Classifying the Risk of Default Payments of Online Purchases

Terry Lay

An online trader is interested in knowing whether a customer will eventually pay for the goods they ordered. The training data provided consists of 30,000 purchase details and 44 attributes. The testing data contains 20,000 incoming orders of unknown risk. The objective of this classification problem is to determine whether a purchase order is at a high or low risk of default payment.

The description for the variables in the dataset can be found in the **risk-attributes.txt** file.

Data Exploration and Preprocessing

First, let's check for missing values in the dataset.

```
# Counting the number of missing values
check_missing(risk_train)
```

```
B_BIRTHDATE      2942
Z_CARD_ART      18654
Z_LAST_NAME     14808
TIME_ORDER         20
ANUMMER_02      22147
ANUMMER_03      26802
ANUMMER_04      28668
ANUMMER_05      29459
ANUMMER_06      29794
ANUMMER_07      29905
ANUMMER_08      29966
ANUMMER_09      29993
ANUMMER_10      30000
DATE_LORDER     15856
MAHN_AKT        15856
MAHN_HOECHST    15856
dtype: int64
```

The dataset has many missing values. Some of the values are missing because the values are not applicable to that specific purchase order. For example, new customers would not have previous purchase details and orders made without a card would not have card information. We will analyze these features to see if they provide any information in our risk assessment or if they can be dropped.

First, we notice that if a check is given then there is no data for the last name of the card holder. The number of missing values in Z_CARD_ART also corresponds to the number of payment methods that were checks or debit notes in Z_METHODE.

```
risk_train['Z_METHODE'].value_counts()
```

```
check          14808
credit_card     9796
debit_note      3846
debit_card      1550
Name: Z_METHODE, dtype: int64
```

```
risk_train['Z_CARD_ART'].value_counts()
```

```
?              18654
Eurocard        5096
Visa            3927
debit_card      1550
Amex             773
Name: Z_CARD_ART, dtype: int64
```

Let's replace 'credit_card' in Z_METHODE with the type of card used and drop Z_CARD_ART and Z_LAST_NAME from our dataset.

```
risk_train['Z_METHODE'].value_counts()
```

```
check          14808
Eurocard        5096
Visa            3927
debit_note      3846
debit_card      1550
Amex             773
Name: Z_METHODE, dtype: int64
```

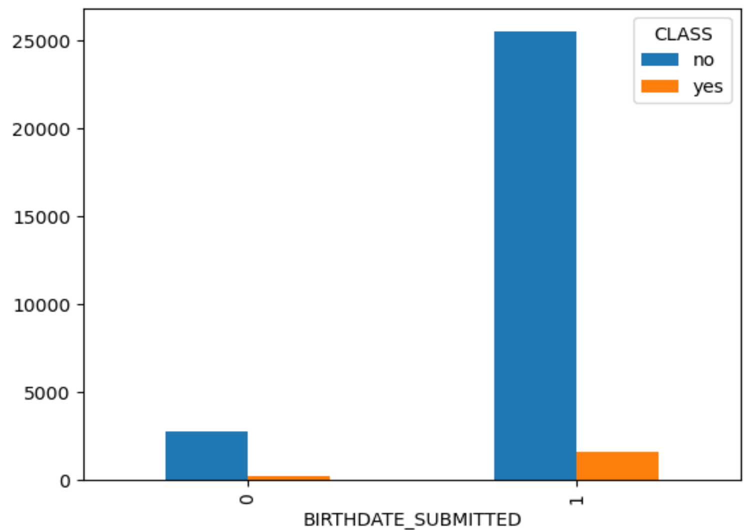
Next, we will drop the ANUMMER columns which indicate the item IDs as the item number is unlikely to influence the target variable.

```
risk_train = risk_train.drop(['ANUMMER_01',
                 'ANUMMER_02',
                 'ANUMMER_03',
                 'ANUMMER_04',
                 'ANUMMER_05',
                 'ANUMMER_06',
                 'ANUMMER_07',
                 'ANUMMER_08',
                 'ANUMMER_09',
                 'ANUMMER_10'], axis=1)
```

Let’s create a column to indicate whether a birthdate was submitted with the order to get an indication of whether B_BIRTHDATE has an effect on the class.

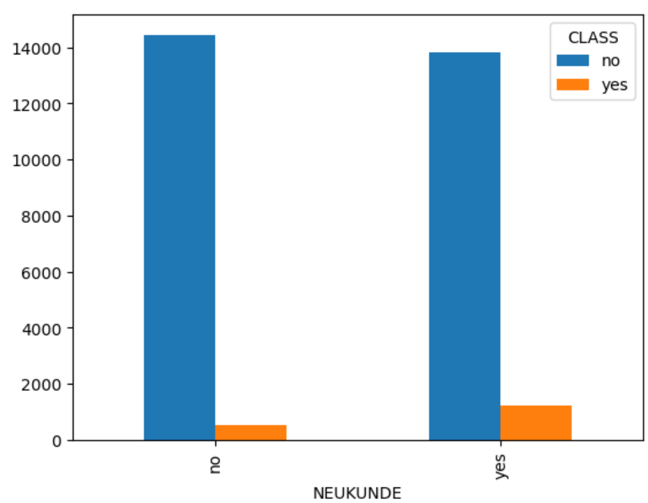
```
risk_train['BIRTHDATE_SUBMITTED'] = risk_train['B_BIRTHDATE'].apply(lambda x: 1 if not (x=='?') else 0)
```

CLASS	no	yes	All
BIRTHDATE_SUBMITTED			
0	2765	177	2942
1	25489	1569	27058
All	28254	1746	30000



We will drop the birthdate columns because it appears that the proportion of people who are high risk is not associated with whether they submitted their birthdate. There are also a large proportion of customers who choose not to provide their birthdate. In this case, removing these rows may result in a large loss of data so we will not explore whether certain ages of people affect their risk level.

In terms of purchase history, we suspect that there may be a higher risk associated among new customers. We will plot a bar chart to see if this is the case.



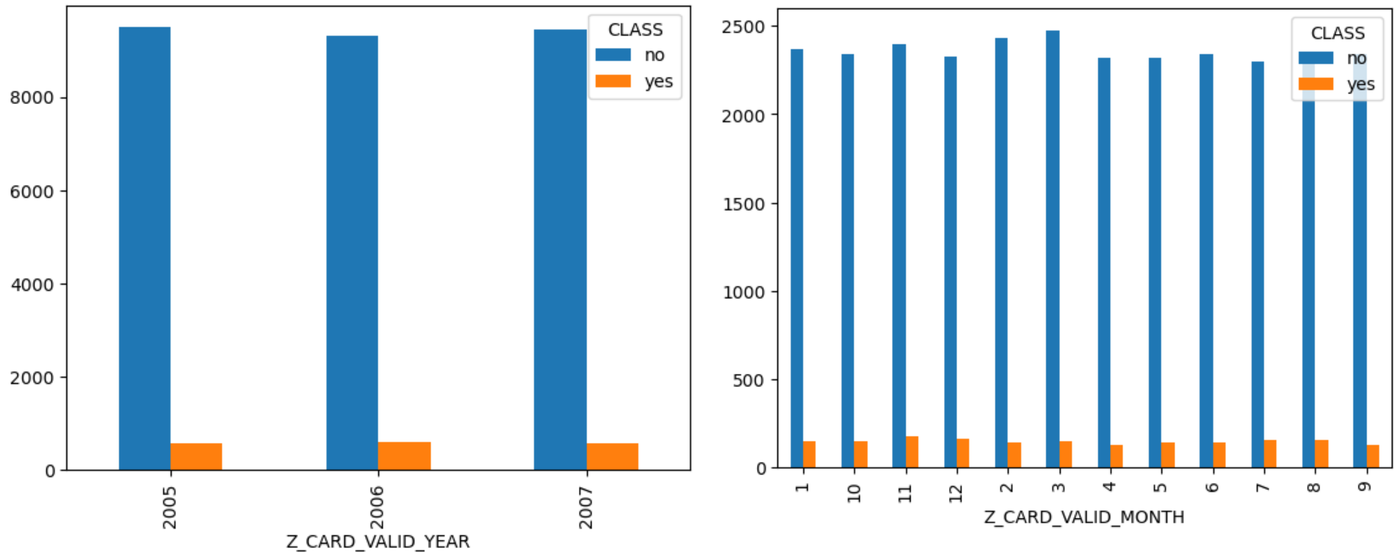
It appears that new customers have a higher proportion of high risk to low risk purchases. We will drop columns related to returning customers since we also have many missing values for them.

The last column with missing values is TIME_ORDER. We will drop this column as well since we do not suspect that the time of order has an association with the risk of default payment.

Let's continue to explore our dataset. The expiration date of the card may not be worth keeping in our model. Let's split the expiration date by year and month to visualize this and confirm whether this is true.

```
risk_train['Z_CARD_VALID_YEAR'] = risk_train['Z_CARD_VALID'].astype(str).str[-4:]
```

```
risk_train['Z_CARD_VALID_MONTH'] = risk_train['Z_CARD_VALID'].astype(int).astype(str)
```



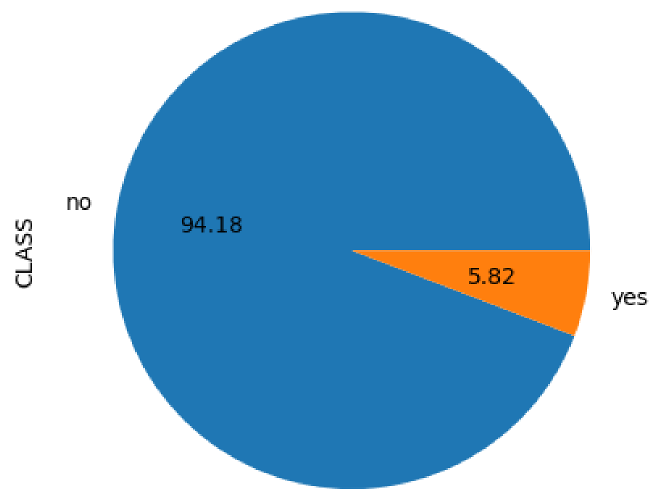
It seems like the class distribution is fairly uniform between the years and months so we will drop this column.

In our dataset, we have two numerical features: VALUE_ORDER and SESSION_TIME. Let's scale them to ensure that they do not dominate the model using MinMaxScaler.

```
scaler = MinMaxScaler()
risk_train[['VALUE_ORDER', 'SESSION_TIME']] = scaler.fit_transform(
    risk_train[['VALUE_ORDER', 'SESSION_TIME']])
```

Before we start model training, let's take a look at the class distribution.

```
risk_train.CLASS.value_counts().plot.pie(autopct="%.2f")
```



There is a large imbalance between our majority and minority class because most of the purchase orders are low risk. We will need to consider weights when we are fitting our models to ensure we are penalizing misclassification in the minority class as this is more costly.

Model Training and Validation

Let's explore various models to fit the data. Based on the previous analysis, we need to consider the imbalance in our class distribution. For simplicity, we will start by using balanced class weights, but we may need to specify our own weights to tune our selected model.

One of the metrics we will use to compare model performances is the misclassification cost. The cost of a false positive is 5, whereas the cost of a false negative is 50. The online trader is more interested in knowing which purchases are high risk, so if our model considers a purchase order that is high risk as low risk, it will be a much bigger problem than if a low risk order is classified as high risk. We will calculate the cost based on the confusion matrix as defined in the function below:

```
def check_cost(cm):
    # Gets the confusion matrix and calculates the misclassification cost
    cost = cm[0][1]*5 + cm[1][0]*50 # False negative is more expensive
    return cost
```

Before we split our data, we need to make sure that we encode our categorical variables so that our models can operate on them.

```
risk_train_dummies = pd.get_dummies(risk_train, drop_first=True)
risk_train_dummies.columns
```

```
Index(['VALUE_ORDER', 'AMOUNT_ORDER', 'SESSION_TIME', 'CLASS_yes',
       'B_EMAIL_yes', 'B_TELEFON_yes', 'FLAG_LRIDENTISCH_yes',
       'FLAG_NEWSLETTER_yes', 'Z_METHODE_Eurocard', 'Z_METHODE_Visa',
       'Z_METHODE_check', 'Z_METHODE_debit_card', 'Z_METHODE_debit_note',
       'WEEKDAY_ORDER_Monday', 'WEEKDAY_ORDER_Saturday',
       'WEEKDAY_ORDER_Sunday', 'WEEKDAY_ORDER_Thursday',
       'WEEKDAY_ORDER_Tuesday', 'WEEKDAY_ORDER_Wednesday', 'CHK_LADR_yes',
       'CHK_RADR_yes', 'CHK_KTO_yes', 'CHK_CARD_yes', 'CHK_COOKIE_yes',
       'CHK_IP_yes', 'FAIL_LPLZ_yes', 'FAIL_LORT_yes', 'FAIL_LPLZORTMATCH_yes',
       'FAIL_RPLZ_yes', 'FAIL_RORT_yes', 'FAIL_RPLZORTMATCH_yes',
       'NEUKUNDE_yes'],
      dtype='object')
```

Now we can split our data. We will perform an 80% to 20% split into training and test sets with stratification on our label to ensure both sets are representative of the population distribution.

```
X = risk_train_dummies.drop('CLASS_yes', axis=1)
y = risk_train_dummies['CLASS_yes']
# Split the data 80%-20% into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    stratify=y,
                                                    test_size=0.2,
                                                    random_state=0)
```

Random Forest model

The first model we will evaluate is the random forest model. We will use 150 estimators, a balanced class weight and specify the minimum nodes to consider a split as 10 to avoid overfitting.

```
rf_model = RandomForestClassifier(n_estimators=150,
                                  class_weight = 'balanced',
                                  min_samples_split = 10,
                                  random_state=0)
rf_model.fit(X_train, y_train)
```

```
RandomForestClassifier(class_weight='balanced', min_samples_split=10,
                       n_estimators=150, random_state=0)
```

```python
# Check important features
feature_importances_df = pd.DataFrame({"feature": list(X.columns), "importance": rf_model.feature_importances_})
feature_importances_df.sort_values("importance", ascending=False, inplace=True)

feature_importances_df
```

	feature	importance
0	VALUE_ORDER	0.290539
2	SESSION_TIME	0.211717
30	NEUKUNDE_yes	0.064356
1	AMOUNT_ORDER	0.047989
3	B_EMAIL_yes	0.047231
4	B_TELEFON_yes	0.027541
18	CHK_LADR_yes	0.025930
5	FLAG_LRIDENTISCH_yes	0.023415
13	WEEKDAY_ORDER_Saturday	0.020637
14	WEEKDAY_ORDER_Sunday	0.020623
17	WEEKDAY_ORDER_Wednesday	0.019287
9	Z_METHODE_check	0.018556
7	Z_METHODE_Eurocard	0.017784
12	WEEKDAY_ORDER_Monday	0.017660
15	WEEKDAY_ORDER_Thursday	0.017072
8	Z_METHODE_Visa	0.015245
11	Z_METHODE_debit_note	0.015228
16	WEEKDAY_ORDER_Tuesday	0.015061
6	FLAG_NEWSLETTER_yes	0.011650
10	Z_METHODE_debit_card	0.010879
28	FAIL_RORT_yes	0.008967
25	FAIL_LORT_yes	0.007877
23	CHK_IP_yes	0.007359
22	CHK_COOKIE_yes	0.007009
19	CHK_RADR_yes	0.005778
27	FAIL_RPLZ_yes	0.005637
29	FAIL_RPLZORTMATCH_yes	0.004911
24	FAIL_LPLZ_yes	0.004615
26	FAIL_LPLZORTMATCH_yes	0.003828
20	CHK_KTO_yes	0.002870
21	CHK_CARD_yes	0.002748

According to the random forest classifier, VALUE_ORDER, SESSION_TIME and NEUKUNDE are the features that are most important in classifying the label.

```python
print(classification_report(y_test, y_pred_rf))

cm = confusion_matrix(y_test, y_pred_rf)
print(cm)
```

```
              precision    recall  f1-score   support

           0       0.95      0.95      0.95      5651
           1       0.17      0.15      0.16       349

    accuracy                           0.91      6000
   macro avg       0.56      0.55      0.55      6000
weighted avg       0.90      0.91      0.90      6000

[[5396  255]
 [ 298   51]]
```

```python
check_cost(cm)
```

```
16175
```

The precision and recall are low on the high-risk class. This is understandable because decision trees and forests do not perform well on imbalanced data. The accuracy score is misleading because the dataset has many low-risk instances overall. This model may fail to capture the minority class effectively.

XGBoost Model

The next model we will evaluate is XGBoost. We will fit the model with scale_pos_weight = 10 since we are dealing with imbalanced data.

```
model_xgb = XGBClassifier(random_state=0, scale_pos_weight = 10, )
model_xgb.fit(X_train, y_train)
```

```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None, feature_types=None,
              gamma=None, gpu_id=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=None, max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=None, max_leaves=None,
              min_child_weight=None, missing=nan, monotone_constraints=None,
              n_estimators=100, n_jobs=None, num_parallel_tree=None,
              predictor=None, random_state=0, ...)
```

```
y_pred_xgb = model_xgb.predict(X_test)
cm = confusion_matrix(y_test, y_pred_xgb)
print(cm)
```

```
[[5048  603]
 [ 236  113]]
```

```
print(classification_report(y_test, y_pred_xgb))
```

```
              precision    recall  f1-score   support

           0       0.96      0.89      0.92      5651
           1       0.16      0.32      0.21       349

    accuracy                           0.86      6000
   macro avg       0.56      0.61      0.57      6000
weighted avg       0.91      0.86      0.88      6000
```

```
check_cost(cm)
```

```
14815
```

This model seems to capture the high risk class better than the random forest model.

Logistic Regression Model

The next model we will evaluate is Logistic Regression. We will fit the model with balanced weights.

```
log_model = LogisticRegression(class_weight='balanced', random_state=0, n_jobs=-1)
log_model.fit(X_train, y_train)
```

```
LogisticRegression(class_weight='balanced', n_jobs=-1, random_state=0)
```

```
y_pred_lr = log_model.predict(X_test)
cm = confusion_matrix(y_test, y_pred_lr)
print(cm)
```

```
[[3675 1976]
 [ 101  248]]
```

```
print(classification_report(y_test, y_pred_lr))
```

```
              precision    recall  f1-score   support

           0       0.97      0.65      0.78      5651
           1       0.11      0.71      0.19       349

    accuracy                           0.65      6000
   macro avg       0.54      0.68      0.49      6000
weighted avg       0.92      0.65      0.75      6000
```

```
check_cost(cm)
```

```
14930
```

So far, this model captures the most high-risk purchase orders but is biased towards the high risk class so has a lower recall score on the majority group. We will need to tune the class weights to see if we can get a better result. The misclassification cost is similar to XGBoost.

SVC Model

The next model we will evaluate is SVC. We will fit the model with balanced weights.

```
svc_model = SVC(class_weight='balanced', random_state=0)
```

```
svc_model.fit(X_train, y_train)
```

```
SVC(class_weight='balanced', random_state=0)
```

```
y_pred_svc = svc_model.predict(X_test)
```

```
print(classification_report(y_test, y_pred_svc))

cm = confusion_matrix(y_test, y_pred_svc)
print(cm)
```

```
              precision    recall  f1-score   support

           0       0.97      0.65      0.78      5651
           1       0.11      0.69      0.19       349

    accuracy                           0.65      6000
   macro avg       0.54      0.67      0.48      6000
weighted avg       0.92      0.65      0.74      6000

[[3655 1996]
 [ 107  242]]
```

```
check_cost(cm)
```

```
15330
```

SVC and Logistic Regression had very similar results.

To summarize, introducing bias to ensure more instances are classified as high risk sacrificed our accuracy but seemed to improve F1-scores and reduced cost. Although it had the highest accuracy, the random forest model is sensitive to class imbalance.

Tuning Class Weights

Let's try different class weights and compare SVC with Logistic Regression

```
log_model1 = LogisticRegression(class_weight={0:1, 1:10}, random_state=0, n_jobs=-1)
log_model1.fit(X_train, y_train)
```

```
LogisticRegression(class_weight={0: 1, 1: 10}, n_jobs=-1, random_state=0)
```

```
y_pred_lr = log_model1.predict(X_test)
cm = confusion_matrix(y_test, y_pred_lr)
print(cm)
```

```
[[4791  860]
 [ 186  163]]
```

```
print(classification_report(y_test, y_pred_lr))
```

```
              precision    recall  f1-score   support

           0       0.96      0.85      0.90      5651
           1       0.16      0.47      0.24       349

    accuracy                           0.83      6000
   macro avg       0.56      0.66      0.57      6000
weighted avg       0.92      0.83      0.86      6000
```

```
check_cost(cm)
```

```
13600
```

With a class weight ratio of 1 to 10, we were able to increase the F1 score and obtain the lowest misclassification cost with our logistic regression model so far.

```
svc_model = SVC(class_weight={0:1, 1:11}, random_state=0, probability=True)
```

```
svc_model.fit(X_train, y_train)
y_pred_svc = svc_model.predict(X_test)
```

```
print(classification_report(y_test, y_pred))

cm = confusion_matrix(y_test, y_pred_svc)
print(cm)
```

```
              precision    recall  f1-score   support

           0       0.96      0.85      0.90      5651
           1       0.15      0.44      0.23       349

    accuracy                           0.83      6000
   macro avg       0.56      0.65      0.57      6000
weighted avg       0.91      0.83      0.86      6000

[[4909  742]
 [ 217  132]]
```

```
check_cost(cm)
```

```
14560
```

The logistic regression and SVC models have relatively similar performance. We tested weight ratios of 1:10, 1:11, and 1:12 which seemed to give us the lowest costs overall without sacrificing too much in the F1 score.

Model Selection

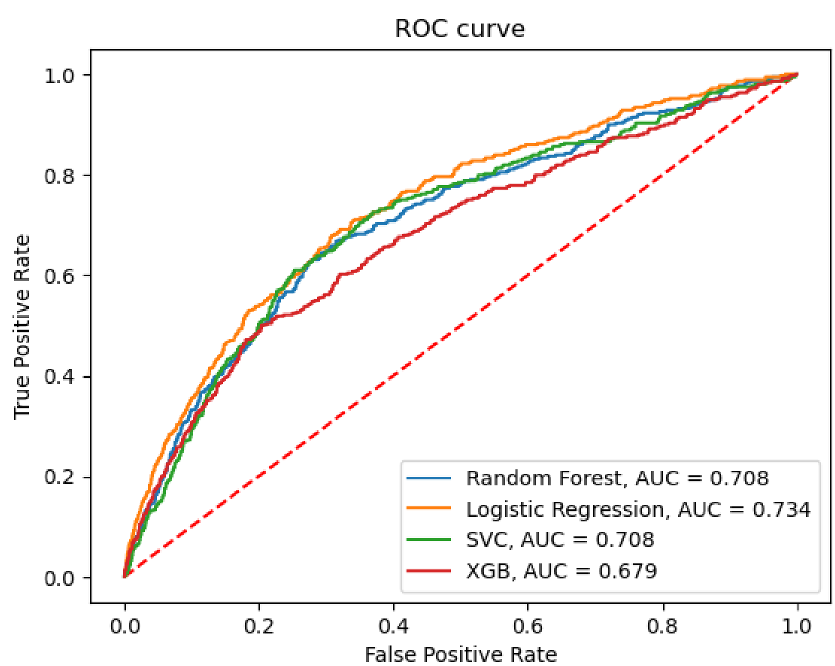
Let's plot the ROC curve and calculate the AUC to compare our different models.

```
plt.plot([0, 1], [0, 1],'r--')

# Random forest
prob_rf = rf_model.predict_proba(X_test)
fpr, tpr, thresh = roc_curve(y_test, prob_rf[:,1])
aucrf = roc_auc_score(y_test, prob_rf[:,1])
plt.plot(fpr, tpr, label=f'Random Forest, AUC = {str(round(aucrf,3))}')
# Logistic regression
prob_log = log_model1.predict_proba(X_test)
fpr, tpr, thresh = roc_curve(y_test, prob_log[:,1])
auclog = roc_auc_score(y_test, prob_log[:,1])
plt.plot(fpr, tpr, label=f'Logistic Regression, AUC = {str(round(auclog,3))}')
# SVC
prob_svc = svc_model.predict_proba(X_test)
fpr, tpr, thresh = roc_curve(y_test, prob_svc[:,1])
aucsvc = roc_auc_score(y_test, prob_svc[:,1])
plt.plot(fpr, tpr, label=f'SVC, AUC = {str(round(aucsvc,3))}')

prob_xgb = model_xgb.predict_proba(X_test)
fpr, tpr, thresh = roc_curve(y_test, prob_xgb[:,1])
aucxgb = roc_auc_score(y_test, prob_xgb[:,1])
plt.plot(fpr, tpr, label=f'XGB, AUC = {str(round(aucxgb,3))}')

plt.ylabel("True Positive Rate")
plt.xlabel("False Positive Rate")
plt.title("ROC curve")
plt.legend()
plt.show()
```



Logistic regression has the highest AUC value so we will select this model to perform our classification on the incoming purchase orders.

Data Preparation

Now that we have selected Logistic Regression, let's prepare the training and testing data to deploy our model.

Based on what we accomplished in our data exploration, we will first begin by specifying the credit card type in the payment method and then drop columns that we did not include in our initial analysis. The following columns will be removed in the training and testing sets:

- Z_CARD_ART
- Z_CARD_VALID
- Z_LAST_NAME
- ANUMMER_01
- ANUMMER_02
- ANUMMER_03
- ANUMMER_04
- ANUMMER_05
- ANUMMER_06
- ANUMMER_07
- ANUMMER_08
- ANUMMER_09
- ANUMMER_10
- B_BIRTHDATE
- AMOUNT_ORDER_PRE
- VALUE_ORDER_PRE
- DATE_LORDER
- MAHN_AKT
- MAHN_HOECHST
- TIME_ORDER

Next, we will scale the data and encode our categorical features.

```
scaler = MinMaxScaler()
training[['VALUE_ORDER', 'SESSION_TIME']] = scaler.fit_transform(training[['VALUE_ORDER', 'SESSION_TIME']])
testing[['VALUE_ORDER', 'SESSION_TIME']] = scaler.transform(testing[['VALUE_ORDER', 'SESSION_TIME']])
```

```
training_dummies = pd.get_dummies(training, drop_first=True)
training_dummies.columns
```

```
testing_dummies = pd.get_dummies(testing, drop_first=True)
testing_dummies.columns
```

We can now export the training and testing set with dummy variables to CSV files for our model training.

```
training_dummies.to_csv('training-set.csv', index=False)
testing_dummies.to_csv('testing-set.csv', index=False)
```

Model Deployment

Last but not least, we will run our training algorithm on the exported training data and perform classification on the testing data. We will need to ensure the ORDER ID is dropped from our training set before fitting the logistic regression model.

Importing the training data

```
training = pd.read_csv('training-set.csv')
training = training.drop(['ORDER_ID'], axis=1)
```

```
training.columns
```

```
Index(['VALUE_ORDER', 'AMOUNT_ORDER', 'SESSION_TIME', 'CLASS_yes',
       'B_EMAIL_yes', 'B_TELEFON_yes', 'FLAG_LRIDENTISCH_yes',
       'FLAG_NEWSLETTER_yes', 'Z_METHODE_Eurocard', 'Z_METHODE_Visa',
       'Z_METHODE_check', 'Z_METHODE_debit_card', 'Z_METHODE_debit_note',
       'WEEKDAY_ORDER_Monday', 'WEEKDAY_ORDER_Saturday',
       'WEEKDAY_ORDER_Sunday', 'WEEKDAY_ORDER_Thursday',
       'WEEKDAY_ORDER_Tuesday', 'WEEKDAY_ORDER_Wednesday', 'CHK_LADR_yes',
       'CHK_RADR_yes', 'CHK_KTO_yes', 'CHK_CARD_yes', 'CHK_COOKIE_yes',
       'CHK_IP_yes', 'FAIL_LPLZ_yes', 'FAIL_LORT_yes', 'FAIL_LPLZORTMATCH_yes',
       'FAIL_RPLZ_yes', 'FAIL_RORT_yes', 'FAIL_RPLZORTMATCH_yes',
       'NEUKUNDE_yes'],
      dtype='object')
```

We will fit the logistic regression model using a class weight ratio of 1:10 which gave us the best results in our previous analysis.

```
X_train = training.drop('CLASS_yes', axis=1)
y_train = training['CLASS_yes']
```

```
log_model = LogisticRegression(class_weight={0:1, 1:10}, random_state=0, n_jobs=-1)
```

```
log_model.fit(X_train, y_train)
```

```
LogisticRegression(class_weight={0: 1, 1: 10}, n_jobs=-1, random_state=0)
```

```
y_pred = log_model.predict(X_train)
cm = confusion_matrix(y_train, y_pred)
print(cm)
```

```
[[23664  4590]
 [  868   878]]
```

```
print(classification_report(y_train, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.96      0.84      0.90     28254
           1       0.16      0.50      0.24      1746

    accuracy                           0.82     30000
   macro avg       0.56      0.67      0.57     30000
weighted avg       0.92      0.82      0.86     30000
```

The precision, recall and F1 score on the entire training set is similar to what we observed during our model selection stage. Let's make predictions on our testing data which consists of incoming purchase orders.

```
testing = pd.read_csv('testing-set.csv')
X_test = testing.drop('ORDER_ID', axis=1)
```

```
predictions = log_model.predict(X_test)
```

We want to ensure we have the order ID corresponding to our predicted result so we will concatenate the predictions to our initial data frame and store these results in a text file consisting of ORDER_ID and CLASS.

```
testing['CLASS'] = predictions
testing.loc[testing.CLASS == 0, 'CLASS'] = 'no'
testing.loc[testing.CLASS == 1, 'CLASS'] = 'yes'
```

```
testing = testing.rename(columns={'ORDER_ID':'ORDER-ID'})
```

```
result = testing[['ORDER-ID', 'CLASS']]
```

```
result.to_csv('classification-result.txt', index=False, sep='\t')
```

Here is a sample of our results:

```
ORDER-ID	CLASS
49916	no
49918	no
49920	yes
49921	no
49922	no
49925	no
```

Conclusion

To summarize our results, we analyzed and preprocessed our data by removing features that were unlikely to contribute information when classifying our labels and evaluated four different models. These models were the Random Forest, XGBoost, Logistic Regression and SVC models.

We evaluated the models by comparing their precision, recall and F1 score since we had a highly imbalanced dataset. This meant that our accuracy scores were misleading since we had a large proportion of low risk purchase orders to high risk. Some of our models were sensitive to this imbalance so we had to make sure we set the appropriate class weights to introduce some bias that would help us capture the high risk class more effectively.

To select the appropriate model, we compared our results using ROC curves and calculated the AUC. Logistic Regression was the model that had the highest AUC score.