 6,
  'learning_rate': 0.01,
  'eval_metric': 'logloss'}}
```

Figure 39: Executing the best set of hyperparameter

The similar to Figure 40, 41 and 42 are executing the hyperparameter to the dataset with sampling, the highest F1 score and the best set of hyperparameter for each classifier.

```
model_rs_scores_2, model_rs_best_param_2 = randomsearch_cv_scores(models, params, X_train_sampling, )
```

Figure 40: Executing the hyperparameter tuning process

```
model_rs_scores_2
```

```
{'RandomForest': 0.530628803245436,
 'CatBoost': 0.5601547388781432,
 'XGBoost': 0.5726299694189603}
```

Figure 41: Highest F1 score of each classifier

```
model_rs_best_param_2
```

```
{'RandomForest': {'n_estimators': 500,
  'min_samples_split': 5,
  'min_samples_leaf': 2,
  'max_features': 5,
  'max_depth': 10},
 'CatBoost': {'loss_function': 'CrossEntropy',
  'learning_rate': 0.01,
  'iterations': 1000,
  'depth': 8},
 'XGBoost': {'objective': 'binary:logistic',
  'n_estimators': 500,
  'max_depth': 6,
  'learning_rate': 0.01,
  'eval_metric': 'logloss'}}
```

Figure 42: Executing the best set of hyperparameter

After this step, the hyper-parameter tuning process stated that there is not significant gap between the data with and without sampling. To serce for the purpose of model interpretation, the dataset without sampling would be used as input at next stages.

5.3 Catboost

Figures 43, 44, 45 and 46 display the Catboost training process and its model accuracy in terms of AUC, F1 score, Recall, Precision and Accuracy.

```
# Training Catboost model
model_catboost = CatBoostClassifier(loss_function= 'CrossEntropy',
                            learning_rate = 0.01,
                            iterations = 1000,
                            depth = 8,
                                    random_seed=42)

model_catboost.fit(X_train,y_train)
```

Figure 43: Catboost classifier training

```
y_preds = model_catboost.predict(X_test)
print(classification_report(y_test,y_preds))
```

```
              precision    recall  f1-score   support

           0       0.89      0.96      0.92      6937
           1       0.72      0.46      0.56      1578

    accuracy                           0.87      8515
   macro avg       0.80      0.71      0.74      8515
weighted avg       0.86      0.87      0.85      8515
```

Figure 44: Model accuracy on testing set

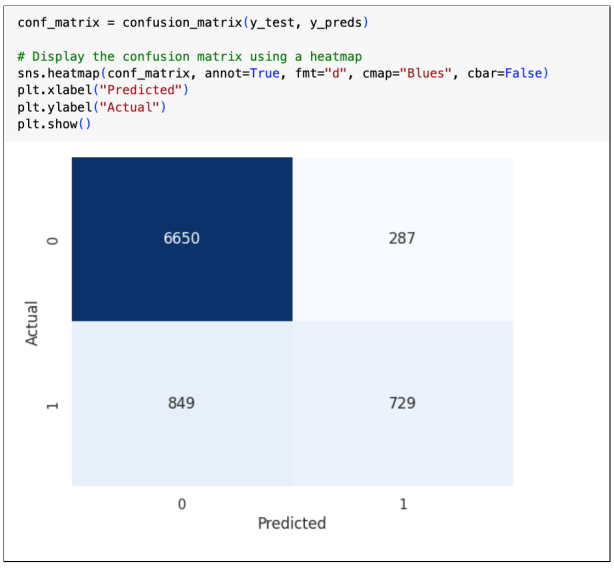

Figure 45: Confusion matrix

```
from sklearn.metrics import RocCurveDisplay
y_train_pred_prob = model_catboost.predict_proba(X_train)[:, 1]

# Compute ROC curve and AUC for training data
fpr_train, tpr_train, _ = roc_curve(y_train, y_train_pred_prob)
roc_auc_train = auc(fpr_train, tpr_train)

# Plot ROC curve for training data
roc_display_train = RocCurveDisplay(fpr=fpr_train, tpr=tpr_train, roc_auc=roc_auc_train, estimator_name='CatBoost (Train)')
roc_display_train.plot()

roc_display_test = RocCurveDisplay.from_estimator(model_catboost, X_test, y_test, ax=plt.gca())
roc_display_test.plot()

# Display the plot
plt.show()
```

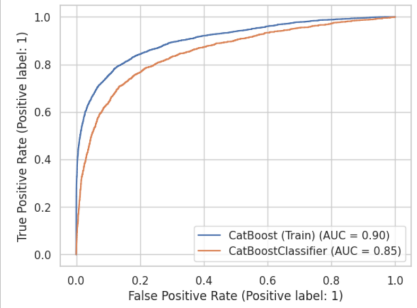

Figure 46: Model's AUC

Figures 47, 48, 49 and 50 describe the implementation of SHAP explainer and its corresponding output for interpreting the Catboost classifier.

```
shap.initjs() # Enable JavaScript visualization
# Create and visualize a SHAP explainer
explainer_catboost = shap.TreeExplainer(model_catboost)
shap_values_catboost = explainer_catboost(X_test, y_test)
shap.summary_plot(shap_values_catboost, X_test)
```

Figure 47: SHAP value explanation

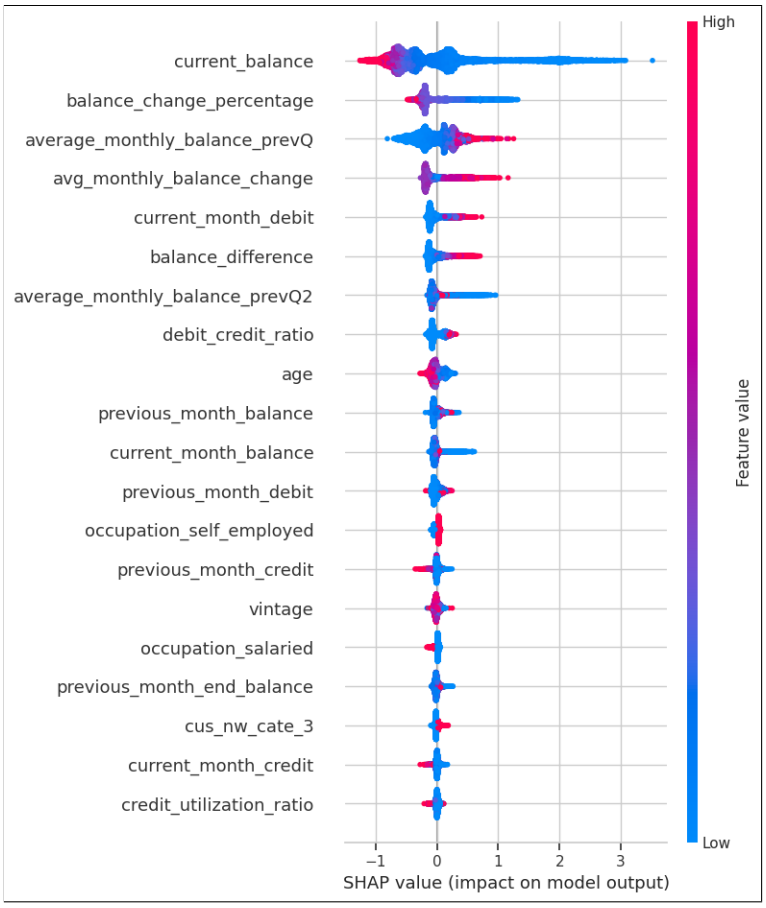

Figure 48: SHAP value explanation

```
shap.initjs()
shap.plots.force(shap_values_catboost[0, ...])
```

Figure 49: Displaying SHAP value explanation output

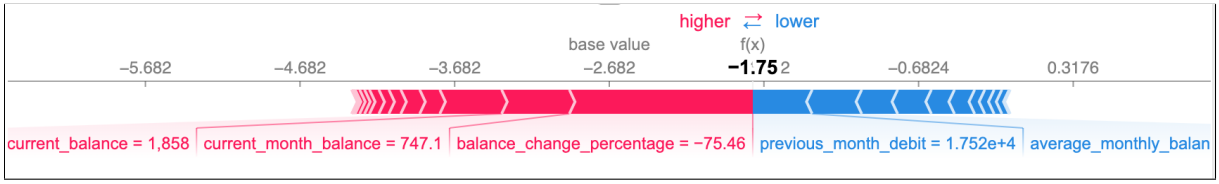

Figure 50: SHAP value explanation for specific observation

5.4 XGBoost

Figures 51, 52, 53 and 54 illustrate the training process of XGBoost classifier and the obtained model accuracy in terms of AUC, F1 score, Recall, Precision and Accuracy.

```python
# XGBoost
model_xgb = xgb.XGBClassifier(objective= 'binary:logistic',
                              learning_rate = 0.01,
                              n_estimators = 500,
                              max_depth = 6,
                              eval_metric ='logloss')
```

Figure 51: XGBoost classifier training

```python
model_xgb.fit(X_train,y_train)
y_preds = model_xgb.predict(X_test)
print(classification_report(y_test,y_preds))
```

```
              precision    recall  f1-score   support

           0       0.89      0.96      0.92      6937
           1       0.72      0.47      0.57      1578

    accuracy                           0.87      8515
   macro avg       0.80      0.71      0.75      8515
weighted avg       0.86      0.87      0.86      8515
```

Figure 52: Model accuracy

```python
conf_matrix = confusion_matrix(y_test, y_preds)

# Display the confusion matrix using a heatmap
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues", cbar=False)
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
```

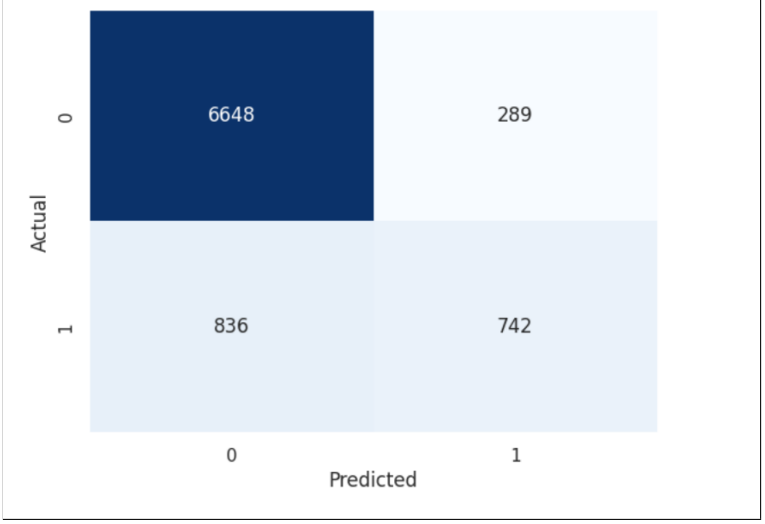

Figure 53: Confusion matrix

```
y_train_pred_prob = model_xgb.predict_proba(X_train)[:, 1]

# Compute ROC curve and AUC for training data
fpr_train, tpr_train, _ = roc_curve(y_train, y_train_pred_prob)
roc_auc_train = auc(fpr_train, tpr_train)

# Plot ROC curve for training data
roc_display_train = RocCurveDisplay(fpr=fpr_train, tpr=tpr_train, roc_auc=roc_auc_train, estimator_name='XGB (Train)')
roc_display_train.plot()

roc_display_test = RocCurveDisplay.from_estimator(model_xgb, X_test, y_test, ax=plt.gca())
roc_display_test.plot()

# Display the plot
plt.show()
```

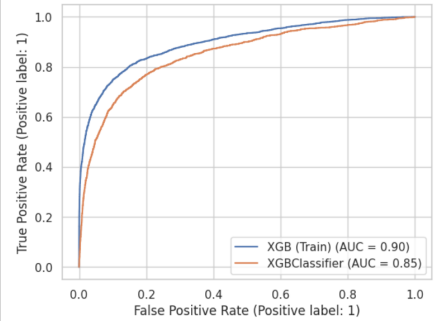

Figure 54: Model's AUC

Figures 55, 56, 57 and 58 describe the execution of SHAP value explainer and its outcome for interpreting XGBoost classifier.

```
shap.initjs() # Enable JavaScript visualization in notebook
# Create a SHAP explainer
explainer_xgb = shap.TreeExplainer(model_xgb)
shap_values_xgb = explainer_xgb(X_test, y_test)
# Visualize summary plot
shap.summary_plot(shap_values_xgb, X_test)
```

Figure 55: SHAP value explanation

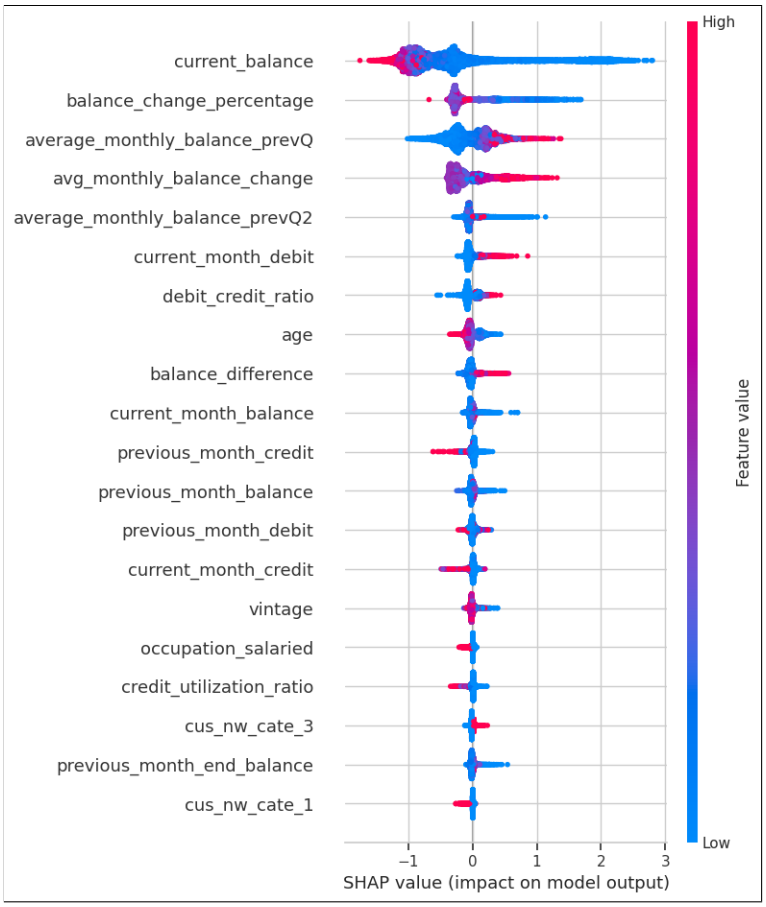

Figure 56: SHAP value explanation

```
shap.initjs()
shap.plots.force(shap_values_xgb[5, ...])
```

Figure 57: SHAP value explanation

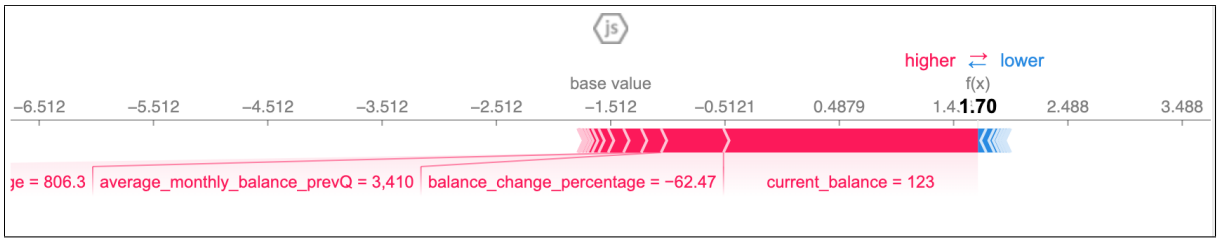

Figure 58: SHAP value explanation for specific observation

5.5 Random Forest

The training procedure of the Random Forest classifier and the obtained model accuracy in terms of AUC, F1 score, Recall, Precision, and Accuracy are shown in Figures 59, 60, 61 and 62.

```
# Random Forest
model_rf = RandomForestClassifier(n_estimators = 500,
                                  min_samples_split = 5,
                                  min_samples_leaf = 2,
                                  max_features = 5,
                                  max_depth = 10)
```

Figure 59: Random Forest classifier training

```
model_rf.fit(X_train,y_train)
y_preds = model_rf.predict(X_test)
print(classification_report(y_test,y_preds))
```

```
              precision    recall  f1-score   support

           0       0.88      0.96      0.92      6937
           1       0.73      0.41      0.53      1578

    accuracy                           0.86      8515
   macro avg       0.80      0.69      0.72      8515
weighted avg       0.85      0.86      0.85      8515
```

Figure 60: Random Forest classifier model accuracy

```
conf_matrix = confusion_matrix(y_test, y_preds)

# Display the confusion matrix using a heatmap
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues", cbar=False)
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
```

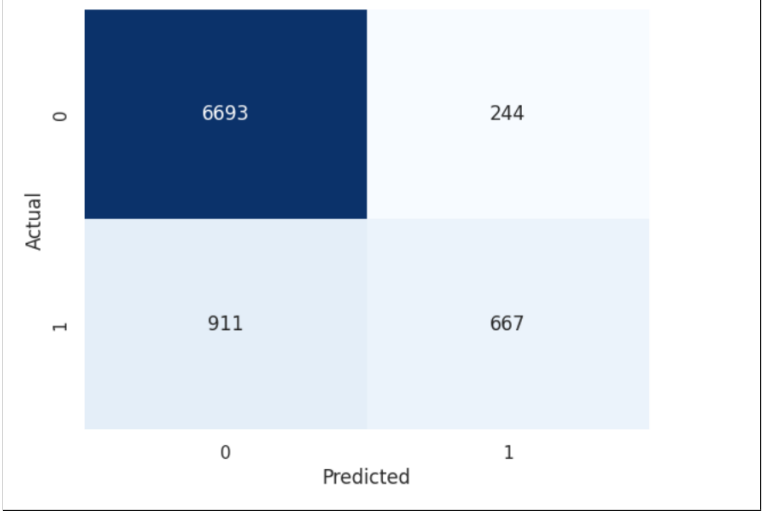

Figure 61: Confusion matrix

```
# Get predicted probabilities for the training data
y_train_pred_prob = model_rf.predict_proba(X_train)[:, 1]

# Compute ROC curve and AUC for training data
fpr_train, tpr_train, _ = roc_curve(y_train, y_train_pred_prob)
roc_auc_train = auc(fpr_train, tpr_train)

# Plot ROC curve for training data
roc_display_train = RocCurveDisplay(fpr=fpr_train, tpr=tpr_train, roc_auc=roc_auc_train, estimat
roc_display_train.plot()

roc_display_test = RocCurveDisplay.from_estimator(model_rf, X_test, y_test, ax=plt.gca())
roc_display_test.plot()

# Display the plot
plt.show()
```

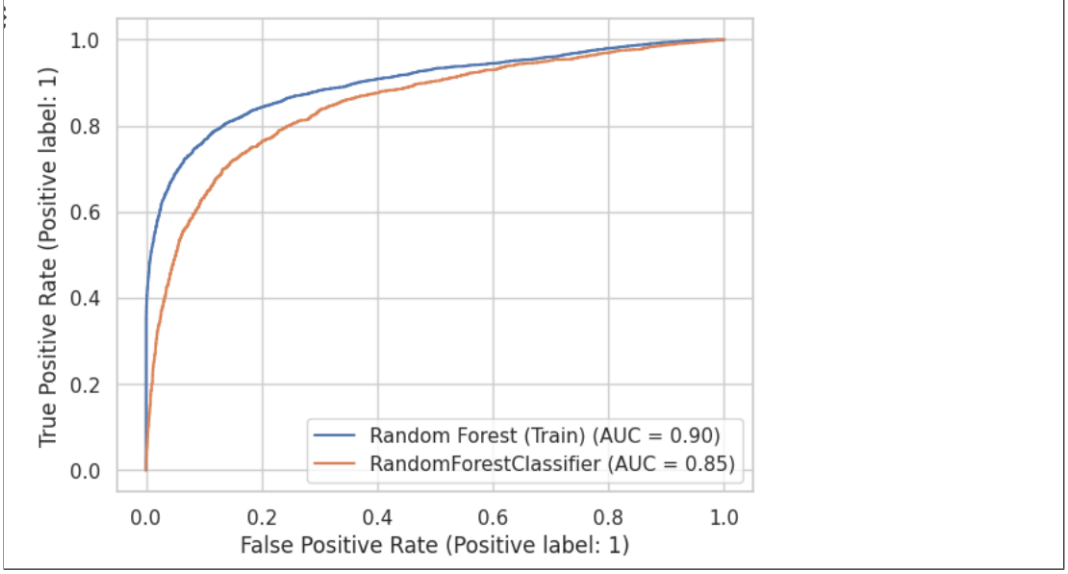

Figure 62: Model's AUC

Figures 63, 64, 65 and 66 describe the execution of SHAP value explainer and its outcome for interpreting Random Forest classifier.

```
shap.initjs() # Enable JavaScript visualization in notebook

# Create a SHAP explainer
explainer_rf = shap.TreeExplainer(model_rf)

# Generate SHAP values using RF
shap_values_rf = explainer_rf.shap_values(X_test)
shap.summary_plot(shap_values_rf, X_test)
```

Figure 63: SHAP value explanation

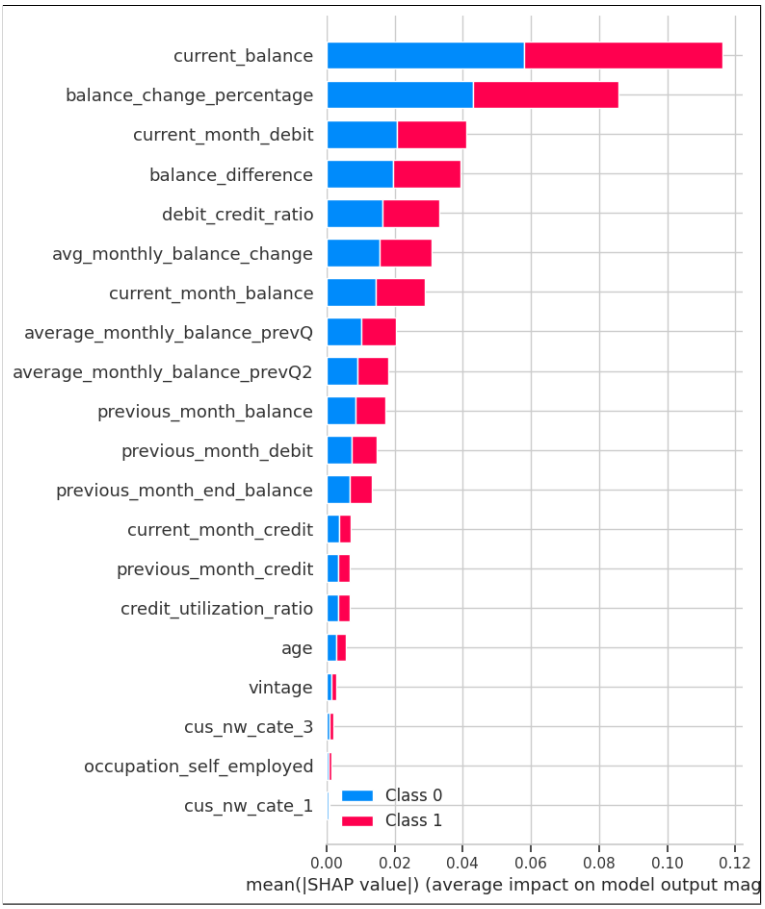

Figure 64: SHAP value explanation

```
# Shap values
choosen_instance = X_test.loc[[1]]
shap_values = explainer_rf.shap_values(choosen_instance)
shap.initjs()
shap.force_plot(explainer_rf.expected_value[1], shap_values[1], choosen_instance)
```

Figure 65: SHAP value explanation

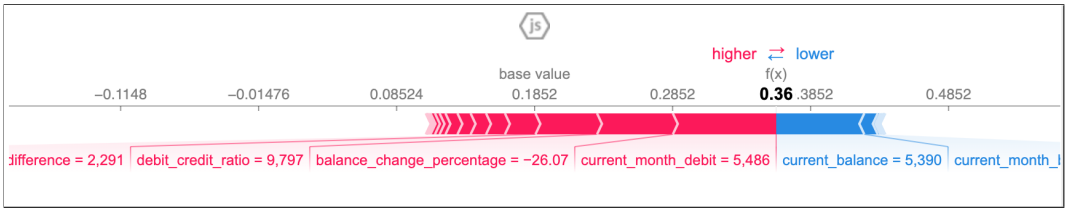

Figure 66: SHAP value explanation for specific observation

5.6 ANN

Figure 67 illustrate the function defined for Bayesian optimization, the function is to maximize the performance based on input parameter. Input parameters illustrated in Figure 67 and the outcome shown in Figure 69.

```python
# Define function for Bayesian optimization
score_acc = make_scorer(accuracy_score)
# opt = Adam(learning_rate=0.001)

def nn_cl_bo(learning_rate, epochs ):
  # neurons = round(neurons)
  batch_size = 32
  epochs = round(epochs)
  def nn_cl_fun():
    opt = Adam(learning_rate = learning_rate)
    nn = Sequential()
    nn.add(Dense(30, input_dim=30, activation='relu'))
    nn.add(Dense(30, activation='relu'))
    nn.add(Dense(20, activation='relu'))
    nn.add(Dense(20, activation='relu'))
    nn.add(Dense(10, activation='relu'))
    nn.add(Dense(1, activation='sigmoid'))
    nn.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
    return nn
  es = EarlyStopping(monitor='accuracy', mode='max', verbose=0, patience=20)
  nn = KerasClassifier(build_fn=nn_cl_fun, epochs=epochs, batch_size=batch_size,verbose=0)
  kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=123)
  score = cross_val_score(nn, X_train, y_train, scoring=score_acc, cv=kfold, fit_params={'callbacks':[es]}).mean()
  return score
```

Figure 67: Define function for Bayesian optimization

Parameters in terms of epochs and learning rate would be tuned as in Figure 67. Because the limitation of computational resources, the chosen number of epochs has been ranged from 100 to 200.

```python
# Set paramaters
params_nn ={'learning_rate':(0.01, 0.1),
    'epochs':(100, 200)
}
# Run Bayesian Optimization
nn_bo = BayesianOptimization(nn_cl_bo, params_nn, random_state=123)
nn_bo.maximize(init_points=6, n_iter=4)
```

```
|   iter    |  target   |  epochs   | learni... |
-------------------------------------------------
| 1         | 0.8147    | 169.6     | 0.03575   |
| 2         | 0.8147    | 122.7     | 0.05962   |
| 3         | 0.8147    | 171.9     | 0.04808   |
| 4         | 0.8147    | 198.1     | 0.07163   |
| 5         | 0.8147    | 148.1     | 0.04529   |
| 6         | 0.8147    | 134.3     | 0.07561   |
| 7         | 0.8147    | 100.0     | 0.08086   |
| 8         | 0.8147    | 200.0     | 0.08011   |
| 9         | 0.8147    | 100.0     | 0.08832   |
| 10        | 0.8147    | 200.0     | 0.02743   |
=================================================
```

Figure 68: Parameters input for Bayesian optimization

```python
params_nn_ = nn_bo.max['params']
params_nn_
```

```
{'epochs': 169.64691855978617, 'learning_rate': 0.035752540145534153}
```

Figure 69: Executing Bayesian optimization

Figure 70 displays the stage of defining the ANN sequential model and model architecture obtain in Figure 71.

```
# Define sequential model
ann = keras.Sequential([
    # input layer
    keras.layers.Dense(30, input_shape=(30,)
    , activation='relu'),
    keras.layers.Dense(30, activation='relu'),
    keras.layers.Dense(20, activation='relu'),
    keras.layers.Dense(20, activation='relu'),
    keras.layers.Dense(10,activation = 'relu'),
    # output layer
    keras.layers.Dense(1, activation='sigmoid')]) # sigmoid for binary output
learning_rate = 0.0001
optimizer = Adam(learning_rate=learning_rate)
# time for compilation of neural net.
ann.compile(optimizer = optimizer,
             loss = 'binary_crossentropy',
             metrics = ['accuracy'])
```

Figure 70: Define ANN sequential model

```
ann.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 30)                930

 dense_1 (Dense)             (None, 30)                930

 dense_2 (Dense)             (None, 20)                620

 dense_3 (Dense)             (None, 20)                420

 dense_4 (Dense)             (None, 10)                210

 dense_5 (Dense)             (None, 1)                 11

=================================================================
Total params: 3121 (12.19 KB)
Trainable params: 3121 (12.19 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Figure 71: Execution of ANN model architecture

Figure ?? and 73 display the execution of the ANN sequential model. The ANN model starts training and evaluating on testing set.

```
# Fit model for training data
ann.fit(X_train,y_train,epochs=700)
```

Figure 72: Define ANN sequential model

```
# evalute the model
ann.evaluate(X_test,y_test)
```

Figure 73: Execution of ANN model architecture

The obtained model is applied to predict churn on testing set as shown in Figure 74.

```
# predict the churn values
y_pred = ann.predict(X_test)
print(y_pred)
# Convert to 1 and 0
y_pred_lis = []
for i in y_pred:
    if i>0.5:
        y_pred_lis.append(1)
    else:
        y_pred_lis.append(0)
print(y_pred_lis)
```

Figure 74: Predicting in the test set

The training procedure of the ANN classifier and the obtained model accuracy in terms of AUC, F1 score, Recall, Precision, and Accuracy are shown in Figures 75, 76 and 77.

```
# print classification_report
print(classification_report(y_test,y_pred_lis))
```

	precision	recall	f1-score	support
0	0.87	0.96	0.91	6937
1	0.68	0.39	0.49	1578
accuracy			0.85	8515
macro avg	0.77	0.67	0.70	8515
weighted avg	0.84	0.85	0.84	8515

Figure 75: Model performance

```
# ploting the confusion metrix plot
conf_mat = tf.math.confusion_matrix(labels=y_test,predictions=y_pred_lis)
plt.figure(figsize = (7,7))
sns.heatmap(conf_mat, annot=True, fmt="d", cmap="Blues", cbar=False)
plt.xlabel('Predicted')
plt.ylabel('Actual')
```

```
Text(53.75, 0.5, 'Actual')
```

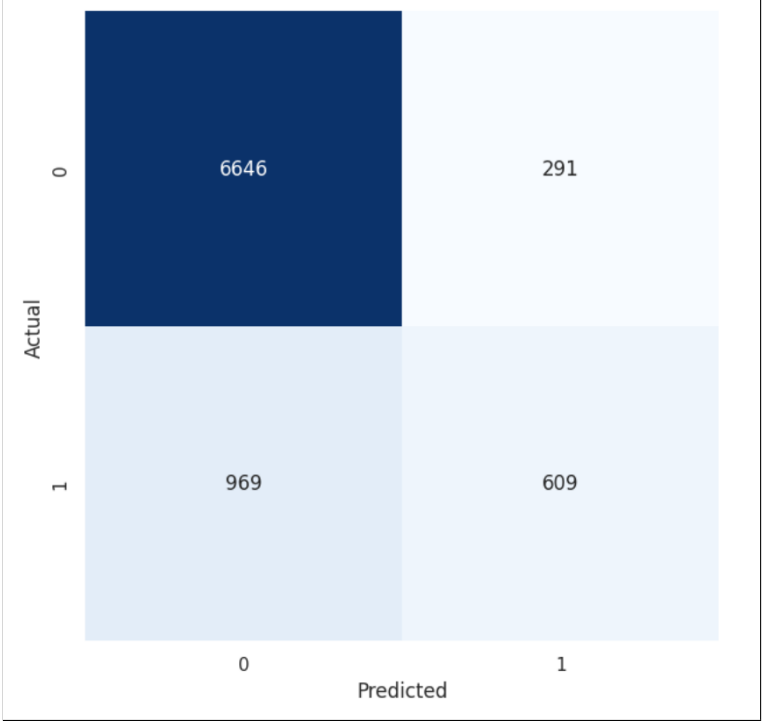

Figure 76: Confustion matrix

```
# Calculate ROC curve for test data
y_score_test = ann.predict(X_test)
fpr_test, tpr_test, _ = roc_curve(y_test, y_score_test)
roc_auc_test = auc(fpr_test, tpr_test)

# Calculate ROC curve for training data
y_score_train = ann.predict(X_train)
fpr_train, tpr_train, _ = roc_curve(y_train, y_score_train)
roc_auc_train = auc(fpr_train, tpr_train)

# Plot ROC curves
plt.figure(figsize=(4, 4))
plt.plot(fpr_test, tpr_test, color='darkorange', lw=2, label='Test ROC curve (area = {:.2f})'.format(roc_auc_test))
plt.plot(fpr_train, tpr_train, color='green', lw=2, label='Train ROC curve (area = {:.2f})'.format(roc_auc_train))
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
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 (ROC) Curve')
plt.legend(loc='lower right')
plt.show()
```

```
267/267 [==============================] - 1s 3ms/step
621/621 [==============================] - 2s 3ms/step
```

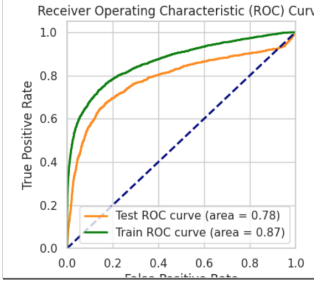

Figure 77: Model AUC

Figures 78 and 79 describe the execution of SHAP value explainer and its outcome

for interpreting Random Forest classifier.

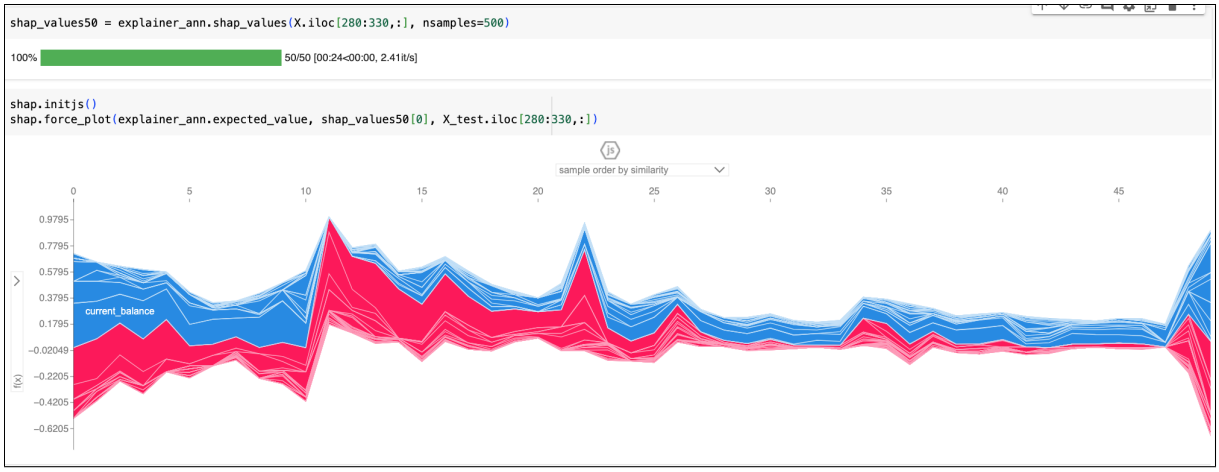

Figure 78: Sample prediction's explanation for ANN using SHAP in banking churn prediction

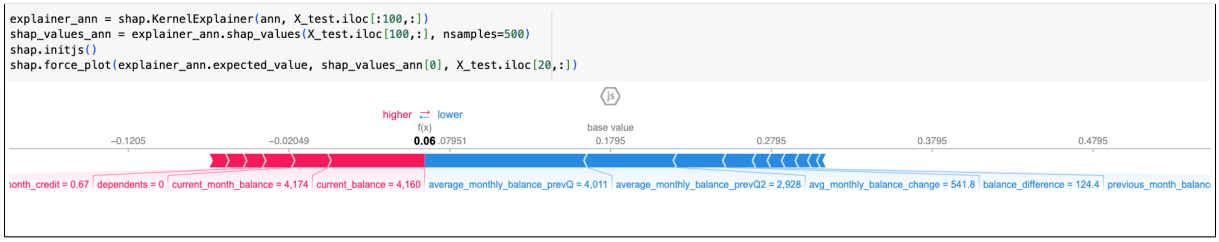

Figure 79: Sample prediction's explanation for ANN using SHAP in banking churn prediction

6 Deployment

Obtained model is saved as described in Figure 80. The obtained then would be used for deployment in personalized customer retention efforts.

```
# Save RF
# save the model to disk
filename_rf = 'RandomeForest.sav'
pickle.dump(model_rf, open(filename_rf, 'wb'))

filename_ann = 'ANN.sav'
pickle.dump(ann, open(filename_ann, 'wb'))

filename_catboost = 'catboost.sav'
pickle.dump(model_catboost, open(filename_catboost, 'wb'))

filename_xgb = 'xgb.sav'
pickle.dump(model_xgb, open(filename_xgb, 'wb'))
```

Figure 80: Saving obtained model

In Figure 81, it is assumed that mock_data is up to date dataset obtained from the bank. The data would be input to the pre-processing step and fed into the saved model (in this case, xgboost classifier) for extracting the niche list, more specially, potential churners as illustrated in Figure 82. Niche list is the customer id with remaked '1'.

```
# Extract randomly 100 samples from X_test as an example
random_seed = 42
random_samples = df.sample(n=100, random_state=random_seed)
mock_data = pd.DataFrame(data=random_samples.values, columns=random_samples.columns)

def pre_process(df):
  df.drop_duplicates(inplace=True)
  df.drop(columns= ['customer_id',"last_transaction","city","branch_code"], inplace=True)

  # Fill NaN values with a specific value
  df['gender'].fillna(value='other', inplace=True)
  df['dependents'] = df['dependents'].fillna(df['dependents'].median())
  df['occupation'].fillna(value='unknown', inplace=True)

  # Create a new attribute 'balance_difference'
  df['balance_difference'] = abs(df['current_month_balance'] - df['previous_month_balance'])
  # Create a new attribute 'credit_utilization_ratio'
  df['credit_utilization_ratio'] = df['current_month_credit'] / df['previous_month_end_balance']
  # Create a new attribute 'debit_credit_ratio'
  df['debit_credit_ratio'] = df['current_month_debit'] / df['current_month_credit']
  # Create a new attribute 'balance_change_percentage'
  df['balance_change_percentage'] = ((df['current_month_balance'] - df['previous_month_balance']) / df['previous_month_balance']) * 100
  # Create a new attribute 'avg_monthly_balance_change'
  df['avg_monthly_balance_change'] = (df['average_monthly_balance_prevQ'] - df['average_monthly_balance_prevQ2']) / 2

  y = pd.get_dummies(df.occupation, prefix='occupation')
  df = pd.concat([df, y], axis=1)

  y1 = pd.get_dummies(df.gender, prefix='gender')
  df = pd.concat([df, y1], axis=1)

  y2 = pd.get_dummies(df.customer_nw_category, prefix='cus_nw_cate')
  df = pd.concat([df, y2], axis=1)

  df.drop(columns = ['gender','occupation','customer_nw_category'], inplace = True)
  return df
```

Figure 81: Conducting pre-processing input data

```
random_seed = 42
mock_data = pre_process(df1)
random_samples = mock_data.sample(n=100, random_state=random_seed)
niche_list = pd.DataFrame(data=random_samples.values, columns=random_samples.columns)

# Convert non-numeric columns to numeric
niche_list = niche_list.apply(pd.to_numeric, errors='coerce')

# Load the model from disk
loaded_model = pickle.load(open(filename_xgb, 'rb'))

# Make predictions on input data
predictions = loaded_model.predict(niche_list)
```

Figure 82: Making prediction

References

Kaggle (2019). Abc bank customer churn dataset, `https://www.kaggle.com/datasets/sainathreddys/banking-churn`.

Schröer, C., Kruse, F. and Gómez, J. M. (2021). A systematic literature review on applying crisp-dm process model, *Procedia Computer Science* **181**: 526–534. CENTERIS 2020 - International Conference on ENTERprise Information Systems / ProjMAN 2020 - International Conference on Project MANagement / HCist 2020 - International Conference on Health and Social Care Information Systems and Technologies 2020, CENTERIS/ProjMAN/HCist 2020.
URL: *https://www.sciencedirect.com/science/article/pii/S1877050921002416*