2022

Random Forest and Support Vector classification algorithms in Python

LAKSHAY GOEL

Table of Contents

Introduction:

The bank wants to use a classification model that can predict customer churn. Construct a suitable classification model for the bank by implementing both random forest and support vector classification algorithms in Python.

Dataset:

Link: CustomerChurn.csv

Number of Instances: 6237

Number of Attributes: 15 independent variables + 1 target variable

Independent Variables:

Customer_Age – Customer age in years Gender – M = Male, F = Female

Dependent_count – Number of dependents

Education_Level – Highest education level of the customer

Marital_Status – Married, Single, Divorced

Income_Category – Annual income category of the customer

Card_Category – Type of card. Platinum cards offer more reward points than gold cards, which offer more reward points than silver cards. Blue cards offer the least number of reward points on purchases.

Months_on_book – Period of relationship (in months) with the bank.

Total_Relationship_Count – Total number of products held by the customer (e.g., Saving account, Car loan, Credit Card, etc.)

Months_Inactive - Number of months inactive in the last 12 months

Contacts_Count – Number of customer service contacts in the last 12 months

Credit_Limit – credit limit on the credit card

Total_Revolving_Bal – Total revolving balance on the credit card (the portion of credit card spending that goes unpaid at the end of a billing cycle)

Total_Trans_Amt – Total transaction amount in the last 12 months

Total_Trans_Ct – Total transaction count in the last 12 months

Target Variable:

Attrition_Flag – Two labels - ‘Existing Customer’, ‘Attrited Customer’ (Customer who has churned)

Data Preparation

Import dataset

Import the dataset using pandas read_csv and provide the path where the file is stored.

```
dataset = pd.read_csv("/content/drive/MyDrive/Applied Statistics and Machine Learning/CA-ONE /CustomerChurn.csv")
pd.set_option('display.max_columns', None)
```

Examining the data

Using commands to examine the data frame to make sure the data is imported correctly and if there are any missing values in the dataset.

- **dataset.head()**- Displays the top 5 rows in the data.
- **dataset.shape**- Displays the number of rows and columns in the dataset.
- **dataset.info()**- Displays the data types and missing values in the dataset.
- **dataset.describe()**- Displays the basic statistics like mean, min, max, etc.

```
print(dataset.head())
print(dataset.shape)
print(dataset.info())
print(dataset.describe())
```

Summary:

- Total number of variables or columns: 16
- Total number of entries: 6237
- Imbalanced dataset, 5256 Existing customers and 981 Attrited customers.
- No missing value in the dataset.
- Dataset contains non-numerical data type that needs to be converted.
- Looking at the statistics from the dataset.describe, it is clear that there is a big contrast in the distribution between the columns. for e.g., 'Customer_Age' column ranges between 26 and 73 with a mean value of 46.38 whereas Credit_Limit has a range of 1438.3 to 34516 with a mean value of 8474.2. Passing the data like this to the algorithm will make it biased towards the column with larger values.

Step 1: Converting Categorical columns to Numerical columns

The dataset has six category categories that need to be transformed into numerical columns:

- Attrition_Flag (Target Variable)
- Income_Category
- Gender
- Card_Category
- Education_Level
- Marital_Status

Attrition_Flag (Target Variable) Column contains only 2 labels Existing Customer and Attrited Customer and can be converted to Binary with:

- Existing Customer: 0 (Negative)
- Attrited Customer: 1 (Positive)

Income_Category contains 5 categories which can be divided on the salary range:

- Less than $40K : 0
- $40K-$60K : 1
- $60K-$80K : 2
- $80K-$120K : 3
- $120K+ : 4

Gender Column contains 2 labels which can be converted to 0 and 1:

- M: 0
- F: 1

Card_Category Column contains 4 labels that can be converted into numerical based on the reward points the card offers:

- Blue: 0
- Silver: 1
- Gold: 2
- Platinum: 3

Education_Level Column contains 5 labels that can be converted into numerical based on the level of education:

- Uneducated: 0
- High School: 1
- Graduate: 2
- Post-Graduate: 3
- Doctorate: 4

```
dataset['Attrition_Flag'] = dataset['Attrition_Flag'].map({'Existing Customer
':0, 'Attrited Customer':1})
dataset['Income_Category'] = dataset['Income_Category'].map({'Less than $40K'
:0, '$40K - $60K': 1, '$60K - $80K':2,'$80K - $120K':3,'$120K +':4 })
dataset['Gender'] = dataset['Gender'].map({'F': 1, 'M': 0})
dataset['Card_Category'] = dataset['Card_Category'].map({'Blue':0, 'Silver':1
,'Gold':2,'Platinum':3})
dataset['Education_Level'] = dataset['Education_Level'].map({'Uneducated': 0,
 'High School': 1,'Graduate': 2,'Post-Graduate':3,'Doctorate':4})
```

Marital_Status Column contains "Single," "Married," and "Divorced" labels which cannot be transformed into numerical features, so we must use a technique known as the "Dummy Column Approach" or "One-Hot Encoding," in which we turn each categorical value into a new categorical column and give it a binary value of 0 or 1. For instance, if a customer is "Married," the value in the "Married" column will be 1 and 0 in the other columns. The Marital Status Column will be divided into 3 columns for 3 different labels..

```
categorical_features = ['Marital_Status']
final_data = pd.get_dummies(dataset, columns = categorical_features)
print(final_data.info())
print(final_data.head(2))
```

Summary:

- final_data is the new name of the transformed dataset
- All columns are of numerical data types i.e. int64, float64, and uint8
- Total number of variables or columns: 18
- Total number of entries: 6237
- ‘Marital_Status’ Column has been divided into 3 columns as shown below screenshot:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6237 entries, 0 to 6236
Data columns (total 18 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   Attrition_Flag            6237 non-null   int64
 1   Customer_Age              6237 non-null   int64
 2   Gender                    6237 non-null   int64
 3   Dependent_count           6237 non-null   int64
 4   Education_Level           6237 non-null   int64
 5   Income_Category           6237 non-null   int64
 6   Card_Category             6237 non-null   int64
 7   Months_on_book            6237 non-null   int64
 8   Total_Relationship_Count  6237 non-null   int64
 9   Months_Inactive           6237 non-null   int64
 10  Contacts_Count            6237 non-null   int64
 11  Credit_Limit              6237 non-null   float64
 12  Total_Revolving_Bal       6237 non-null   int64
 13  Total_Trans_Amt           6237 non-null   int64
 14  Total_Trans_Ct            6237 non-null   int64
 15  Marital_Status_Divorced   6237 non-null   uint8
 16  Marital_Status_Married    6237 non-null   uint8
 17  Marital_Status_Single     6237 non-null   uint8
```

Step 2: Dividing the dataset into feature sets and label

The dataset must be split up so that the target variable is in the label column (the algorithm's target column), and all independent variables are in a single data frame.

Use the 'drop' method to remove the target column from independent variables and making it a separate series as shown below:

```
X = final_data.drop('Attrition_Flag', axis = 1) # Features
Y = final_data['Attrition_Flag'] # Labels
print(type(X))
print(type(Y))
print(X.shape)
print(Y.shape)
```

Step 3: Normalizing the numerical features

In order to prevent the algorithm from favouring a column with greater values (such as Credit Limit) over a column with lower values (such as Customer Age), the data frame must now be changed so that all columns have a mean of 0 and a variance of 1.

Use the StandardScaler() method from the sklearn package and pass the dataset named 'X' from the last step as a variable:

```
feature_scaler = StandardScaler()
X_scaled = feature_scaler.fit_transform(X)
```

We now use 'X_scaled' as our standardized and normalized dataset and 'Y' as our target dataset.

Step 4: Diving the dataset into Training data and Test Data

Use the 'K-Fold Cross Validation' method when running the 'Random Forest' and 'Support Vector' Classification Algorithms to divide the dataset into training data and test data instead of manually dividing the data.

Implementation of the Random Forest Classifier

Now, that our data is prepared, we can now create the classifiers. Starting with Random Forest Classifier.

Step 1: Create a pipeline

A pipeline is a set of data processing elements connected in series (or a sequence of steps) to be carried out on our training dataset. The pipeline will contain 2 steps which are as below:

1. Balancing: We are unable to pass the dataset to the algorithm because of the previously mentioned imbalanced dataset. Before submitting the dataset to the algorithm to extract a pattern, it must be balanced. We'll employ a technique called "SMOTE" (Synthetic Minority Oversampling Technique) to add some synthetic samples to the minority class using the samples we currently have.
2. Classification: Passing our dataset into the Random Forest Classifier to create the model with parameters.

```
model = Pipeline([
        ('balancing', SMOTE(random_state = 101)),
        ('classification', RandomForestClassifier(criterion='entropy', max_fe
atures='auto', random_state=1) )
    ])
```

Hyperparameter:

One of the Random Forest Classifier's parameters, the number of decision trees to develop, is still unknown. As a response, we'll use the hyperparameter "n estimator." This is how many decision trees our Random Forest Classifier will grow. This Hyperparameter is part of our pipeline's second step. Have to test a variety of possible numbers in the form of a list as we are unsure of the precise number of trees.

```
grid_param = {'classification__n_estimators': [10,20,30,40,50,100]}
```

We start with smaller values like 10 and move upwards until we have the optimal number of decision trees.

Choice of Evaluation Metric

We have 4 different types of evaluation metrics which are all derived from the Confusion Matrix (diagram below):

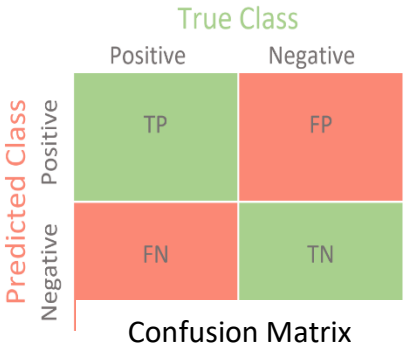
- Accuracy
- Recall
- Precision
- F1 Score

Accuracy: Measures the overall accuracy of the model but can't be used on an imbalanced dataset.

Recall: Measures the model's ability to avoid 'False negatives'

Precision: Measure the model's ability to avoid 'False Positives'

F1 Score: Measures the model's abilities to avoid 'False Negatives' as well as 'False Positives'



Confusion Matrix

True Positive: Refers to no. of predictions where the classifier predicts the 'positive' class as 'positive'.

True Negative: Refers to no. of predictions where the classifier predicts the 'negative' class as 'negative'.

False Positive: Refers to no. of predictions where the classifier predicts the 'positive' class as 'negative'.

False Positive: Refers to no. of predictions where the classifier predicts the 'negative' class as 'positive'.

Now, looking at the problem statement, we need to create a classification model to predict 'Customer Churn'. Seeing as we took 'Existing Customer' as 0 (negative) and 'Attrited Customer' as 1 (Positive), that means:

True Positive: Customer is churning (positive) and the algorithm also predicts Churn (positive)

True Negative: Customer is not churning (negative) and the algorithm also predicts not churn (negative)

False Positive: Customer is not churning (negative) and algorithm predicts churn (positive)

False Negative: Customer is churning (Positive) and algorithm predicts not churning (negative)

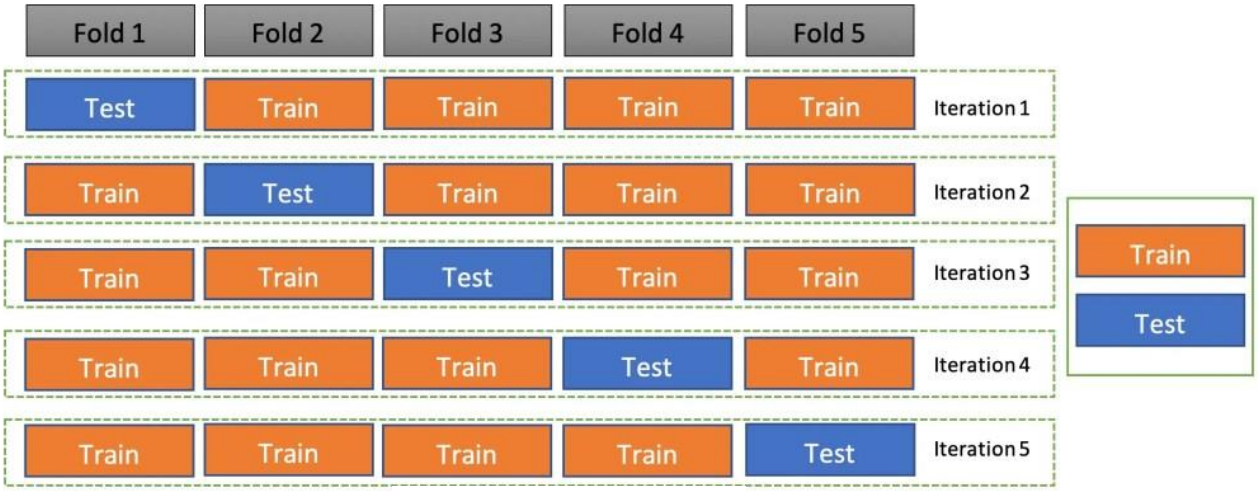
False Positive means the model is predicting the customer will leave but in reality, the customer is not looking to leave, so anyways the bank can reach out to the customer whereas, In False Negative, the model is predicting the customer will not leave but in reality, the customer will leave, and the bank cannot reach out to the customer and provide them benefits so that they will stay. As a result, False Negative seems to be the worse one than False Positive.

The best evaluation metric for our model is 'Recall'.

K-fold Cross-validation:

As discussed earlier, we are going to use 'K-fold Cross-Validation' to divide our dataset into training and test data where K is the number of groups (or folds) which the data is divided into. The procedure is as follows:

1. Shuffle the dataset randomly.
2. Divide the dataset into 'K' equal number of folds.
3. For each unique fold (or group):
   3.1. Take a group as our test dataset and the remaining groups as the training data.
   3.2. Fit a model on the training dataset and evaluate it on the test data.
   3.3. Retain the evaluation score on the test data set and discard the model.
4. Summarize the skill of the model by taking mean and std. deviation of the test performances.



K-Fold Cross-Validation

```
gd_sr = GridSearchCV(estimator=model, param_grid=grid_param, scoring='recall'
, cv=5)
```

Here, CV means Cross-Validation, cv=5 is the no. of folds, and GridSearch will help us in the hyperparameter tuning.

'K-Fold Cross Validation' is an evaluation technique which avoids overfitting. As we have 6237 instances, 5 folds are probably enough.

Step 2: Implementing the 'GridSearch'

The next step is to implement the GridSearch method. It will create multiple models with different number of decision trees and Cross-Validate each one to give us the best score and optimal number of decision trees.

```
gd_sr.fit(X_scaled, Y)

best_parameters = gd_sr.best_params_
print(best_parameters)

best_result = gd_sr.best_score_ # Mean cross-
validated score of the best_estimator
print(best_result)

featimp = pd.Series(gd_sr.best_estimator_.named_steps["classification"].featu
re_importances_, index=list(X)).sort_values(ascending=False) # Getting featur
e importances list for the best model
print(featimp)
```

Here, we are passing our entire feature set and our target variable into GridSearch and going to print out the 'best_parameter' (or the optimal number of decision trees) and 'best_result' for our model with the significance of each variable. Result is as follows:

```
{'classification__n_estimators': 100}
0.6593235263648607
Total_Trans_Ct              0.243079
Total_Trans_Amt             0.188568
Total_Revolving_Bal         0.152209
Total_Relationship_Count    0.081553
Months_Inactive             0.070021
Contacts_Count              0.054605
Credit_Limit                0.044319
Customer_Age                0.041606
Months_on_book              0.029025
Dependent_count             0.027385
Income_Category             0.022592
Education_Level             0.019013
Gender                      0.009298
Marital_Status_Married      0.006339
Marital_Status_Single       0.005063
Card_Category               0.003563
Marital_Status_Divorced     0.001760
dtype: float64
```

Optimal Number of Trees (n_estimator) : 100

Best Score: 0.659

Hyperparameter Tuning and Avoid Overfitting for Random Forest Classifier:

Hyperparameter tuning:

After having a score of 0.659 and the number of estimators is 100 (that is our max no. in the list), we will try to tune our model by increasing the number of decision trees and running the algorithm again to see if it increases the score.

```
{'classification__n_estimators': 175}
0.6695017093131669
Total_Trans_Ct              0.235930
Total_Trans_Amt             0.196167
Total_Revolving_Bal         0.151952
Total_Relationship_Count    0.082477
Months_Inactive             0.068561
Contacts_Count              0.055352
Credit_Limit                0.044355
Customer_Age                0.041205
Months_on_book              0.029187
Dependent_count             0.026189
Income_Category             0.022029
Education_Level             0.019711
Gender                      0.010071
Marital_Status_Married      0.006394
Marital_Status_Single       0.005126
Card_Category               0.003674
Marital_Status_Divorced     0.001619
dtype: float64
```

Optimal Number of Trees (n_estimator) : 175

Best Score: 0.669

Result:

Increasing the no. of n_estimator (no. of decision trees) did not increase the performance of the model by large margins, so we can try other approaches to increase efficiency.

Avoid Overfitting:

Generally, Random Forest Classifiers are good at avoiding overfitting because they are a combination of multiple decision trees And the 'K-Fold Cross Validation' method helps to avoid overfitting. In addition to hyperparameter tuning and 'K-Fold Cross Validation', we can further tune our model by including the only columns which are more significant (with the higher score in the GridSearch result as seen in the image above).

Performing the same steps as in Random Forest Classifier but specifying the significant columns first and increasing the 'n_estimator' as shown below.

```
# Selecting features with higher significance and redefining the feature set
X_ = final_data[['Total_Trans_Ct','Total_Trans_Amt','Total_Revolving_Bal']]

feature_scaler = StandardScaler()
X_scaled_ = feature_scaler.fit_transform(X_)

model = Pipeline([
        ('balancing', SMOTE(random_state = 101)),
        ('classification', RandomForestClassifier(criterion='entropy', max_fe
atures='auto', random_state=1) )
    ])
grid_param = {'classification__n_estimators': [100,150,175,200,250,300,350,40
0]}

gd_sr = GridSearchCV(estimator=model, param_grid=grid_param, scoring='recall'
, cv=5)

gd_sr.fit(X_scaled_, Y)

best_parameters = gd_sr.best_params_
print(best_parameters)

best_result = gd_sr.best_score_
print(best_result)
```

Result:

```
{'classification__n_estimators': 125}
0.6277219517248525
```

Using the Top 3 significant variables decreased the performance efficiency of the model.

Further tuning the model with more variables:

```
# Selecting features with higher significance and redefining feature set
X_ = final_data[['Total_Trans_Ct','Total_Trans_Amt','Total_Revolving_Bal','To
tal_Relationship_Count','Total_Relationship_Count','Months_Inactive','Contact
s_Count']]

feature_scaler = StandardScaler()
X_scaled_ = feature_scaler.fit_transform(X_)

#Tuning the random forest parameter 'n_estimators' and implementing cross-
validation using Grid Search
model = Pipeline([
        ('balancing', SMOTE(random_state = 101)),
        ('classification', RandomForestClassifier(criterion='entropy', max_fe
atures='auto', random_state=1) )
    ])
grid_param = {'classification__n_estimators': [100,125,150,175,200,225,250]}

gd_sr = GridSearchCV(estimator=model, param_grid=grid_param, scoring='recall'
, cv=5)

gd_sr.fit(X_scaled_, Y)

best_parameters = gd_sr.best_params_
print(best_parameters)

best_result = gd_sr.best_score_ # Mean cross-
validated score of the best_estimator
print(best_result)
```

Result:

```
{'classification__n_estimators': 175}
0.6613488034807832
```

Obtaining a score of 0.661 with the Top 6 Significant Variables, which is better than the Top 3 but still inferior to the value obtained with no Significant Variables. So, with a n estimator of 175, the best result we can achieve from the Random Forest Classifier is 0.669.

Implementation of the Support Vector Classifier

The Data Preparation Steps for Support Vector are the same as in Random Forest. SVC can be implemented using the same code. The only changes are using SVC for the pipeline and the Hyperparameters.

Step 1 : Create a Pipeline

Method is the same as in RFC, the only change is passing our dataset into SVC instead of RFC as shown below.

```
model = Pipeline([
        ('balancing', SMOTE(random_state = 101)),
        ('classification', SVC(random_state=1))
    ])
```

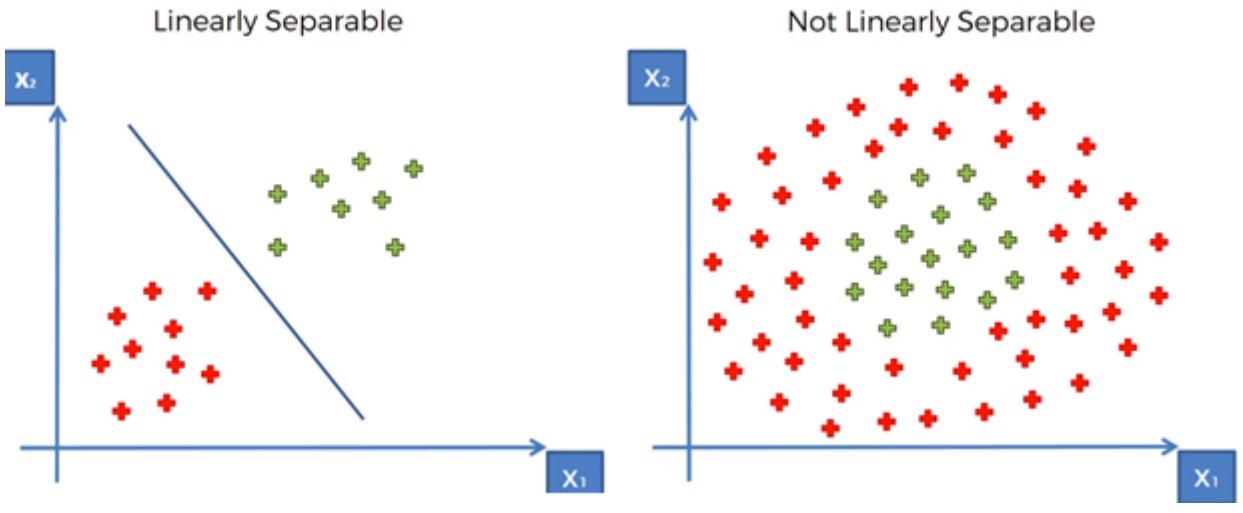
Hyperparameters:

Unlike, RFC which includes only 1 hyperparameter in the form of n_estimator, we are using 2 hyperparameters in the case of SVC which are 'kernel' and 'Regularization Parameter' depicted by 'C' which is used to avoid overfitting.

```
grid_param = {'classification__kernel': ['linear','poly','rbf','sigmoid'], 'c
lassification__C': [.001,.01,.1,1,10,100]}
```

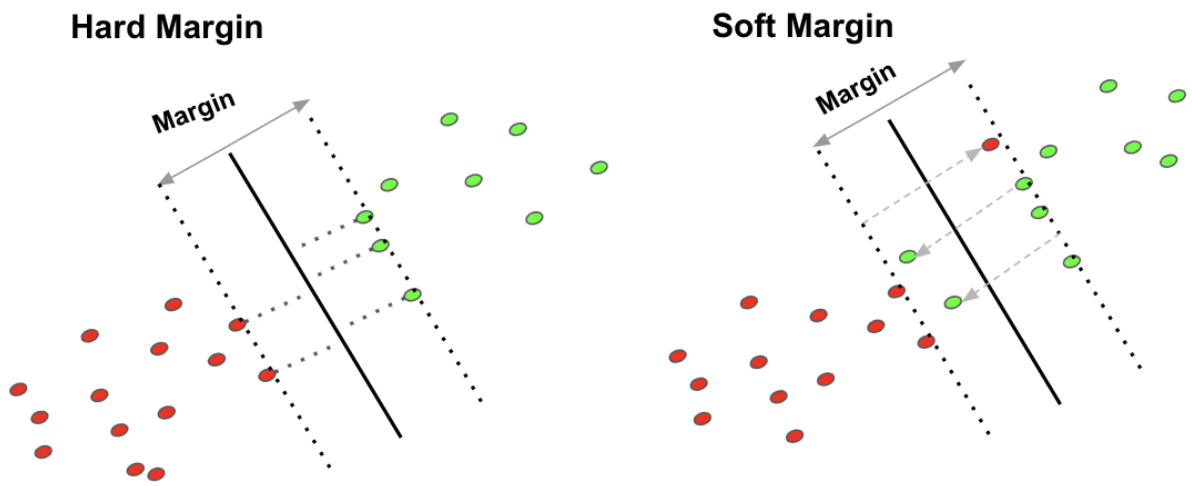
Kernel:

In cases where the two classes are non-linearly separated in consideration of a two-dimensional space, a kernel can be used to transform a two-dimensional space into a higher dimension by creating as many additional variables as needed in order to achieve a position where the groups can now be linearly separable.



Regularization Parameter 'C' (to avoid Overfitting):

Regularization is a technique used to calibrate our machine learning models in order to avoid underfitting and overfitting. In case of SVC, we use 'C' as a regularization parameter whose value can be controlled in order to allow some data points to end up within or on the other side of the margin which are generally called as 'Slack Data Points'.



Higher the value of 'C', the smaller the no. of allowed slack data points (Hard Margin SVM) and on the other hand the smaller the value of 'C', the higher number of slack data points are allowed (Soft Margin SVM). Hence, the value of 'C' will control the slack data points and it's a hyperparameter that can be controlled.

Choice of Evaluation Metric and Cross-Validation

Same evaluation metric will be used as in RFC which is 'recall' and the 5 cross validation folds.

```
gd_sr = GridSearchCV(estimator=model, param_grid=grid_param, scoring='recall'
, cv=5)
```

Step 2 : Implementing GridSearch

```
gd_sr.fit(X_scaled, Y)

best_parameters = gd_sr.best_params_
print(best_parameters)

best_result = gd_sr.best_score_ # Mean cross-
validated score of the best_estimator
print(best_result)
```

This Step is the same as in RFC with the only difference in no feature sets or significant variables as SVC divides the groups by using the maximum distance from the closest data points and does not depend on other variables.

Result:

```
{'classification__C': 0.001, 'classification__kernel': 'sigmoid'}
0.8469594944576816
```

The best result for SVC is 0.84 using 'Sigmoid' kernel and with a 'C' value of 0.001. This means that we are getting better results by allowing more slack data points.

Hyperparameter Tuning and overfitting

The hyperparameter we can tune is the value of 'C'. We will try tuning by reducing the value of 'C' by running the model again and look at the results.

Result:

```
{'classification__C': 0.0001, 'classification__kernel': 'sigmoid'}
0.8949031389205429
```

After tuning the value of 'C' we are having a improved result of 0.89. Although, we can even go lower than 0.0001 but allowing that many slack data points may cause our model to overfit. So, we are going to stop here with the score of 0.89.

Results Analysis:

a. In the Real-world scenario, I will recommend the SVC (Support Vector Classifier) model as compared to RFC (Random Forest Classifier) as the score of SVC is 0.89 which is far better than 0.669 as in the case of RFC.
   Although, we wouldn't know the significant variables in the case of SVC as the model in Non-Interpretable, whereas the RFC is Interpretable but having is low score of 0.669 does not justify using this model.

b. Random Forest Classifier model is underfitting with a score of 0.669.
   Possible reasons includes:
   - Because Random Forest Classifier uses all data points to divide the data into groups as opposed to Support Vector, which employs the maximum distance between nearest data points, 6237 records are insufficient to train our model to provide the best prediction.
   - We have 15 independent variables to train our model which may not be enough for RFC and needs more data or variables.
   - One of the causes of underfitting may be an imbalanced dataset, as RFC may become biased if there is not enough data for customers who have attrition to make predictions.

References:

- https://towardsdatascience.com/confusion-matrix-for-your-multi-class-machine-learning-model-ff9aa3bf7826
- https://sqlrelease.com/introduction-to-k-fold-cross-validation-in-python
- https://medium.com/analytics-vidhya/the-svm-we-need-to-know-the-svm-we-implemented-47740d65aa5b
- https://medium.com/pursuitnotes/day-12-kernel-svm-non-linear-svm-5fdefe77836c
- https://www.linkedin.com/pulse/churn-prediction-machine-learning-ot%C3%A1vio-silveira/?trk=public_profile_article_view