Classification Models Project

Setup imports and encode values

Ensure that you have these installations installed to your python:

`pip install pydotplus`

`pip install graphviz`

`pip install seaborn`

In [1]:
```
import os
import numpy as np
import pandas as pd
import collections
from sklearn import preprocessing
import matplotlib.pyplot as plt
import shutil
from sklearn.metrics import accuracy_score, f1_score, precision_score, recall_score, c
from sklearn.tree import DecisionTreeClassifier # Import Decision Tree Classifier
from sklearn.model_selection import train_test_split # Import train_test_split functic
from sklearn import metrics #Import scikit-learn metrics module for accuracy calculati

# Encode text values to indexes(i.e. [1],[2],[3] for red,green,blue).
def encode_text_index(data, name):
    le = preprocessing.LabelEncoder()
    data[name] = le.fit_transform(data[name])
    return le.classes_

def classification_report_and_cm(y_test, y_pred, display_labels):
  print('Accuracy on test data is %.2f' % (accuracy_score(y_test, y_pred)))
  print('F1 score on test data is %.2f' % (f1_score(y_test, y_pred)))
  print('Precision Score on test data is %.2f' % (precision_score(y_test, y_pred)))
  print('Recall score on test data is %.2f' % (recall_score(y_test, y_pred)))
  print(classification_report(y_test,y_pred))

  cm = confusion_matrix(y_test, y_pred)
  disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=display_labels)
  disp.plot()
  plt.show()
```

(A) Classification Models on Titanic dataset

Load and preprocess data: drop columns not used in classification, fill N/A values with mean and encode categorical columns

In [2]:
```
titanic_train = pd.read_csv("./titanic_train.csv")
# Drop passenger name, ticket number, cabin, embarked.
titanic_train.drop(columns=['PassengerId', 'Name', 'Ticket', 'Cabin', 'Embarked'], inp
# Replace NaNs in Age with mean
```

```
titanic_train['Age'].fillna(titanic_train['Age'].mean(), inplace=True)
# Encode 'Sex' column
titanic_train['Sex'] = pd.get_dummies(titanic_train['Sex'], drop_first=True)
titanic_train.head(10)
```

Out[2]:

	Survived	Pclass	Sex	Age	SibSp	Parch	Fare
0	0	3	1	22.000000	1	0	7.2500
1	1	1	0	38.000000	1	0	71.2833
2	1	3	0	26.000000	0	0	7.9250
3	1	1	0	35.000000	1	0	53.1000
4	0	3	1	35.000000	0	0	8.0500
5	0	3	1	29.699118	0	0	8.4583
6	0	1	1	54.000000	0	0	51.8625
7	0	3	1	2.000000	3	1	21.0750
8	1	3	0	27.000000	0	2	11.1333
9	1	2	0	14.000000	1	0	30.0708

Plot Correlation Matrix for Titanic Data attributes

In [3]:
```
corr = titanic_train.corr()
corr.style.background_gradient(cmap='coolwarm')
```

Out[3]:

	Survived	Pclass	Sex	Age	SibSp	Parch	Fare
Survived	1.000000	-0.338481	-0.543351	-0.069809	-0.035322	0.081629	0.257307
Pclass	-0.338481	1.000000	0.131900	-0.331339	0.083081	0.018443	-0.549500
Sex	-0.543351	0.131900	1.000000	0.084153	-0.114631	-0.245489	-0.182333
Age	-0.069809	-0.331339	0.084153	1.000000	-0.232625	-0.179191	0.091566
SibSp	-0.035322	0.083081	-0.114631	-0.232625	1.000000	0.414838	0.159651
Parch	0.081629	0.018443	-0.245489	-0.179191	0.414838	1.000000	0.216225
Fare	0.257307	-0.549500	-0.182333	0.091566	0.159651	0.216225	1.000000

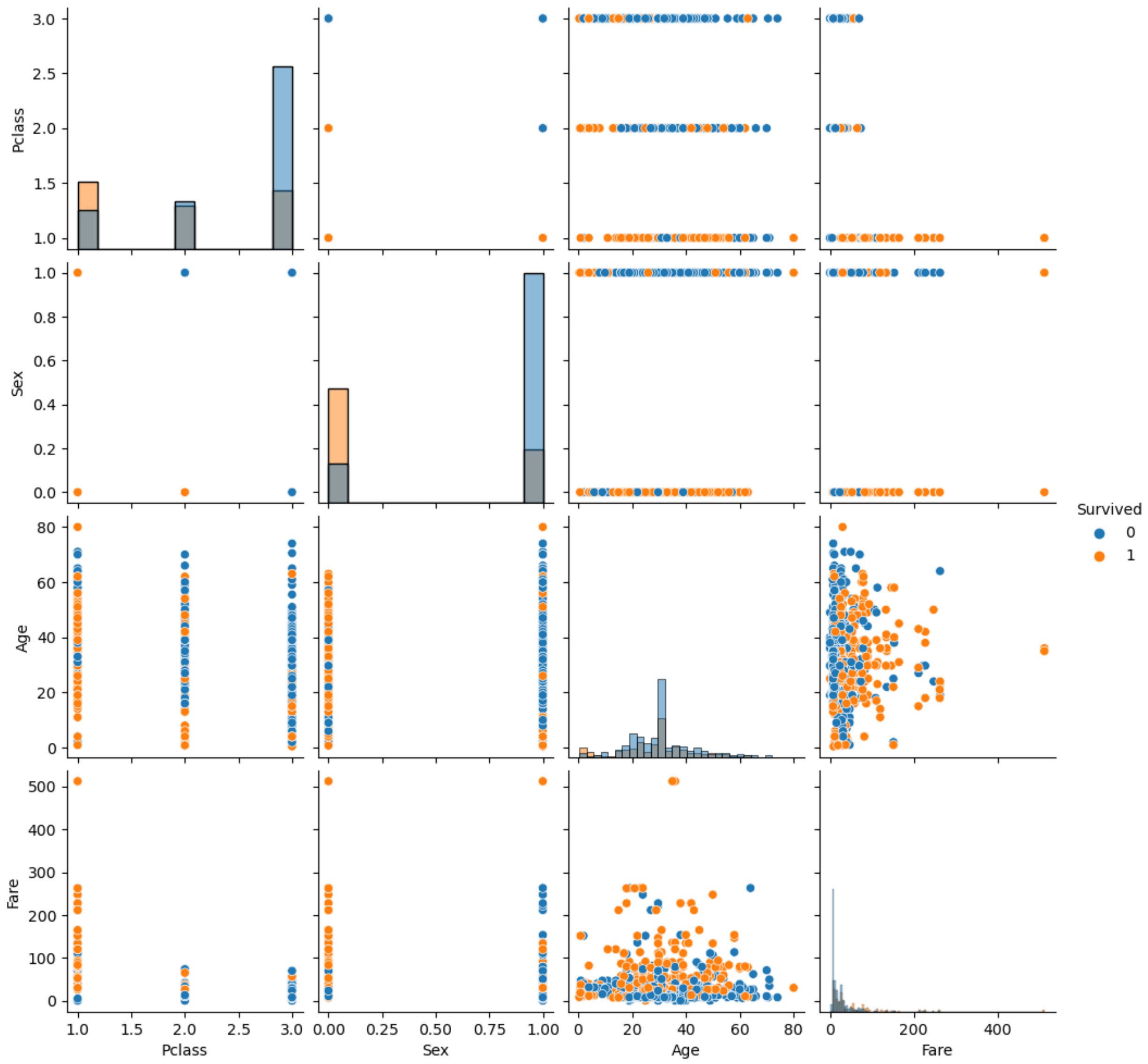
If we look at 'Survived' column which is our target for prediction, we can see that the cabin class ('Pclass' column), the gender('Sex' column) and the ticket price ('Fare' column) are the most important features. The gender and ticket class are anticorrelated with survival since gender is 0-female, 1-male and for class, increasing class number decreased chances of survival.

Features pairplot with Survived hue to get insights on the pairwise relationships within the features.

In [4]:
```
import seaborn as sns
# Paired plot using seaborn
```

```
sns.pairplot(titanic_train[['Survived', 'Pclass', 'Sex', 'Age', 'Fare']],
             hue='Survived', diag_kind="hist")
```

Out[4]:
```
<seaborn.axisgrid.PairGrid at 0x1e4138c5550>
```



From the pairplot we can see that when Sex=1(Male) only Age < 10 or so had good chances of survival. Also when Sex=0(Female) and Fare > 25 passengers had good chances of survival. Passengers with Pclass=1 had good chances of survival, much better than Pclass=2 or 3.

Prepare training features and labels for classification.

In [5]:
```
X_titanic = titanic_train[['Pclass', 'Sex', 'Age', 'SibSp', 'Fare']].values
y_titanic = titanic_train['Survived'].values

X_train, X_test, y_train, y_test = train_test_split(X_titanic, y_titanic, test_size=0.
print("X_train shape: ", X_train.shape)
print("X_test shape: ", X_test.shape)
print("y_train shape: ", y_train.shape)
print("y_test shape: ", y_test.shape)

X_train[:5]
```

```
X_train shape:  (623, 5)
X_test shape:  (268, 5)
y_train shape:  (623,)
y_test shape:  (268,)
```

Out[5]:
```
array([[ 1.        ,  1.        , 51.        ,  0.        , 26.55      ],
       [ 1.        ,  0.        , 49.        ,  1.        , 76.7292    ],
       [ 3.        ,  1.        ,  1.        ,  5.        , 46.9       ],
       [ 1.        ,  1.        , 54.        ,  0.        , 77.2875    ],
       [ 3.        ,  0.        , 29.69911765,  1.        , 14.4583    ]])
```

Apply decision tree similar to Tutorial_6_Classification.ipynb

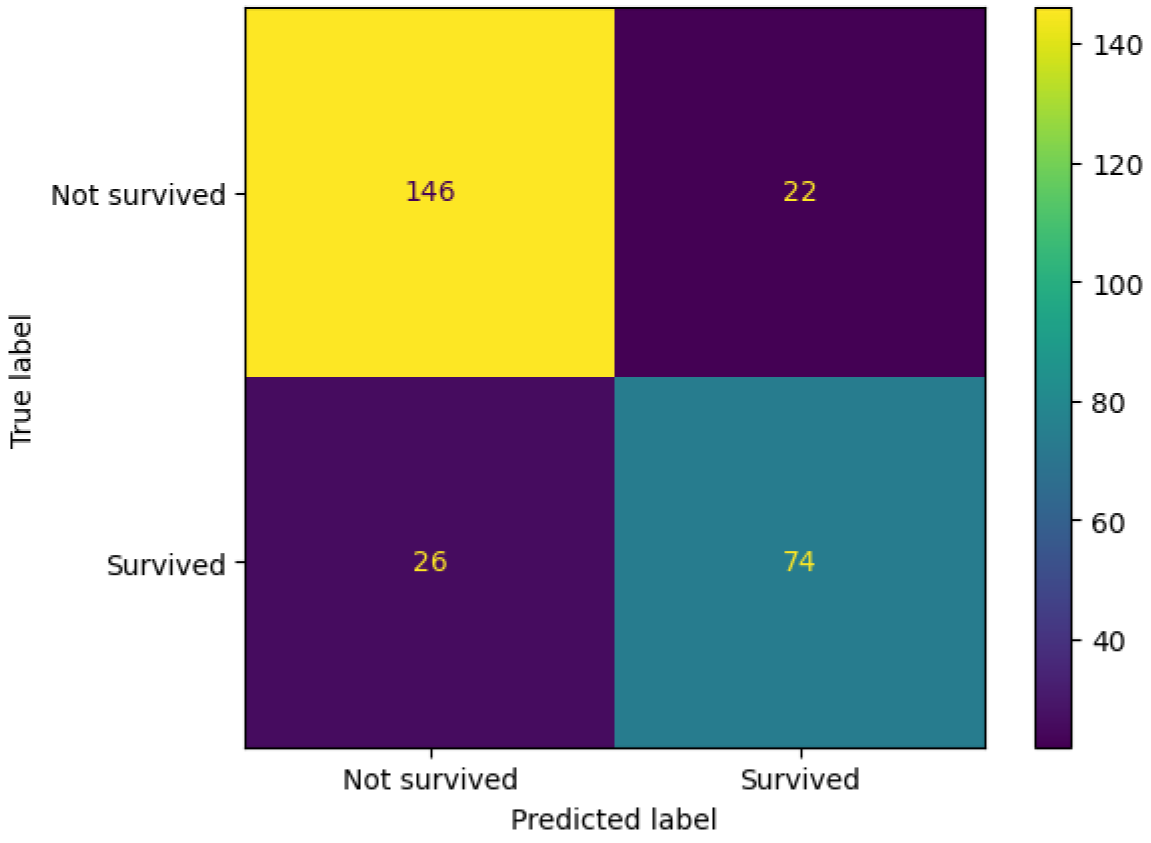
In [6]:
```python
from sklearn import tree

titanic_clf = tree.DecisionTreeClassifier(criterion='entropy', max_depth=3)
titanic_clf = titanic_clf.fit(X_train, y_train)

# Predict on test data
titanic_predictions = titanic_clf.predict(X_test)
titanic_predictions
```

Out[6]:
```
array([0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1,
       0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0,
       1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1,
       1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1,
       0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
       0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0,
       1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0,
       1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1,
       0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1,
       0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1,
       0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 1, 0], dtype=int64)
```

Plot confusion matrix for titanic classifier

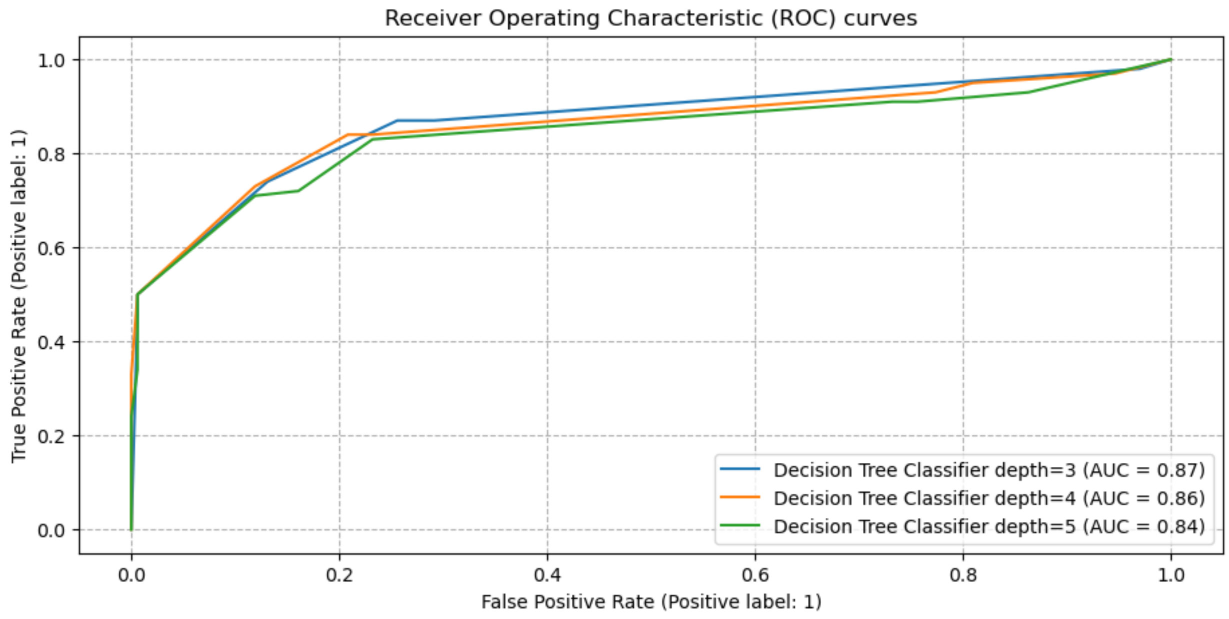
In [7]:
```python
classification_report_and_cm(y_test, titanic_predictions, ["Not survived", "Survived"]
```

```
Accuracy on test data is 0.82
F1 score on test data is 0.76
Precision Score on test data is 0.77
Recall score on test data is 0.74
              precision    recall  f1-score   support

           0       0.85      0.87      0.86       168
           1       0.77      0.74      0.76       100

    accuracy                           0.82       268
   macro avg       0.81      0.80      0.81       268
weighted avg       0.82      0.82      0.82       268
```



Decision tree performance on Titanic data with varying tree depth.

In [8]:
```
from sklearn.metrics import RocCurveDisplay

treeDepth = [3, 4, 5]
testAcc = []
trainAcc = []

fig, ax_roc = plt.subplots(1, 1, figsize=(11, 5))
ax_roc.grid(linestyle="--")

for k in treeDepth:
    clf = tree.DecisionTreeClassifier(criterion='entropy', max_depth=k)
    clf.fit(X_train, y_train)
    RocCurveDisplay.from_estimator(clf, X_test, y_test, ax=ax_roc, name="Decision Tree

ax_roc.set_title("Receiver Operating Characteristic (ROC) curves")
plt.legend()
plt.show()
```

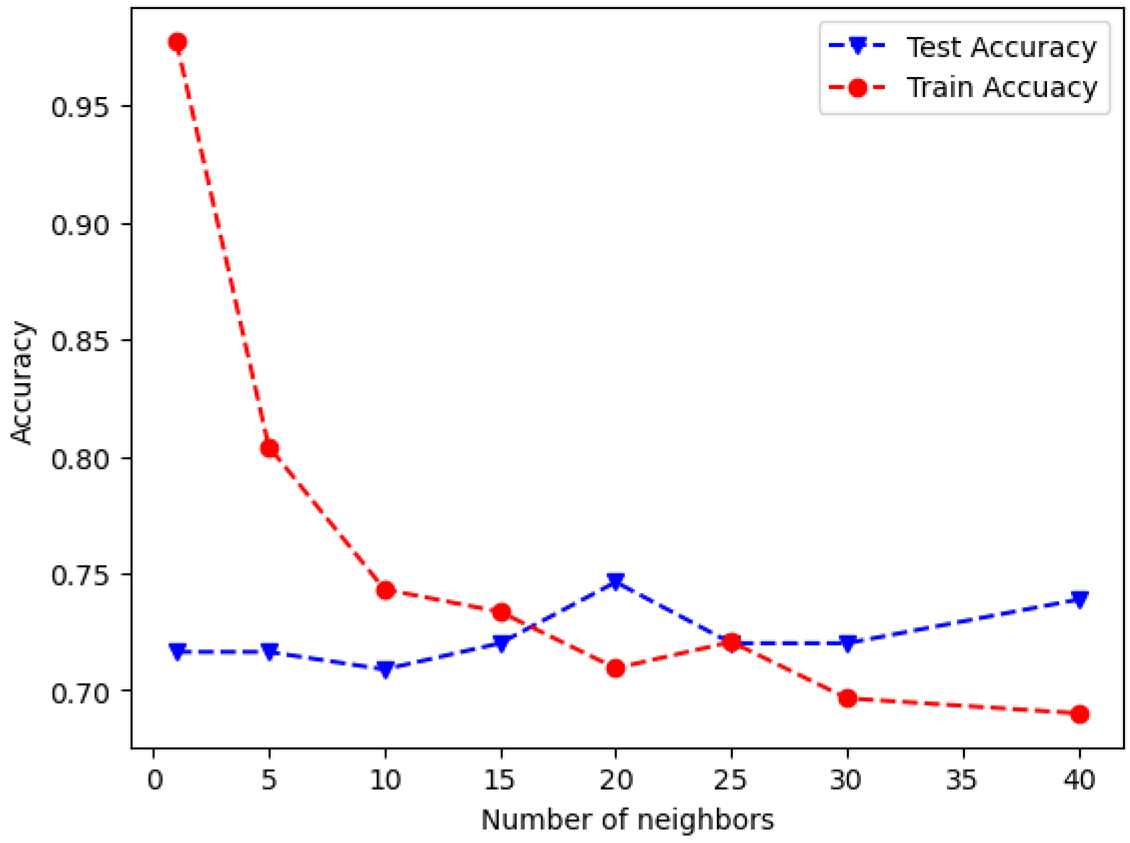


It can be seen that depth = 3, blue line with AUC=0.87 seems to be the best choice between 3, 4, and 5.

KNN Classifier on Titanic data

In [9]:
```
from sklearn.neighbors import KNeighborsClassifier

numNeighbors = [1, 5, 10, 15, 20, 25, 30, 40]
testAcc = []
trainAcc = []

for k in numNeighbors:
    clf = KNeighborsClassifier(n_neighbors=k, metric='minkowski', p=2)
    clf.fit(X_train, y_train)
    knn_pred = clf.predict(X_test)
    knn_pred_train = clf.predict(X_train)
    # print(knn_pred)
    testAcc.append(accuracy_score(y_test, knn_pred))
    trainAcc.append(accuracy_score(y_train,knn_pred_train))

plt.plot(numNeighbors, testAcc,'bv--',numNeighbors, trainAcc, 'ro--')
plt.legend(['Test Accuracy','Train Accuacy'])
plt.xlabel('Number of neighbors')
plt.ylabel('Accuracy')
print('Best accuracy score: %.3f' % max(testAcc))
```

```
Best accuracy score: 0.746
```



The optimal number of neighbors for the KNN classifer on Titanic Data is 20.

Conclusion on Classification experiments on Titanic dataset

Best DecisionTreeClassifier with (criterion='entropy', max_depth=3) obtains an Accuracy score of **0.82** on test dataset. Best KNeighborsClassifier with (n_neighbors=20, metric='minkowski', p=2) obtains an Accuracy score of 0.746 on test dataset.

In our experiments on Titanic dataset DecisionTreeClassifier performed better than KNeighborsClassifier.

(B) Classification using the Churn Data set

Load and preprocess data

Here, we are taking the churn dataset and applying preprocessing and converting rows with String data type into integer types. Aterwards, we separate the dataset into train and testing and measure the accuracy of the trained models. Using a classification decision tree, we will be able to determine what features are uncessary and what are needed.

In [10]:
```
path = "./"
# Open .csv
```

```
filename_read = os.path.join(path,"Churn_Modelling.csv")
churn_Data = pd.read_csv(filename_read, na_values=['NA','?'])
churn_Data.head(10)
```

Out[10]:

	RowNumber	CustomerId	Surname	CreditScore	Geography	Gender	Age	Tenure	Balance	Nu
0	1	15634602	Hargrave	619	France	Female	42	2	0.00	
1	2	15647311	Hill	608	Spain	Female	41	1	83807.86	
2	3	15619304	Onio	502	France	Female	42	8	159660.80	
3	4	15701354	Boni	699	France	Female	39	1	0.00	
4	5	15737888	Mitchell	850	Spain	Female	43	2	125510.82	
5	6	15574012	Chu	645	Spain	Male	44	8	113755.78	
6	7	15592531	Bartlett	822	France	Male	50	7	0.00	
7	8	15656148	Obinna	376	Germany	Female	29	4	115046.74	
8	9	15792365	He	501	France	Male	44	4	142051.07	
9	10	15592389	H?	684	France	Male	27	2	134603.88	

Look for duplicates and missing values

In [11]:
```
dups = churn_Data.duplicated()
print('Number of duplicate rows = %d' % (dups.sum()))
print('Number of empty values in a row = %d' % (churn_Data.isnull().any(axis = 1).sum(
```

```
Number of duplicate rows = 0
Number of empty values in a row = 0
```

Feature selection

Here, we are able to identify that the classification label for this dataset uses 'Exited' columns, signifying if the user has left or not.
We then take all the features from the row, dropping the unnecessary columns for our classfication (which are the RowNumber, CustomerId, and Surname) as they provide no signal as to if the user has exited or not.

In [12]:
```
# Drop unnecessary features from the dataset.
churn_Data = churn_Data.drop(['RowNumber', 'CustomerId', 'Surname'], axis=1)
encode_text_index(churn_Data, 'Geography')
encode_text_index(churn_Data, 'Gender')
churn_Data.head(10)
```

Out[12]:

	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMe
0	619	0	0	42	2	0.00	1	1	
1	608	2	0	41	1	83807.86	1	0	
2	502	0	0	42	8	159660.80	3	1	
3	699	0	0	39	1	0.00	2	0	
4	850	2	0	43	2	125510.82	1	1	
5	645	2	1	44	8	113755.78	2	1	
6	822	0	1	50	7	0.00	2	1	
7	376	1	0	29	4	115046.74	4	1	
8	501	0	1	44	4	142051.07	2	0	
9	684	0	1	27	2	134603.88	1	1	

In [13]:
```
churn_Data.dtypes
```

Out[13]:
```
CreditScore          int64
Geography            int32
Gender               int32
Age                  int64
Tenure               int64
Balance            float64
NumOfProducts        int64
HasCrCard            int64
IsActiveMember       int64
EstimatedSalary    float64
Exited               int64
dtype: object
```

Plot Correlation Matrix for Churn Data attributes

We can observe that 'Age' is the most correlated with 'Exited'

In [14]:
```
corr = churn_Data.corr()
corr.style.background_gradient(cmap='coolwarm')
```

Out[14]:

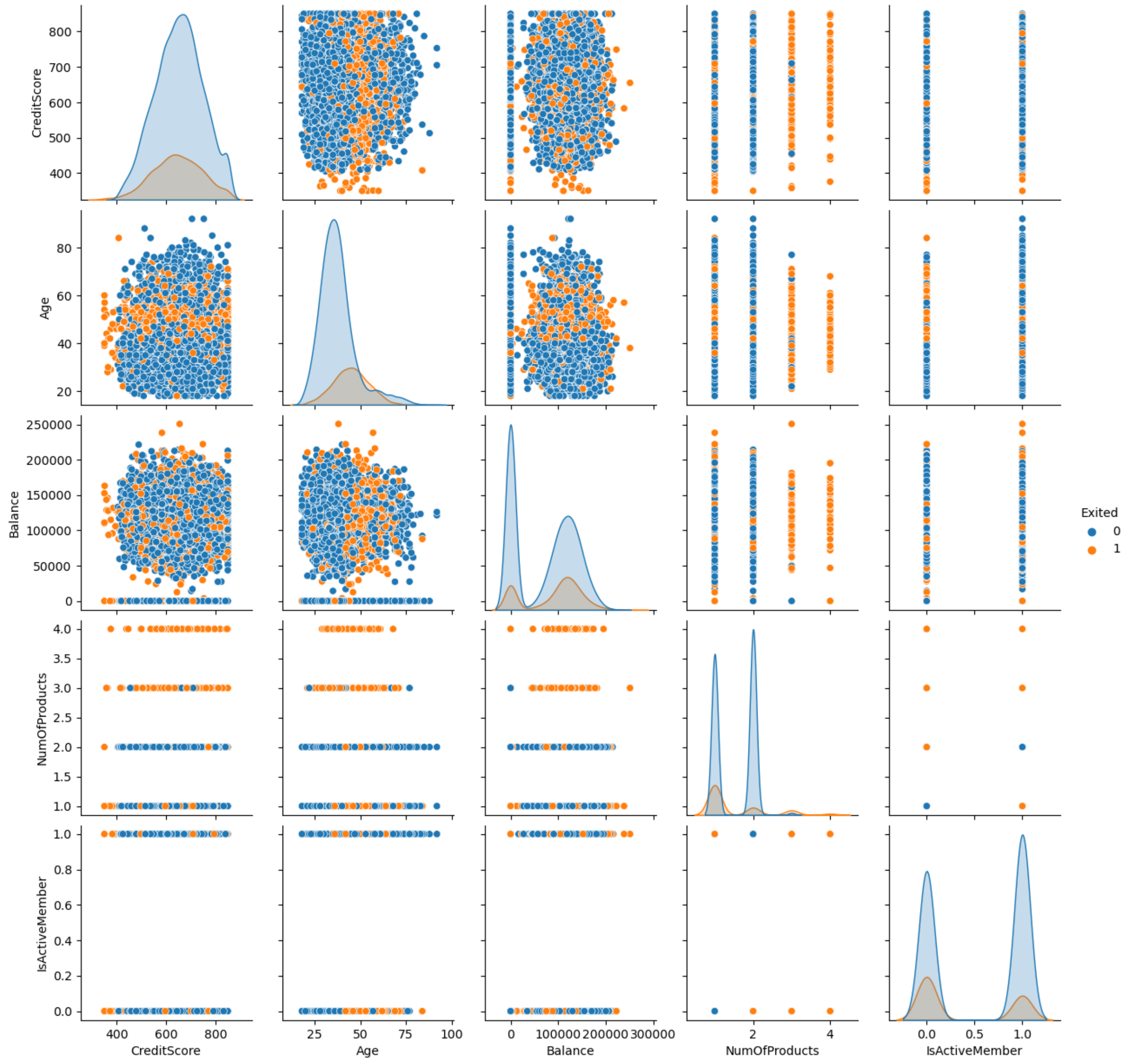
	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts
CreditScore	1.000000	0.007888	-0.002857	-0.003965	0.000842	0.006268	0.012238
Geography	0.007888	1.000000	0.004719	0.022812	0.003739	0.069408	0.003972
Gender	-0.002857	0.004719	1.000000	-0.027544	0.014733	0.012087	-0.021859
Age	-0.003965	0.022812	-0.027544	1.000000	-0.009997	0.028308	-0.030680
Tenure	0.000842	0.003739	0.014733	-0.009997	1.000000	-0.012254	0.013444
Balance	0.006268	0.069408	0.012087	0.028308	-0.012254	1.000000	-0.304180
NumOfProducts	0.012238	0.003972	-0.021859	-0.030680	0.013444	-0.304180	1.000000
HasCrCard	-0.005458	-0.008523	0.005766	-0.011721	0.022583	-0.014858	0.003183
IsActiveMember	0.025651	0.006724	0.022544	0.085472	-0.028362	-0.010084	0.009612
EstimatedSalary	-0.001384	-0.001369	-0.008112	-0.007201	0.007784	0.012797	0.014204
Exited	-0.027094	0.035943	-0.106512	0.285323	-0.014001	0.118533	-0.047820

Pair plot of some features to see relationships between features

We can see that for example ('NumOfProducts', 'CreditScore') pair plot shows a strong separation of the targets.

In [15]:
```
sns.pairplot(churn_Data[['CreditScore', 'Age', 'Balance', 'NumOfProducts', 'IsActiveMe
            hue='Exited', diag_kind="auto")
```

Out[15]:
```
<seaborn.axisgrid.PairGrid at 0x1e4160addd0>
```



Here, we are using all the features given to us to create our decision tree.

```
In [16]: from sklearn.tree import DecisionTreeClassifier # Import Decision Tree Classifier
         from sklearn.model_selection import train_test_split # Import train_test_split functic
         from sklearn import metrics #Import scikit-learn metrics module for accuracy calculati

         feature_cols = ['CreditScore', 'Geography', 'Gender', 'Age', 'Tenure', 'Balance', 'Num

         x = churn_Data[feature_cols]
         y = churn_Data.Exited
```

```
In [17]: # Split dataset into training set and test set
         x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.3, random_state=
```

Training one Decision Tree using all features

Building decision trees involves making a predictive model to recursively divide feature spaces into smaller and smaller sections. Decision trees improves our ability to understand how the model arrived at specific classifications by mimicking the human decision-making patterns.

Since they can work on both numerical and categorical features without needing much data preprocessing they are extremely convenient to use. Decision trees also can show a measure of feature importance, which helps us identify which features are more influential in making decisions, allowing us to tweak weights and aides in feature selection. Furthermore, they can identify non-linear relationships in the data, since they do not make any assumptions about the linearity of the data. This makes them useful for datasets with intricate relationships between features and the target variable.

Decision trees have a tendency to overfit, especially when they "grow too deep" or if the dataset is noisy. They are also unstable, with massive changes in the tree if the dataset changes. They also have tendency to favor features with a lot of unique values because they can create more branches which may result in more specific rules.

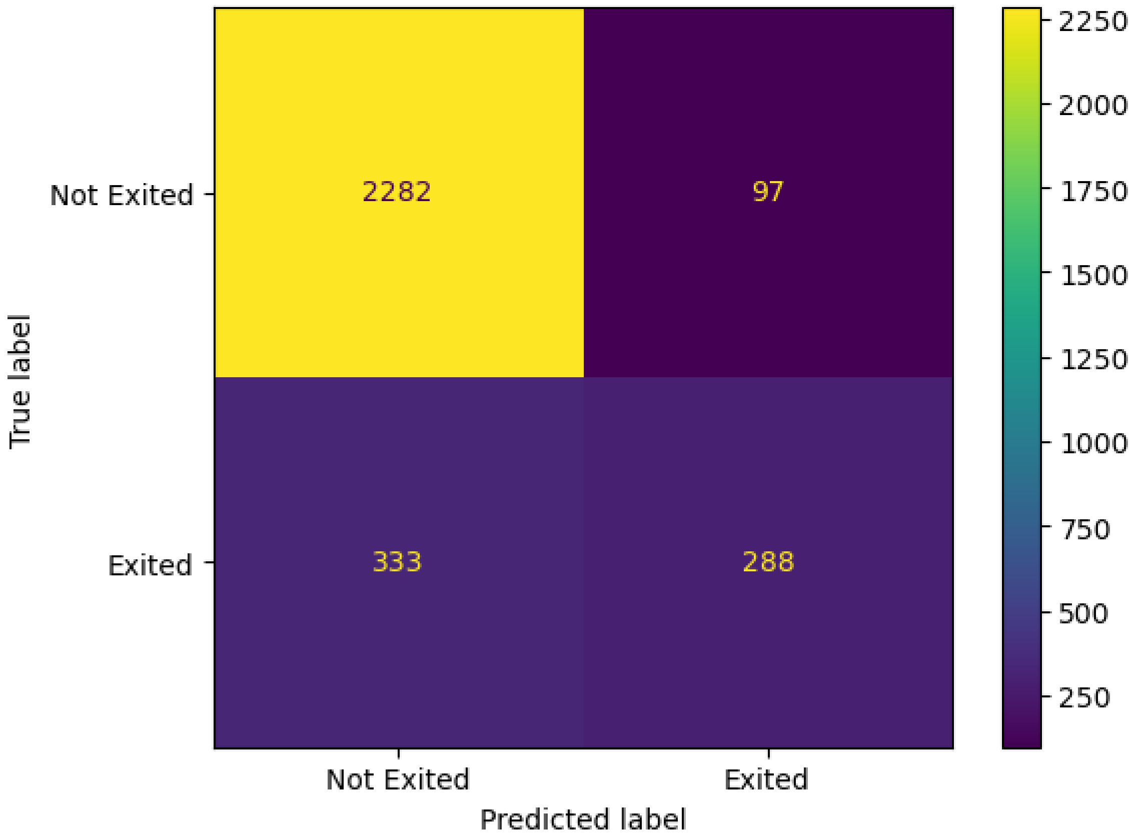
In [18]:
```
# Create Decision Tree classifer object
clf = DecisionTreeClassifier(criterion="entropy", max_depth=5)
# Train Decision Tree Classifer
clf = clf.fit(x_train, y_train)
#Predict the response for test dataset
y_pred = clf.predict(x_test)
```

Evaluating Model accuracy

As we can see here, using all of the features resulted in us have a relatively high accuracy which make sense as we are using all of the provided features to identify between Exited (1) or Not (0) users.

In [19]:
```
classification_report_and_cm(y_test, y_pred, ['Not Exited', 'Exited'])
```

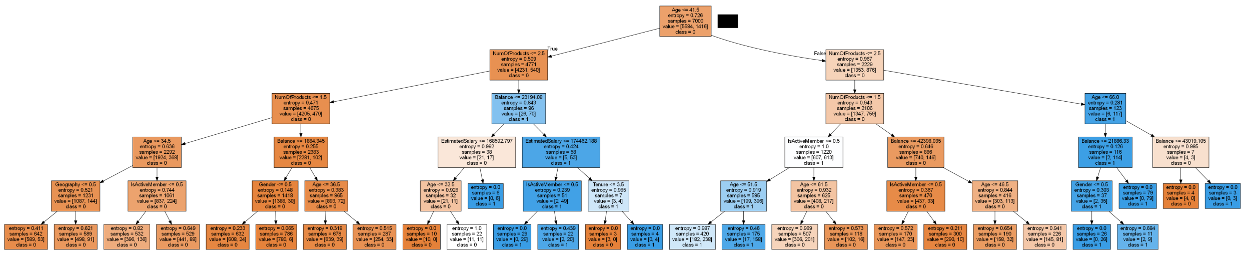
```
Accuracy on test data is 0.86
F1 score on test data is 0.57
Precision Score on test data is 0.75
Recall score on test data is 0.46
              precision    recall  f1-score   support

           0       0.87      0.96      0.91      2379
           1       0.75      0.46      0.57       621

    accuracy                           0.86      3000
   macro avg       0.81      0.71      0.74      3000
weighted avg       0.85      0.86      0.84      3000
```



Using graphviz, we are able to create the decision tree image of all the features from the churn dataset. As seen below, the tree has an imbalance and is having trouble classifying which account has exited or not (1 or 0).

In [20]:
```
import pydotplus
from IPython.display import Image
from sklearn.tree import export_graphviz

dot_data = export_graphviz(clf, feature_names=x.columns, class_names=['0','1'], fillec
                                out_file=None)
graph = pydotplus.graph_from_dot_data(dot_data)
Image(graph.create_png())
```

Out[20]:



Feature selection (continued): Training one Decision Tree with less features, selected based on previous insights

The code below sets up the decision tree classifier, trains it using feature columns: 'CreditScore', 'Age', 'Balance', 'NumOfProducts', and uses 'Exited' as the target variable.

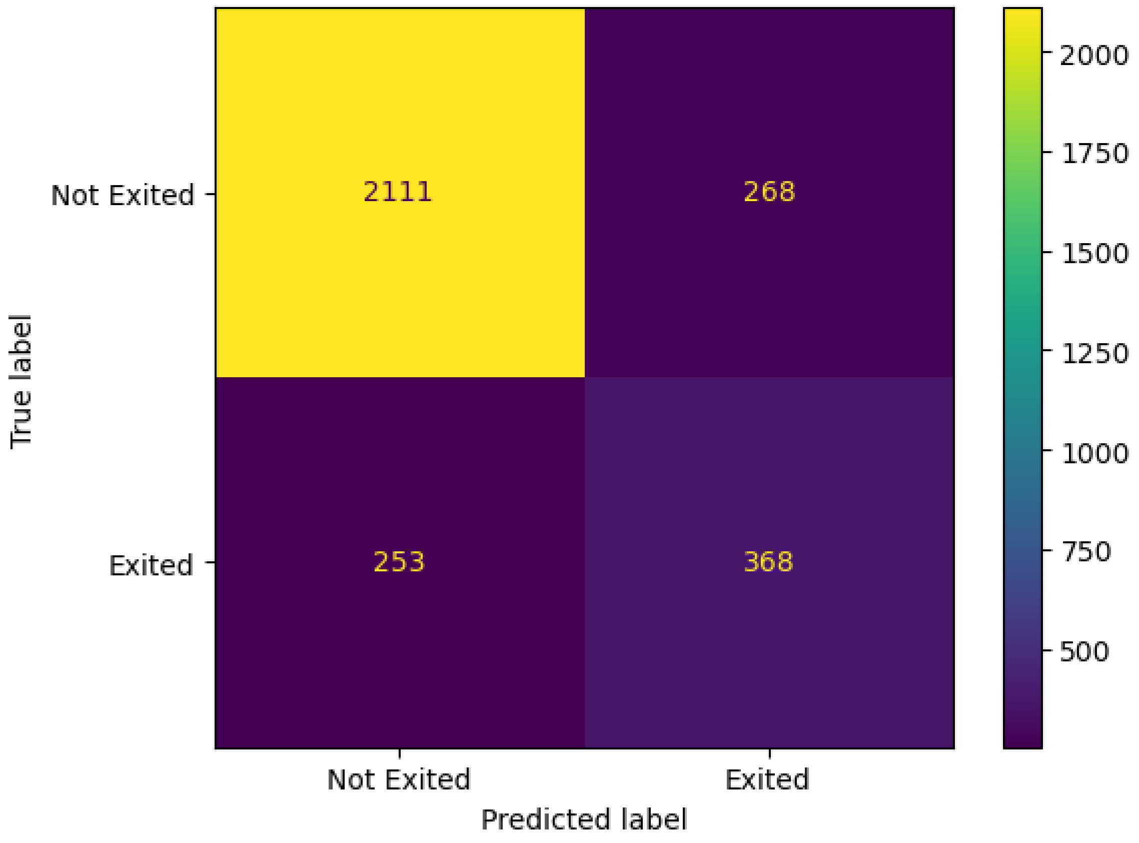
It is shown below that they can be used to train a classifier that achieves 0.83 accuracy when predicting if an account has Exited or Not (1 or 0).

In [21]:
```
feature_cols = ['CreditScore', 'Age', 'Balance', 'NumOfProducts']

x = churn_Data[feature_cols]
y = churn_Data.Exited

# Split dataset into training set and test set
x_train_sel, x_test_sel, y_train_sel, y_test_sel = train_test_split(x, y, test_size=0.
```

Train DecisionTreeClassifier and display the accuracy achieved by the decision tree model in predicting the 'Exited' variable.

In [22]:
```
# Create Decision Tree classifer object
clf = DecisionTreeClassifier(criterion="entropy", max_depth=4)
# Train Decision Tree Classifer
clf = clf.fit(x_train_sel, y_train_sel)
#Predict the response for test dataset
y_pred_sel = clf.predict(x_test_sel)

# Model Accuracy, how often is the classifier correct?
classification_report_and_cm(y_test_sel, y_pred_sel, ['Not Exited', 'Exited'])
```

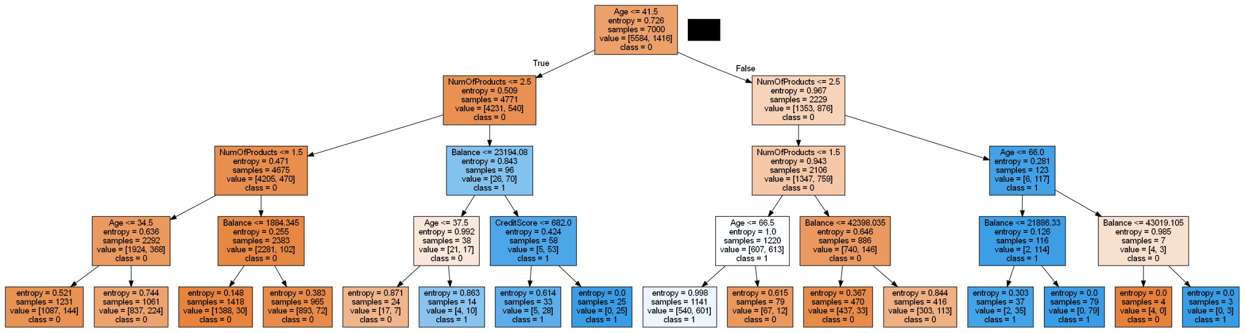
```
Accuracy on test data is 0.83
F1 score on test data is 0.59
Precision Score on test data is 0.58
Recall score on test data is 0.59
              precision    recall  f1-score   support

           0       0.89      0.89      0.89      2379
           1       0.58      0.59      0.59       621

    accuracy                           0.83      3000
   macro avg       0.74      0.74      0.74      3000
weighted avg       0.83      0.83      0.83      3000
```



We can see that the Decision tree classifier retained most of the Accuracy (0.83 vs 0.86) by using a smaller number of features.

```
In [23]: import pydotplus
         from IPython.display import Image
         from sklearn.tree import export_graphviz

         dot_data = export_graphviz(clf, feature_names=x.columns, class_names=['0','1'], fillec
                                    out_file=None)
         graph = pydotplus.graph_from_dot_data(dot_data)
         Image(graph.create_png())
```

Out[23]:


Age <= 41.5 entropy = 0.726 samples = 7000 value = [5584, 1416] class = 0
True
False
NumOfProducts <= 2.5 entropy = 0.509 samples = 4771 value = [4231, 540] class = 0
NumOfProducts <= 2.5 entropy = 0.967 samples = 2229 value = [1353, 876] class = 0
NumOfProducts <= 1.5 entropy = 0.471 samples = 4675 value = [4205, 470] class = 0
Balance <= 23194.08 entropy = 0.843 samples = 96 value = [26, 70] class = 1
NumOfProducts <= 1.5 entropy = 0.943 samples = 2106 value = [1347, 759] class = 0
Age <= 66.0 entropy = 0.281 samples = 123 value = [6, 117] class = 1
Age <= 34.5 entropy = 0.636 samples = 2292 value = [1924, 368] class = 0
Balance <= 1884.345 entropy = 0.255 samples = 2383 value = [2281, 102] class = 0
Age <= 37.5 entropy = 0.992 samples = 38 value = [21, 17] class = 0
CreditScore <= 682.0 entropy = 0.424 samples = 58 value = [5, 53] class = 1
Age <= 66.5 entropy = 1.0 samples = 1220 value = [607, 613] class = 1
Balance <= 42398.035 entropy = 0.646 samples = 886 value = [740, 146] class = 0
Balance <= 21886.33 entropy = 0.126 samples = 116 value = [2, 114] class = 1
Balance <= 43019.105 entropy = 0.985 samples = 7 value = [4, 3] class = 0
entropy = 0.521 samples = 1231 value = [1087, 144] class = 0
entropy = 0.744 samples = 1061 value = [837, 224] class = 0
entropy = 0.148 samples = 1418 value = [1388, 30] class = 0
entropy = 0.383 samples = 965 value = [893, 72] class = 0
entropy = 0.871 samples = 24 value = [17, 7] class = 0
entropy = 0.863 samples = 14 value = [4, 10] class = 1
entropy = 0.614 samples = 33 value = [5, 28] class = 1
entropy = 0.0 samples = 25 value = [0, 25] class = 1
entropy = 0.998 samples = 1141 value = [540, 601] class = 1
entropy = 0.615 samples = 79 value = [67, 12] class = 0
entropy = 0.367 samples = 470 value = [437, 33] class = 0
entropy = 0.844 samples = 416 value = [303, 113] class = 0
entropy = 0.303 samples = 37 value = [2, 35] class = 1
entropy = 0.0 samples = 79 value = [0, 79] class = 1
entropy = 0.0 samples = 4 value = [4, 0] class = 0
entropy = 0.0 samples = 3 value = [0, 3] class = 1

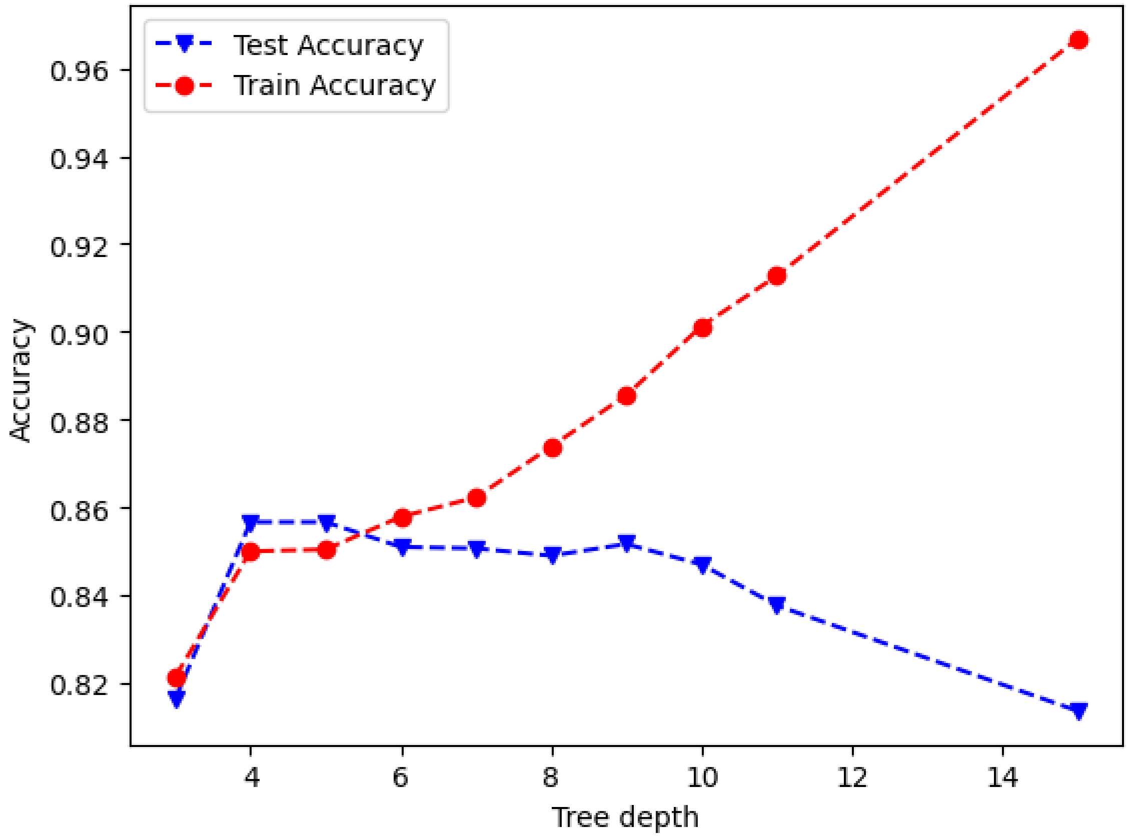
Decision Tree performance when varying tree depth

By looking at the Accuracy vs max_depth plot below we observe that best depth is around [4, 5, 6].
With larger max_depth the DecisionTreeClassifier starts to have lower accuracy on test.

In [24]:
```
treeDepth = [3, 4, 5, 6, 7, 8, 9, 10, 11, 15]
treeTestAcc = []
treeTrainAcc = []

for k in treeDepth:
    clf = tree.DecisionTreeClassifier(criterion='entropy', max_depth=k)
    clf.fit(x_train, y_train)
    tree_pred = clf.predict(x_test)
    tree_pred_train = clf.predict(x_train)
    treeTestAcc.append(accuracy_score(y_test, tree_pred))
    treeTrainAcc.append(accuracy_score(y_train,tree_pred_train))
     #display accuracy of test
    print ("Accuracy for k=%d: "%k,  accuracy_score(y_test, tree_pred))

#display a plot
plt.plot(treeDepth, treeTestAcc,'bv--',treeDepth,treeTrainAcc,'ro--')
plt.legend(['Test Accuracy','Train Accuracy'])
plt.xlabel('Tree depth')
plt.ylabel('Accuracy')
```

```
Accuracy for k=3:  0.8163333333333334
Accuracy for k=4:  0.8566666666666667
Accuracy for k=5:  0.8566666666666667
Accuracy for k=6:  0.851
Accuracy for k=7:  0.8506666666666667
Accuracy for k=8:  0.849
Accuracy for k=9:  0.8516666666666667
Accuracy for k=10:  0.847
Accuracy for k=11:  0.8376666666666667
Accuracy for k=15:  0.8136666666666666
```

Out[24]:
```
Text(0, 0.5, 'Accuracy')
```



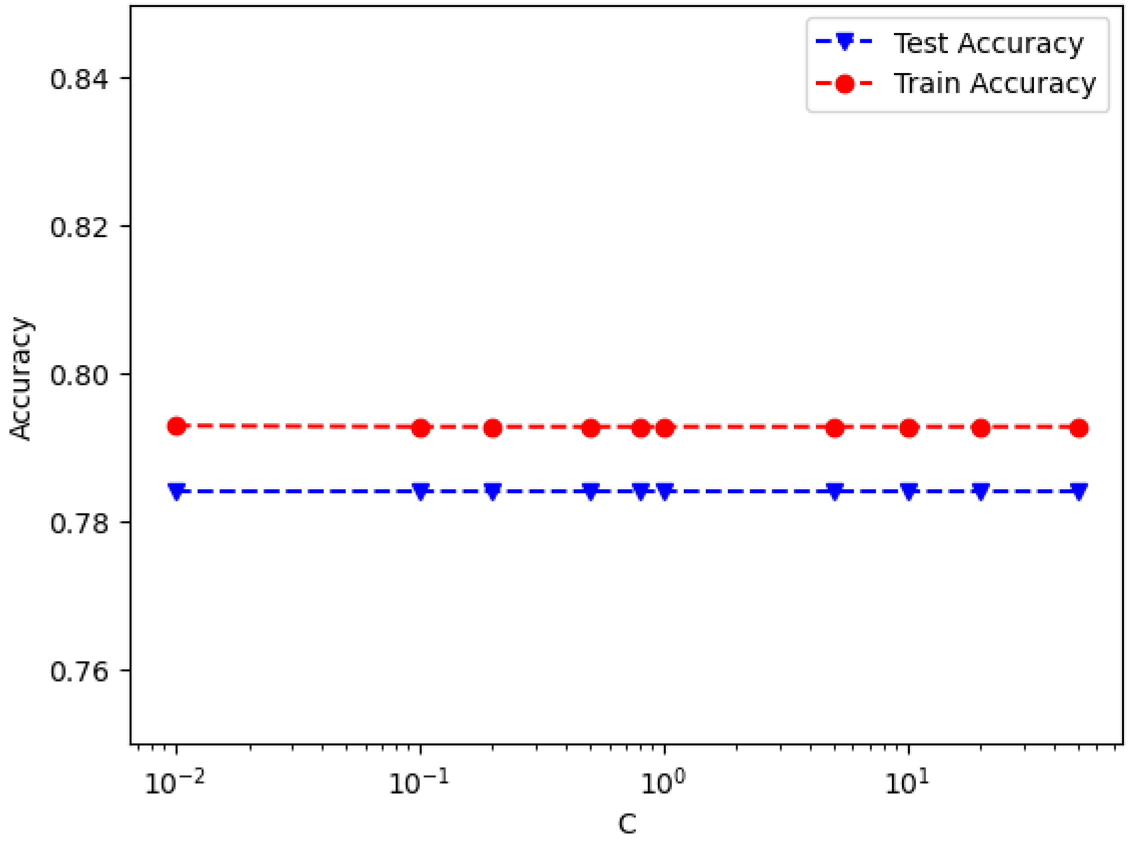
Logistic Regression (Logit)

We apply logistic regression of varying strength to the train and test. We use a range of numbers for C, starting off from a smaller value for strong regularization to greater values. This is to ensure that we are able to obtain the accuracy of our train and test base off the varying regularization strengths that the test and train goes through.

In [25]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

#C = Inverse of regularization strength;  smaller values specify stronger regularizati
C = [0.01, 0.1, 0.2, 0.5, 0.8, 1, 5, 10, 20, 50]

#finding test accuracy and train accuracy
LRtestAcc = []
LRtrainAcc = []

#for loop that applies logistic regression at different C levels and stores values to
for param in C:
    clf = LogisticRegression(C=param)
    clf.fit(x_train,y_train)
    log_reg_pred = clf.predict(x_test)
    log_reg_pred_train = clf.predict(x_train)
    # print(log_reg_pred)
    LRtestAcc.append(accuracy_score(y_test, log_reg_pred))
    LRtrainAcc.append(accuracy_score(y_train,log_reg_pred_train))
     #display accuracy of test
    print ("Accuracy for C=%.2f: "%param,  accuracy_score(y_test, log_reg_pred))

#display a plot
plt.plot(C, LRtestAcc,'bv--',C,LRtrainAcc,'ro--')
plt.legend(['Test Accuracy','Train Accuracy'])
plt.ylim(0.75, 0.85)
plt.xlabel('C')
plt.xscale('log')
plt.ylabel('Accuracy')
```

```
Accuracy for C=0.01:  0.784
Accuracy for C=0.10:  0.784
Accuracy for C=0.20:  0.784
Accuracy for C=0.50:  0.784
Accuracy for C=0.80:  0.784
Accuracy for C=1.00:  0.784
Accuracy for C=5.00:  0.784
Accuracy for C=10.00:  0.784
Accuracy for C=20.00:  0.784
Accuracy for C=50.00:  0.784
```

Out[25]:
```
Text(0, 0.5, 'Accuracy')
```



Naive Bayes

In [26]:
```
from sklearn.naive_bayes import GaussianNB

clf_NB = GaussianNB()
clf_NB.fit(x_train, y_train)
NB_pred = clf_NB.predict(x_test)

print('Accuracy on test data is %.2f' % (accuracy_score(y_test, NB_pred)))
```

```
Accuracy on test data is 0.78
```

Support Vector Machine (SVM) Classifer with Linear kernel

C = determines the strength of regularization (regularization is inversely proportional to C)

We apply normalization for SVM training.

In [27]:
```
from sklearn import svm
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler, MinMaxScaler

svmClassifier = make_pipeline(StandardScaler(), svm.SVC(kernel='linear', C=1))
svmClassifier.fit(x_train, y_train)
y_svmPred = svmClassifier.predict(x_test)

print('Accuracy on test data is %.2f' % (accuracy_score(y_test, y_svmPred)))
```

```
Accuracy on test data is 0.79
```

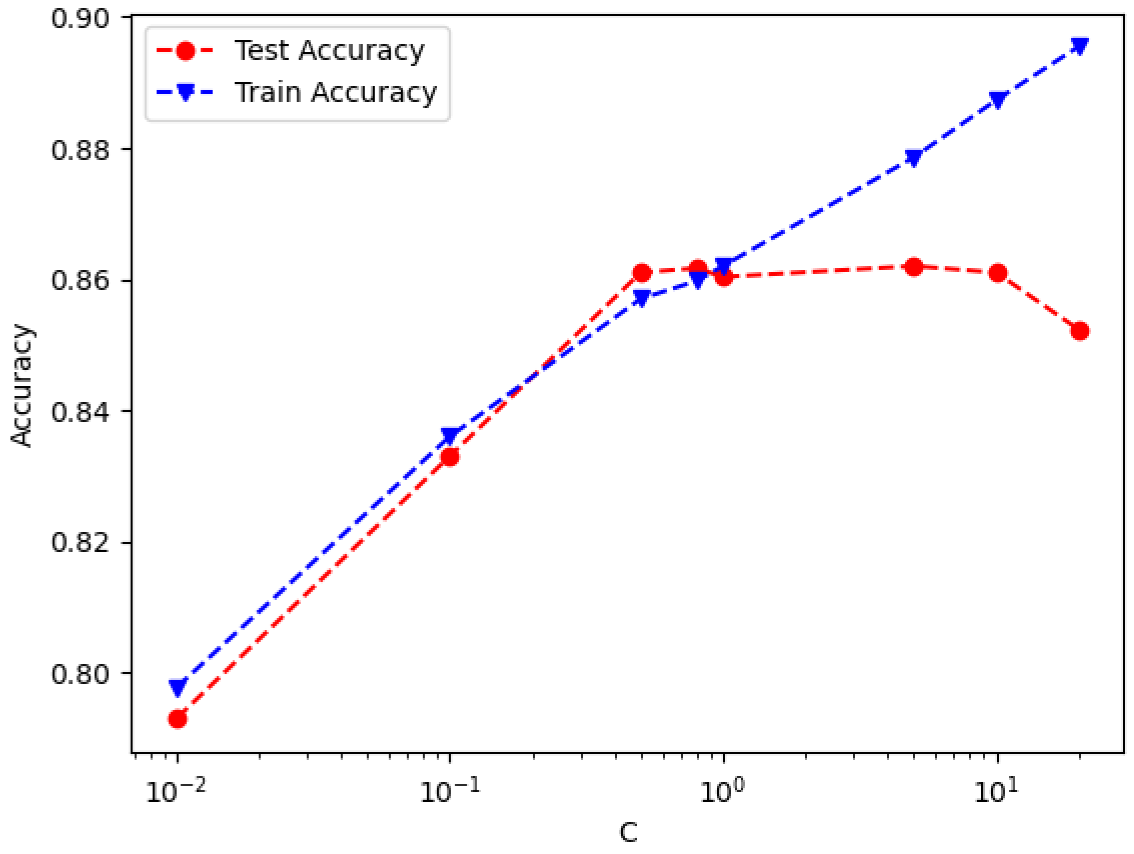
Support Vector Machine (SVM) Classifer with Non-linear (RBF) kernel

This code block trains the non-linear kernel SVM classifier with different regularization strengths C
After running, a graph is generated for the test and train accuracy depending on the C parameter.

In [28]:
```
from sklearn.svm import SVC
svc = SVC()

C = [0.01, 0.1, 0.5, 0.8, 1, 5, 10, 20]


SVMLtestAcc = []
SVMLtrainAcc = []

for param in C:
    # clf = SVC(C=param,kernel='rbf', gamma='auto')
    clf = make_pipeline(StandardScaler(), SVC(kernel='rbf', gamma='auto', C=param))
    clf.fit(x_train,y_train)
    svml_pred = clf.predict(x_test)
    svml_pred_train = clf.predict(x_train)
    # print(svml_pred)
    print ("Accuracy on test dataset for C=%.2f: "%param,  accuracy_score(y_test, svml
    SVMLtestAcc.append(accuracy_score(y_test, svml_pred))
    SVMLtrainAcc.append(accuracy_score(y_train,svml_pred_train))

plt.plot(C, SVMLtestAcc,'ro--', C,SVMLtrainAcc,'bv--')
plt.legend(['Test Accuracy','Train Accuracy'])
plt.xlabel('C')
plt.xscale('log')
# plt.xlim(0, 1)
plt.ylabel('Accuracy')
```

```
Accuracy on test dataset for C=0.01:  0.793
Accuracy on test dataset for C=0.10:  0.833
Accuracy on test dataset for C=0.50:  0.861
Accuracy on test dataset for C=0.80:  0.8616666666666667
Accuracy on test dataset for C=1.00:  0.8603333333333333
Accuracy on test dataset for C=5.00:  0.862
Accuracy on test dataset for C=10.00:  0.861
Accuracy on test dataset for C=20.00:  0.8523333333333334
```

Out[28]:
```
Text(0, 0.5, 'Accuracy')
```



We observe that SVM with radial basis kernel overfits on the training set when regularization is not strong enough. For sklearn.svm.SVC the strength of the regularization is inversely proportional to C. In our tests the best value for C seems to be between 0.5 and 1.

K-Nearest Neighbor (KNN) Classifiers

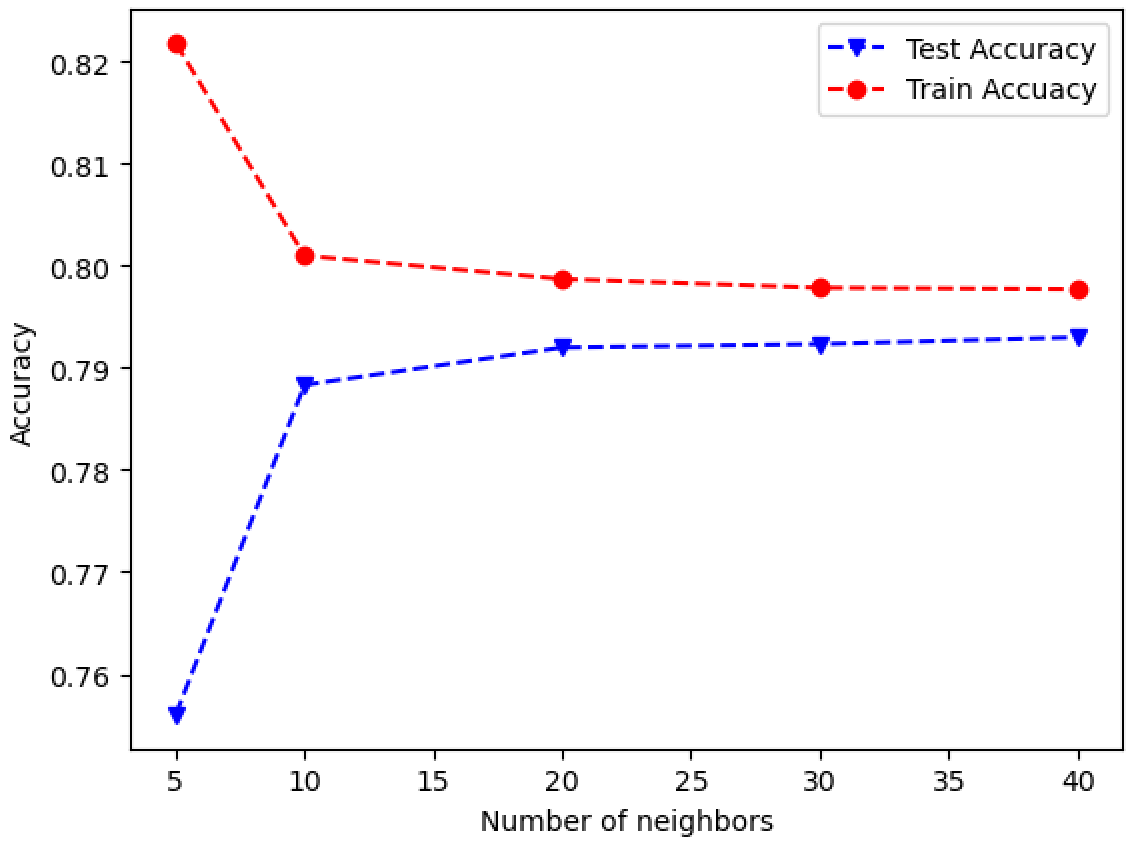
In [29]:

```
from sklearn.neighbors import KNeighborsClassifier

numNeighbors = [5, 10, 20, 30, 40]
testAcc = []
trainAcc = []

for k in numNeighbors:
    clf = KNeighborsClassifier(n_neighbors=k, metric='minkowski', p=2)
    clf.fit(x_train, y_train)
    knn_pred = clf.predict(x_test)
    knn_pred_train = clf.predict(x_train)
    # print(knn_pred)
    testAcc.append(accuracy_score(y_test, knn_pred))
    trainAcc.append(accuracy_score(y_train,knn_pred_train))
    print('Accuracy on test data using k=%i is %.2f' % (k, accuracy_score(y_test, knn_
    # print('Accuracy on train data using k=%i is %.2f' % (k, accuracy_score(y_train,

plt.plot(numNeighbors, testAcc,'bv--',numNeighbors, trainAcc, 'ro--')
plt.legend(['Test Accuracy','Train Accuacy'])
plt.xlabel('Number of neighbors')
plt.ylabel('Accuracy')
```

```
Accuracy on test data using k=5 is 0.76
Accuracy on test data using k=10 is 0.79
Accuracy on test data using k=20 is 0.79
Accuracy on test data using k=30 is 0.79
Accuracy on test data using k=40 is 0.79
```

Out[29]:

```
Text(0, 0.5, 'Accuracy')
```



The K-value that seems to be an accurate representation of the model is when k >= 30 as we can see on the graph. When comparing the two lines, training and testing, we see that the model stops and stay on a constant value on the graph 0.79 and 0.8. Upon analyizing this, we can tell that the accuracy of the model using the KNN technique is around 0.79 ~ 0.8.

Conclusion on Classification using the Churn Data set

We experimented with DecisionTreeClassifier, LogisticRegression, NaiveBayes, SVM with both linear and non-linear kernels and lastly with KNNClassifier.
The best classifiers in our experiments on Churn Data set are:
SVM with 'rbf' kernel (C=5)
DecisionTreeClassifier with max_depth=5

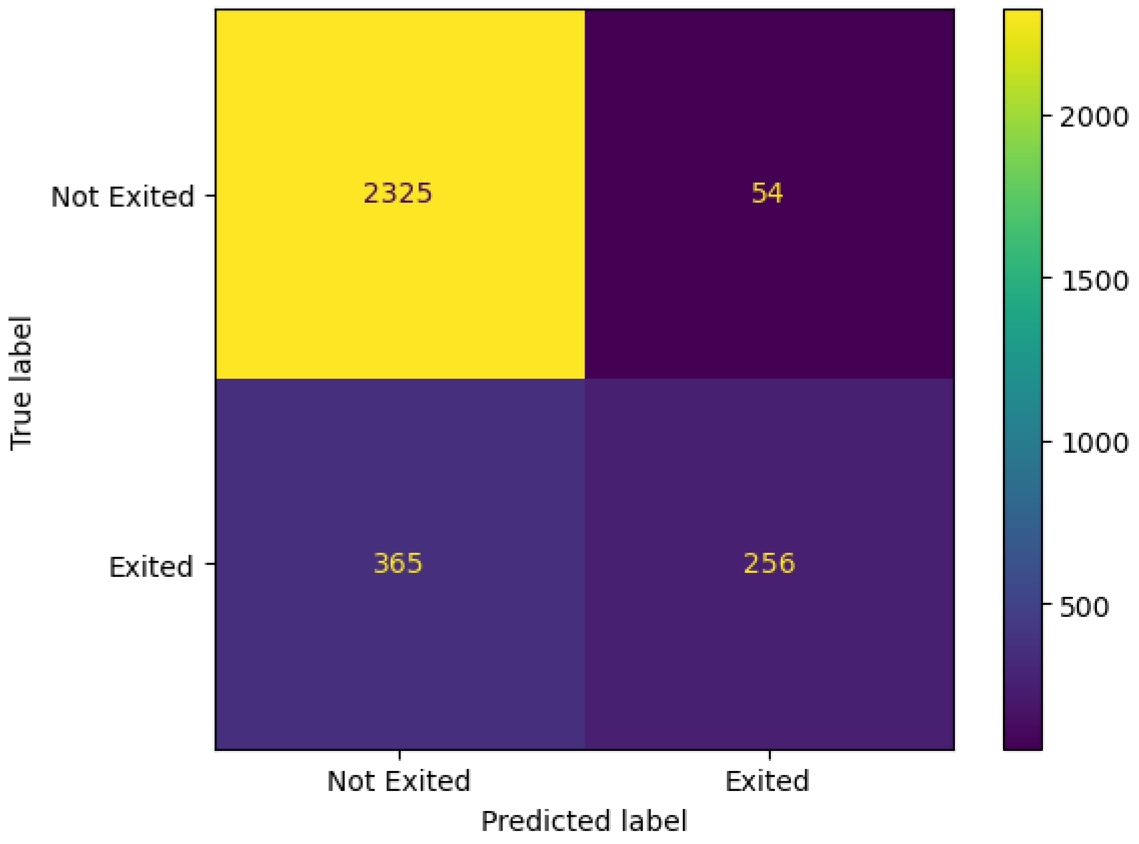
Both obtain 0.86 accuracy on the split test set.

In [30]:

```
svmClassifierRbf = make_pipeline(StandardScaler(), svm.SVC(kernel='rbf', gamma='auto',
svmClassifierRbf.fit(x_train, y_train)
```

```
svmRbfPred = svmClassifierRbf.predict(x_test)
classification_report_and_cm(y_test, svmRbfPred, ['Not Exited', 'Exited'])
```

```
Accuracy on test data is 0.86
F1 score on test data is 0.55
Precision Score on test data is 0.83
Recall score on test data is 0.41
              precision    recall  f1-score   support

           0       0.86      0.98      0.92      2379
           1       0.83      0.41      0.55       621

    accuracy                           0.86      3000
   macro avg       0.85      0.69      0.73      3000
weighted avg       0.86      0.86      0.84      3000
```


Not Exited
Exited
True label
Not Exited
Exited
Predicted label