UFCFAS-15-2 Machine Learning Individual Assignment

Author: Amin Alhawary

Student ID: 21041123

1. Overview

This study uses the diabetes dataset and it will be comparing two different methods for use in this classification problem. The two methods are Support Vector Machines (SVM) and Ensembles. The study will cover the different steps to preparing data and optimising the algorithm, contrasting and comparing the different methodologies and techniques involved in solving the problem.

2. Prerequisites

The file diabetes.csv containing the dataset is required. The following Python libraries are also required:

1. pandas - to read and deal with data from the csv file.
2. numpy - to deal with arrays.
3. matplotlib - to plot and visualise data.
4. seaborn - to plot and visualise data.
5. sklearn - ML library with many built in models.
6. imblearn - library used for undersampling data.

3. The dataset

The dataset is the Pima Indians Diabetes Database from The National Institute of Diabetes and Digestive and Kidney Disease. Based on 8 specific diagnostic measurements present in the dataset, the dataset's goal is to diagnostically determine whether a patient has diabetes or not. There are 9 columns, with the frst 8 being the features and the last one being the outcome of the classification. The 8 features are: 1. Number of times pregnant 2. Plasma glucose concentration a 2 hours in an oral glucose tolerance test 3. Diastolic blood pressure (mm Hg) 4. Triceps skin fold thickness (mm) 5. 2-Hour serum insulin (mu U/ml) 6. Body mass index (weight in kg/(height in m)^2) 7. Diabetes pedigree function 8. Age (years) The outcome is represented by a 1 or 0, with 1 denoting positive for diabetes and 0 being negative.

4. Analysing the data

The data will be analysed in order to give us insight on how the data should be prepared to carry out the study. First, we have to import our dataset from csv using the pandas library.

In [ ]:
```
import pandas as pd

diabetes_data = pd.read_csv('diabetes.csv')
diabetes_data.head(5) # Preview
```

Out[ ]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	A
0	6	148	72	35	0	33.6	0.627	
1	1	85	66	29	0	26.6	0.351	
2	8	183	64	0	0	23.3	0.672	
3	1	89	66	23	94	28.1	0.167	
4	0	137	40	35	168	43.1	2.288	

The first step to analytics is finding out how much data we have, and how many records belong in each category.

In [ ]:
```
print("Total number of records:", str(len(diabetes_data.index)))
print("Number of positives:",sum(diabetes_data["Outcome"] == 1), "({}%)".format(ro
print("Number of negatives:",sum(diabetes_data["Outcome"] == 0), "({}%)".format(ro
```

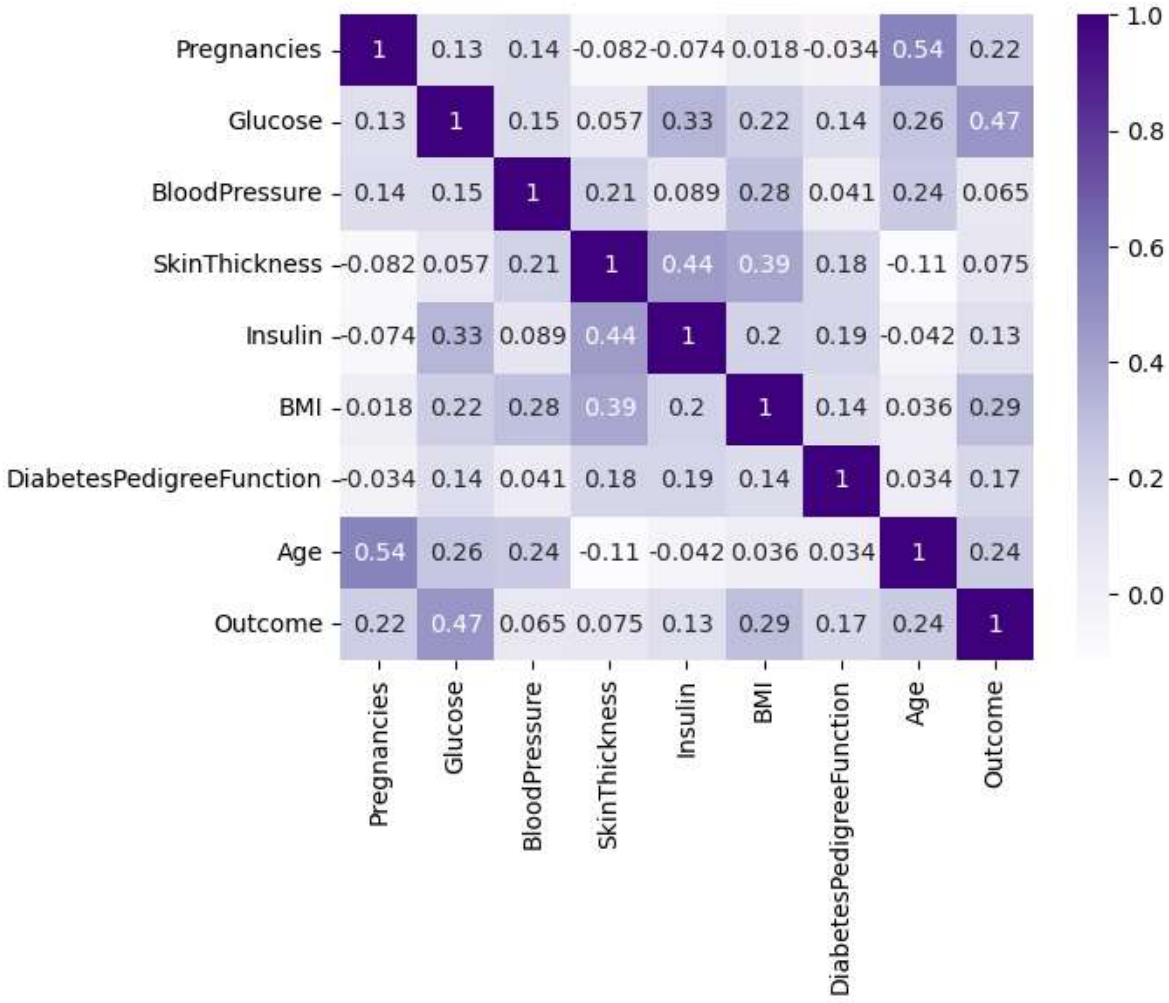
```
Total number of records: 768
Number of positives: 268 (35%)
Number of negatives: 500 (65%)
```

There is a slight umbalance between positive and negative. Unbalanced datasets can cause a bias in the classification, as they tend to make the algorithm biased towards the majaority class and thus perform poorly. In our case, it is not very significant but sampling the data could be key to getting a higher accuracy.

We should check how different features relate to the outcome. For that we use a heatmap showing the Pearson Correlation Coeffecient (r). r is a widely used measure of linear correlation between two variables. It ranges from -1 to 1 and reflects the direction and strength of the relationship between the two variables. A positive value of r indicates a positive correlation, where an increase in one variable is associated with an increase in the other variable, while a negative value of r indicates a negative correlation, where an increase in one variable is associated with a decrease in the other variable.

In [ ]:
```
import seaborn as sns
sns.heatmap(diabetes_data.corr(), annot=True, cmap='Purples')
```

Out[ ]:
```
<Axes: >
```

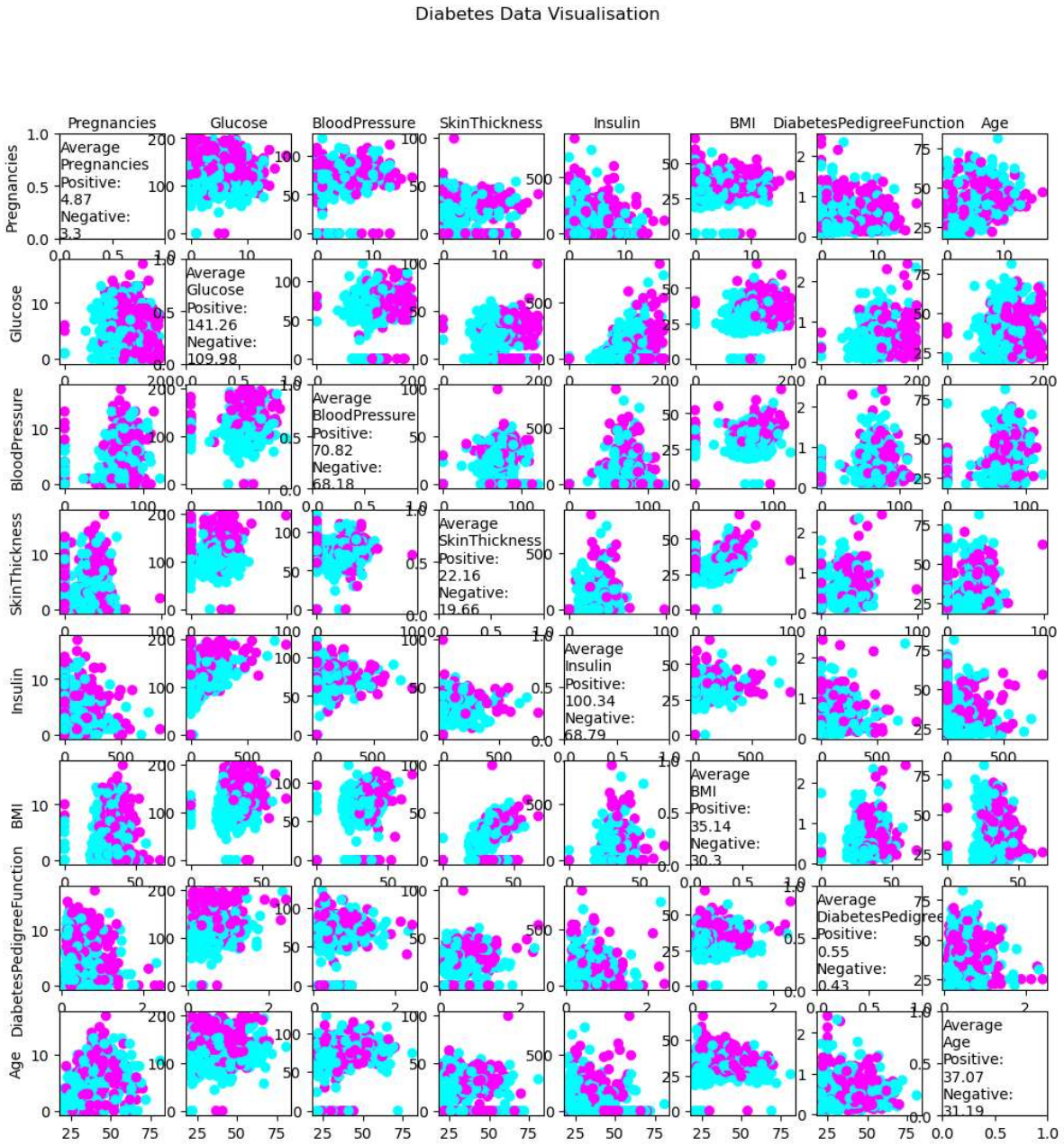


By studying the last row, we can see there is no negative correlation between any feature and the outcome, so any increase in a variable pushes towards a positive result, but of course to varriying degrees. We can also see there are features which notably correlate to a positive outcome such as Glucose (0.47) and BMI (0.29). We can also see there are features which don't notabely correlate to a positive outcome such as Blood Pressure (0.065), Skin Thickness (0.075) and Insulin (0.13).

We can visualise the data by plotting the points on a scatter plot. Each scatter plot shows a feature against other features.

In [ ]:
```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(8,8,figsize=(12,12))

fig.suptitle("Diabetes Data Visualisation")

labels = diabetes_data["Outcome"]
data = diabetes_data.drop("Outcome",axis = 1)

i = 0
j = 0
for feature1 in data:
    X = data[feature1]
    ax[0,i].xaxis.set_label_position('top')
    ax[0,i].set_xlabel(feature1)
    for feature2 in data:
        Y = data[feature2]
        ax[j,0].set_ylabel(feature2)
        if feature1 != feature2:
            ax[i,j].scatter(X, Y, c=labels, cmap='cool')
        else:
```

```
            ax[i,j].text(x=0,y=0,s="Average\n{}\nPositive:\n{}\nNegative:\n{}".for
        j += 1
    i += 1
    j = 0
```



There are three noticable points to discuss.

1. In every feature, the positive had a higher average, which proves the earlier decuction when studying the r values: Any increase in value for any feature correlates to a positive outcome.
2. The points are not very linearly serparable, we can see many irregular patterns on the scatter plots.
3. There are many points on the axis, showing many anomalies or missing data.

And so, the dataset must be checked for missing values.

In [ ]:
```
print("Number of null values per feature")
for column in diabetes_data:
    print("{}: {}".format(column,sum(diabetes_data[column] == 0)))
```

```
Number of null values per feature
Pregnancies: 111
Glucose: 5
BloodPressure: 35
SkinThickness: 227
Insulin: 374
BMI: 11
DiabetesPedigreeFunction: 0
Age: 0
Outcome: 500
```

Null values in pregnancies are normal.
Glucose, BloodPressure, SkinThickness, Insulin, BMI all have missing data, which needs to be addressed.
No null values in Age or DiabetesPedigreeFunction.

5. Preparing the data

After the analytics stage, we know there are two main problems with our data:

1. Missing Data.
2. Unbalanced dataset (65% negative records)

We will be exploring the effect of the data cleaning after we find the ideal parameters and methodoogies (see section 7.), and so we will be saving different versions of the dataset including this version (Version 1) which will have the data unchanged. The data is saved into X and y, with X storing the features and y storing the outcome.

In [ ]:
```
# Version 1

y1 = diabetes_data["Outcome"]
X1 = diabetes_data.drop("Outcome",axis = 1)

X1.head(5) # Preview
```

Out[ ]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	A
0	6	148	72	35	0	33.6	0.627	
1	1	85	66	29	0	26.6	0.351	
2	8	183	64	0	0	23.3	0.672	
3	1	89	66	23	94	28.1	0.167	
4	0	137	40	35	168	43.1	2.288	

Now to begin with the data cleaning. As covered in our analytics section, some features are more important than others. BMI and Glucose both had significant correlation with the outcome (r = 0.29 and r = 0.47 respectively) but also had very little missing data (11 and 5 records respectively). The best option in this case would be to drop the records with the missing data, since the features are important to our dataset.

In [ ]:
```
diabetes_data.drop(diabetes_data[diabetes_data["BMI"]==0].index, inplace = True)
diabetes_data.drop(diabetes_data[diabetes_data["Glucose"]==0].index, inplace = Tru
```

As for blood pressure, it has slightly more missing records (35) and very little correlation (r = 0.065), and so in that case it is probably better to remove the column.

In [ ]:
```
diabetes_data.drop("BloodPressure", axis=1, inplace=True)
diabetes_data.head(5) # Preview
```

Out[ ]:

	Pregnancies	Glucose	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	35	0	33.6	0.627	50	1
1	1	85	29	0	26.6	0.351	31	0
2	8	183	0	0	23.3	0.672	32	1
3	1	89	23	94	28.1	0.167	21	0
4	0	137	35	168	43.1	2.288	33	1

SkinThickness and Insulin had a very large number of missing data, and so the correlation graph does not give the full image. Therefore, we cannot definetly say we will not need those columns, and so another copy (Version 2) will be saved here.

In [ ]:
```
import numpy as np

#Version 2
copy = diabetes_data.drop("Insulin", axis=1)
copy = copy.drop("SkinThickness", axis=1)
y2 = copy["Outcome"]
X2 = copy.drop("Outcome",axis = 1)
```

Now as for our main version of the data (Version 3), we need to solve the issue of the missing data in SkinThickness and Insulin columns. For that we will be using a technique called imputation. Imputation is the process of filling in missing values in a dataset, based on existing data. Imputation is important for data preprocessing and can affect the accuracy of machine learning models. In this case, we opted to use an imputation method called K-nearest neighbors (KNN) imputation.

KNN imputation works by calculating the distance between the missing values and the other values in the dataset, then selecting the K nearest neighbors based on this distance metric, where K is a constant. The missing value is then imputed with the mean or median of the values in the selected neighbors. When working with missing data, KNN imputation can be helpful because it helps maintain the data's underlying distribution and prevents bias from being introduced by other imputation techniques.

In [ ]:
```
from sklearn.impute import KNNImputer
```

A key part of this algorithm is choosing the value of K. If we choose a number too big, it could lead to overfitting and if it is too small it can lead to underfitting. We could go for K = 4, since we have about 700 records and we can say each 0.5% of the records being similar is realistic (0.5% of 768 is 3.84).

In [ ]:
```
imputer = KNNImputer(n_neighbors=4)
```

To perform the imputation, we will replace the null values in Insulin and SkinThickness to NaN using numpy. The imputator will then change the NaN values to the imputed values.

In [ ]:
```
cols = ["SkinThickness","Insulin"]
diabetes_data[cols] = diabetes_data[cols].replace(0,np.NaN)


diabetes_data[cols] = imputer.fit_transform(diabetes_data[cols])
```

This is how the data looks now, after imputing the missing values:

In [ ]:
```
diabetes_data.head() # Preview
```

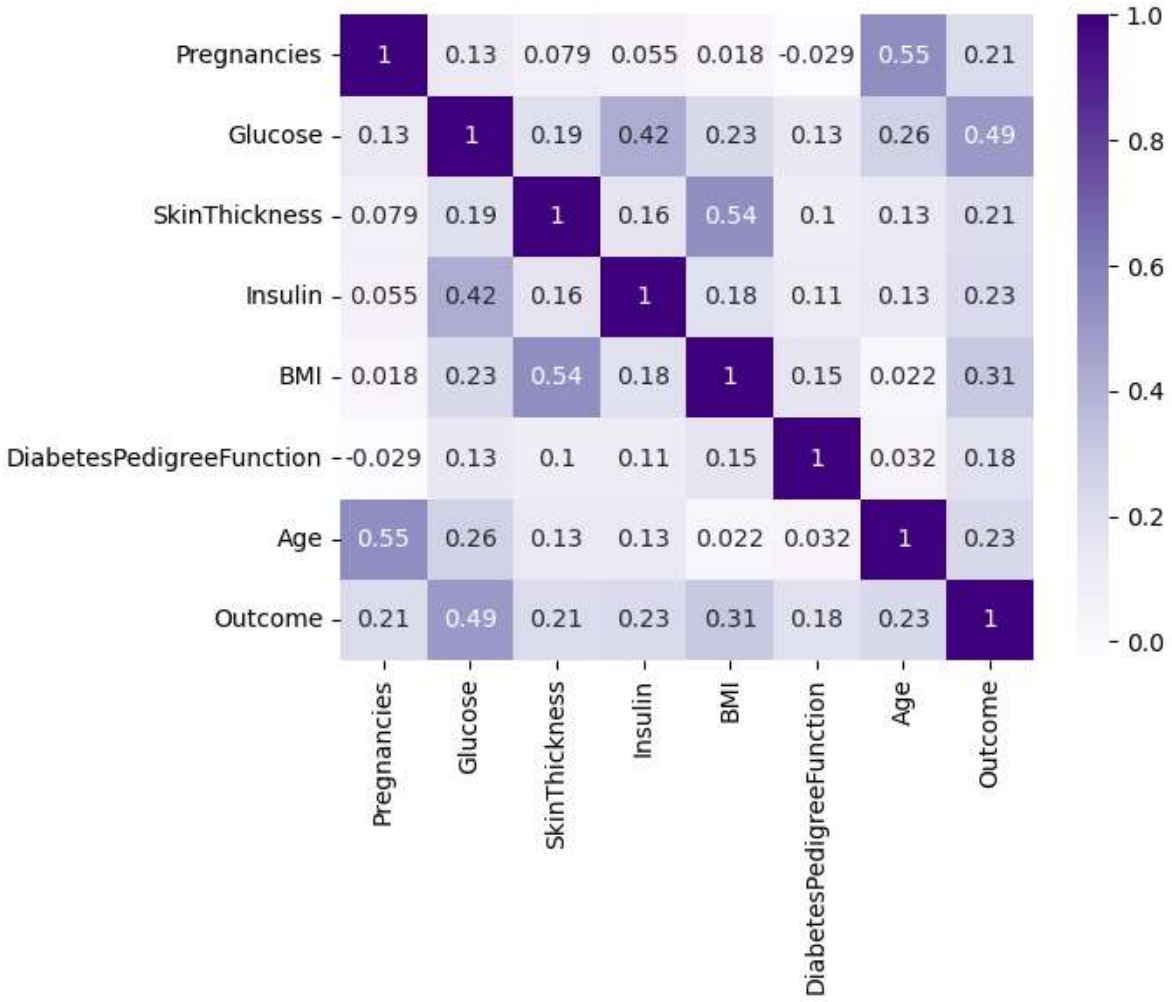
Out[ ]:

	Pregnancies	Glucose	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcom
0	6	148	35.000000	248.750000	33.6	0.627	50	
1	1	85	29.000000	126.750000	26.6	0.351	31	
2	8	183	29.172285	156.056122	23.3	0.672	32	
3	1	89	23.000000	94.000000	28.1	0.167	21	
4	0	137	35.000000	168.000000	43.1	2.288	33	

If we look at the heatmap showing the correlation now, we can see how SkinThickness and Inslin now have higher r values.</h5

In [ ]:
```
import seaborn as sns
sns.heatmap(diabetes_data.corr(), annot=True, cmap='Purples')
```

Out[ ]:
```
<Axes: >
```



We can save our main version now (Version 3).

In [ ]:
```
y3 = diabetes_data["Outcome"]
X3= diabetes_data.drop("Outcome",axis = 1)
```

The missing values in the data have now been fixed, we can look at null values in each version:

In [ ]:
```
print("Number of null values in Version 1 per feature")
for column in X1:
    print("{}: {}".format(column,sum(X1[column] == 0)))
print("")
print("Number of null values in Version 2 per feature")
for column in X2:
    print("{}: {}".format(column,sum(X2[column] == 0)))
print("")
print("Number of null values in Version 3 per feature")
for column in X3:
    print("{}: {}".format(column,sum(X3[column] == 0)))
```

```
Number of null values in Version 1 per feature
Pregnancies: 111
Glucose: 5
BloodPressure: 35
SkinThickness: 227
Insulin: 374
BMI: 11
DiabetesPedigreeFunction: 0
Age: 0

Number of null values in Version 2 per feature
Pregnancies: 108
Glucose: 0
BMI: 0
DiabetesPedigreeFunction: 0
Age: 0

Number of null values in Version 3 per feature
Pregnancies: 108
Glucose: 0
SkinThickness: 0
Insulin: 0
BMI: 0
DiabetesPedigreeFunction: 0
Age: 0
```

Now addressing the unbalanced dataset. As mentioned, an unbalanced dataset can cause bias, with the model giving more predictions with the majority class. 65% - 35% is generally not very unbalanced, but fixing it could be key to better results. To fix this we do sampling, and there are two main ways to sampling: undersampling and oversampling. Undersampling involves removing more of the majority class and so creating a balanced datasets. Oversampling is generating synthetic records of the minority class and thus giving a balanced dataset. Similarly to the versions of the dataset, we will create instances of the versions sampled using an undersampler (RandomUnderSampler), an oversampler (SMOTE) and with no balance. These tets will be carried out in section 7.

Both undersampling and oversampling will give similar effects. Both of them can improve performance but could also result in overfitting (undersampling) or overgeneralisation (oversampling). This happens when too much data is removed that it does more false positives or the other way round where it does more false negatives.

In [ ]:
```
from imblearn.under_sampling import RandomUnderSampler
from imblearn.over_sampling import SMOTE

rus = RandomUnderSampler(sampling_strategy="not minority")
smote = SMOTE()

X1_2 = X1 # Original Version 1
y1_2 = y1 # Original Version 1
```

```
X1, y1 = rus.fit_resample(X1_2, y1_2) #Undersampled Version 1
X1_3, y1_3 = smote.fit_resample(X1, y1) #Oversampled Version 1

X2_2 = X2  # Original Version 2
y2_2 = y2 # Original Version 2
X2, y2 = rus.fit_resample(X2_2, y2_2) #Undersampled Version 2
X2_3, y2_3 = smote.fit_resample(X1, y1) #Undersampled Version 2

X3_2 = X3 # Original Version 3
y3_2 = y3 # Original Version 3
X3, y3 = rus.fit_resample(X3_2, y3_2) #Undersampled Version 3
X3_3, y3_3 = smote.fit_resample(X1, y1) #Oversampled Version 3
```

Here's how much data is in each version of the dataset now, with how many in each category.

In [ ]:

```
print("Number of total records in Version 1 (Undersampled):", str(len(y1.index)))
print("Number of positives:",sum(y1 == 1), "({}%)".format(round(sum(y1 == 1)/len(y1
print("Number of negatives:",sum(y1 == 0), "({}%)".format(round(sum(y1 == 0)/len(y1
print("")
print("Number of total records in Version 2 (Undersampled):", str(len(y2.index)))
print("Number of positives:",sum(y2 == 1), "({}%)".format(round(sum(y2 == 1)/len(y2
print("Number of negatives:",sum(y2 == 0), "({}%)".format(round(sum(y2 == 0)/len(y2
print("")
print("Number of total records in Version 3 (Undersampled):", str(len(y3.index)))
print("Number of positives:",sum(y3 == 1), "({}%)".format(round(sum(y3 == 1)/len(y1
print("Number of negatives:",sum(y3 == 0), "({}%)".format(round(sum(y3 == 0)/len(y1
print("")
print("Number of total records in Version 1 (Not sampled):", str(len(y1_2.index)))
print("Number of positives:",sum(y1_2 == 1), "({}%)".format(round(sum(y1_2 == 1)/le
print("Number of negatives:",sum(y1_2 == 0), "({}%)".format(round(sum(y1_2 == 0)/le
print("")
print("Number of total records in Version 2 (Not sampled):", str(len(y2_2.index)))
print("Number of positives:",sum(y2_2 == 1), "({}%)".format(round(sum(y2_2 == 1)/le
print("Number of negatives:",sum(y2_2 == 0), "({}%)".format(round(sum(y2_2 == 0)/le
print("")
print("Number of total records in Version 3 (Not sampled):", str(len(y3_2.index)))
print("Number of positives:",sum(y3_2 == 1), "({}%)".format(round(sum(y3 == 1)/len
print("Number of negatives:",sum(y3_2 == 0), "({}%)".format(round(sum(y3 == 0)/len
print("")
print("Number of total records in Version 1 (Oversampled):", str(len(y1_3.index)))
print("Number of positives:",sum(y1_3 == 1), "({}%)".format(round(sum(y1_3 == 1)/le
print("Number of negatives:",sum(y1_3 == 0), "({}%)".format(round(sum(y1_3 == 0)/le
print("")
print("Number of total records in Version 2 (Oversampled):", str(len(y2_3.index)))
print("Number of positives:",sum(y2_3 == 1), "({}%)".format(round(sum(y2_3 == 1)/le
print("Number of negatives:",sum(y2_3 == 0), "({}%)".format(round(sum(y2_3 == 0)/le
print("")
print("Number of total records in Version 3 (Oversampled):", str(len(y3_3.index)))
print("Number of positives:",sum(y3_3 == 1), "({}%)".format(round(sum(y3 == 1)/len
print("Number of negatives:",sum(y3_3 == 0), "({}%)".format(round(sum(y3 == 0)/len
```

```
Number of total records in Version 1 (Undersampled): 536
Number of positives: 268 (50%)
Number of negatives: 268 (50%)

Number of total records in Version 2 (Undersampled): 528
Number of positives: 264 (50%)
Number of negatives: 264 (50%)

Number of total records in Version 3 (Undersampled): 528
Number of positives: 264 (50%)
Number of negatives: 264 (50%)

Number of total records in Version 1 (Not sampled): 768
Number of positives: 268 (35%)
Number of negatives: 500 (65%)

Number of total records in Version 2 (Not sampled): 752
Number of positives: 264 (35%)
Number of negatives: 488 (65%)

Number of total records in Version 3 (Not sampled): 752
Number of positives: 264 (35%)
Number of negatives: 488 (35%)

Number of total records in Version 1 (Oversampled): 536
Number of positives: 268 (50%)
Number of negatives: 268 (50%)

Number of total records in Version 2 (Oversampled): 536
Number of positives: 268 (50%)
Number of negatives: 268 (50%)

Number of total records in Version 3 (Oversampled): 536
Number of positives: 268 (49%)
Number of negatives: 268 (49%)
```

With a good amount of records remaining, which do not have any missing values or imbalance, the next step is to split the data. We will be taking 10% to do tests. The remaining 90% will undergo further splits when using cross validation and its folds in the testing section (see 6.).

In [ ]:
```
from sklearn.model_selection import train_test_split

X1_train, X1_test, y1_train, y1_test = train_test_split(X1,y1,test_size=0.1,random_
X2_train, X2_test, y2_train, y2_test = train_test_split(X2,y2,test_size=0.1,random_
X3_train, X3_test, y3_train, y3_test = train_test_split(X3,y3,test_size=0.1,random_

X1_train_2, X1_test_2, y1_train_2, y1_test_2 = train_test_split(X1_2,y1_2,test_size
X2_train_2, X2_test_2, y2_train_2, y2_test_2 = train_test_split(X2_2,y2_2,test_size
X3_train_2, X3_test_2, y3_train_2, y3_test_2 = train_test_split(X3_2,y3_2,test_size

X1_train_3, X1_test_3, y1_train_3, y1_test_3 = train_test_split(X1_3,y1_3,test_size
X2_train_3, X2_test_3, y2_train_3, y2_test_3 = train_test_split(X2_3,y2_3,test_size
X3_train_3, X3_test_3, y3_train_3, y3_test_3 = train_test_split(X3_3,y3_3,test_size
```

The data is now ready. The test involving the different versions with the different sampling methods will be carried out when we find the best model, and so we need one version to begin with right now.

We will use the undersampled version 3 dataset. My personal hypothesis is that undersampling is better for this project. Since positives are the minority class, errors when undersampling will probably be false positives, while in oversampling they would probably be false negatives. In this diabetes scenario, a false negative is way more dangerous than a false positive and so undersampling would be preferred. In addition,

the amount of records after undersampling is not that low and so it doesnt necessetate generating more records. We will test it out nonetheless in section 7.

6. Testing Section

The tests will involve different parts. 6.1 will look into SVC and its best parametres using gridsearchCV, while randomsearchCV will be used for the two Ensemble methods in 6.2.1 and 6.2.2. Both types search methods are used to find the best parametres. They also involve Cross Validation (CV), where data will be split into folds and repeat the training and testing process using each fold of the data. This repeats the tests and ensures we get the best possible confuguration for the model.

A standard scaler will be included in the tests, as the data needs to be scaled in order to have good performance and stability. We will be using pipelines to keep code organised as well.

It is also important to note the metrics which we will be using in the evaluation of the models.

- Accuracy: the proportion of correct predictions made by the model.
- Precision: the proportion of true positives (correctly identified instances of a class) to the total number of instances predicted as positive by the model.
- Recall: the proportion of true positives to the total number of instances that actually belong to the positive class.
- F1-score: a weighted average of precision and recall, with equal weight given to both measures.
- Area Under the Curve (AUC): the area under the receiver operating characteristic (ROC) curve, which plots the true positive rate (recall) against the false positive rate.

We'll set the scoring to both accuracy and recall.

In [ ]:
```
from sklearn.metrics import accuracy_score,recall_score,make_scorer
scoring = {"accuracy":make_scorer(accuracy_score),"recall":make_scorer(recall_scor
```

6.1. Support Vector Machines (SVM)

SVMs are a type of machine learning method used for regression and classification. They function by identifying the ideal decision boundary for classifying data into various groups. Using a kernel function, SVMs convert the data into a high-dimensional feature space and identify the best hyperplane for separating the data.

Hyperparameters:

1. C - regularization which balances the margin for error and penalties.
2. Gamma (γ) - kernal coefficient which determines the influence of each training example in the decision boundary.
3. Class Wights (cw) - weights assigned to each class to balance the importance of correct classification of each class.
4. (For Polynomial Kernal) d - degree of polynomial kernal function.
5. (For Radial Basis Function Kernal) r - radius of hypersphere, etermines the influence of each training example in the decision boundary in RBF kernal.

6.1.1. Test One: Support Vector Classifier (SVC)

The search process will involve using gridsearchCV. Grid Search tries every possible combination of the specified parametres and find the best combination while performing cross validation.

How we will test the hyperparameters:

We will try three different C values, three different gamma values and the three curves. For this we will use GridSearchCV as we do not have huge ranges of values to search in.

In [ ]:
```
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import RandomizedSearchCV, GridSearchCV
from sklearn.metrics import f1_score, make_scorer
from sklearn.svm import SVC

model = make_pipeline(StandardScaler(),SVC())

parametres = {'svc__C': [5,15,25],
              'svc__gamma': [0.01,0.05,0.1],
              'svc__kernel': ['linear', 'poly', 'rbf']}

grid = GridSearchCV(estimator=model, param_grid = parametres, cv = 5, scoring=scor
```

Grid search is ran and the best score is given along with the configuration which resulted in that score.

In [ ]:
```
grid_fit = grid.fit(X1_train,y1_train)
best_clf = grid.best_estimator_
p = grid.best_params_

print("Best score =", grid.best_score_)
print("Parameters:")
for i in p:
    print(i,":",p[i])
```

```
Best score = 0.7708333333333334
Parameters:
svc__C : 15
svc__gamma : 0.1
svc__kernel : rbf
```

As expected, the points were not linearly seperable and so rbf was the best curve.

We then run our test data in order to study the metrics.

In [ ]:
```
from sklearn.metrics import classification_report
y_pred = best_clf.predict(X1_test)

print(classification_report(y1_test,y_pred))
```

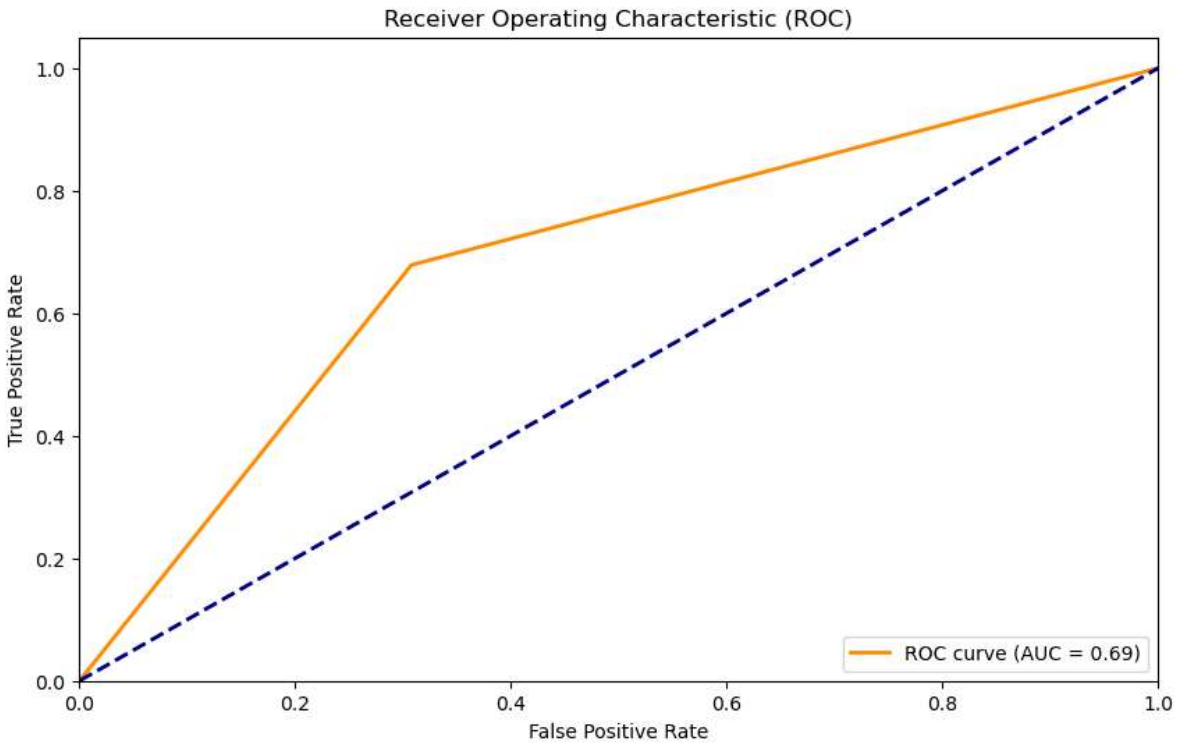
```
              precision    recall  f1-score   support

           0       0.67      0.69      0.68        26
           1       0.70      0.68      0.69        28

    accuracy                           0.69        54
   macro avg       0.69      0.69      0.69        54
weighted avg       0.69      0.69      0.69        54
```

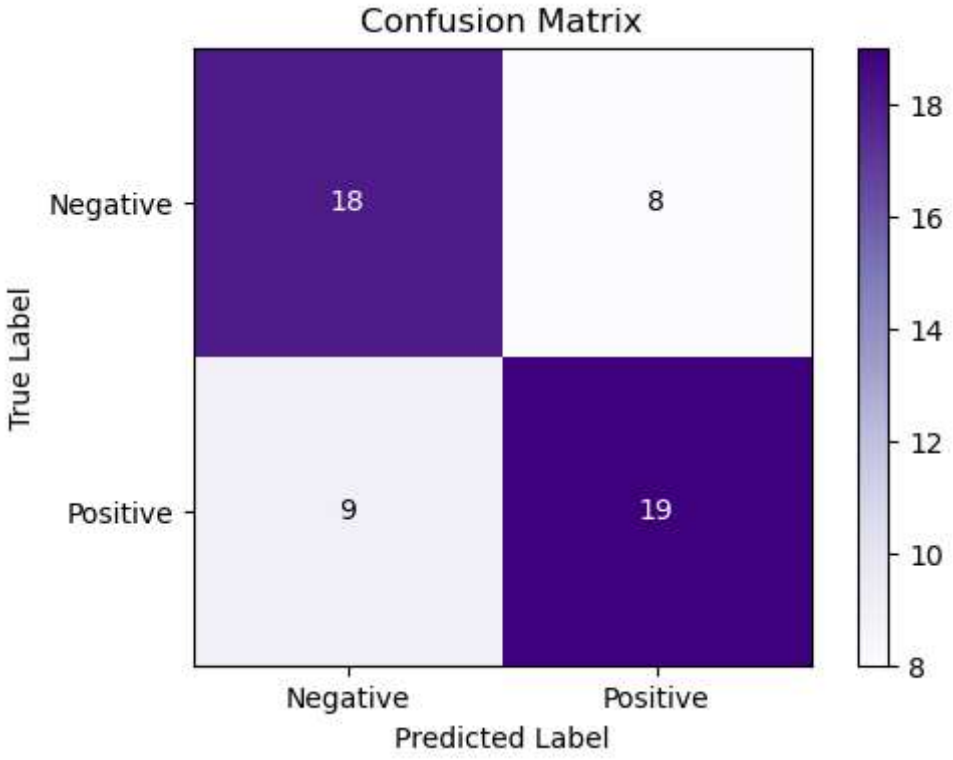
Good scores overall, the recall is close in both classes which is great and the accuracy is decent.

We can plot an ROC with the AUC and a confusion matrix too. (Function defined here for later use)

In [ ]:
```
from sklearn.metrics import roc_curve, auc, confusion_matrix

def plot(y_test,y_pred):
    fpr, tpr, thresholds = roc_curve(y_test, y_pred)
    roc_auc = auc(fpr, tpr)

    plt.figure(figsize=(10, 6))
    plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (AUC = %0.2f)' %
    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)')
    plt.legend(loc="lower right")
    plt.show()

    cm = confusion_matrix(y_test, y_pred)

    plt.figure(figsize=(6, 4))
    plt.imshow(cm, cmap=plt.cm.Purples)
    plt.title('Confusion Matrix')
    plt.colorbar()
    plt.xticks([0, 1], ['Negative', 'Positive'])
    plt.yticks([0, 1], ['Negative', 'Positive'])
    plt.xlabel('Predicted Label')
    plt.ylabel('True Label')

    for i in range(2):
        for j in range(2):
            plt.text(j, i, format(cm[i, j], 'd'), ha="center", va="center", color=

    plt.show()
```

In [ ]:
```
plot(y1_test,y_pred)
```





6.2. Ensemble Methods

Ensemble methods are a type of machine learning technique that combine multiple models to improve the predictive power and stability of the model. The goal of ensemble methods is to combine multiple base models' predictions, each trained on a separate portion of the dataset or with a different algorithm, into a single, more precise estimate. There are too main types of ensemble methods: bagging (bootstrap aggregating) and boosting. We will be trying one method from each type.

For the ensemble methods, we will use RandomizedSearchCV which is useful for techniques with many parametres, it seeks to find the best parametres within the ranges specfied.

6.2.1. Test Two: Ensemble Method #1 - RandomForestClassifier

Random Forest Classifier is a bagging ensemble technique. Random Forest Classifier involves creating a lot of decision trees, each of which is trained on a random portion of the input features and a subset of the training data. Each tree learns to anticipate the output class for a specific input based on the input's features during training. Following training of all the trees, the algorithm generates a forecast by combining the predictions of each individual decision tree. The class that gets the most votes across all the trees is the final prediction. This method increases the predictability and accuracy while lowering the danger of overfitting.

Parametres:

1. n_estimators: Number of decision trees in the random forest.
2. max_depth: The maximum depth of each decision tree in the random forest.
3. min_samples_split: The minimum number of samples required to split an internal node.
4. min_samples_leaf: The minimum number of samples required to be at a leaf node.
5. max_features: The number of features to consider when looking for the best split.
6. bootstrap: Whether bootstrap samples are used when building trees.
7. class_weight: Weights associated with classes. Useful for imbalanced data

We'll repeat the same test but with this setup now.

In [ ]:
```
from sklearn.ensemble import RandomForestClassifier

model = make_pipeline(StandardScaler(), RandomForestClassifier())

n_estimators = [int(x) for x in np.linspace(start = 10, stop = 80, num = 10)]
max_features = ['auto', 'sqrt']
max_depth = [2,4]
min_samples_split = [2, 5]
min_samples_leaf = [1, 2]
bootstrap = [True, False]

parameters = {'randomforestclassifier__n_estimators': n_estimators,
              'randomforestclassifier__max_features': max_features,
              'randomforestclassifier__max_depth': max_depth,
              'randomforestclassifier__min_samples_split': min_samples_split,
              'randomforestclassifier__min_samples_leaf': min_samples_leaf,
              'randomforestclassifier__bootstrap': bootstrap}


randomized = RandomizedSearchCV(estimator=model, param_distributions = parameters,
```

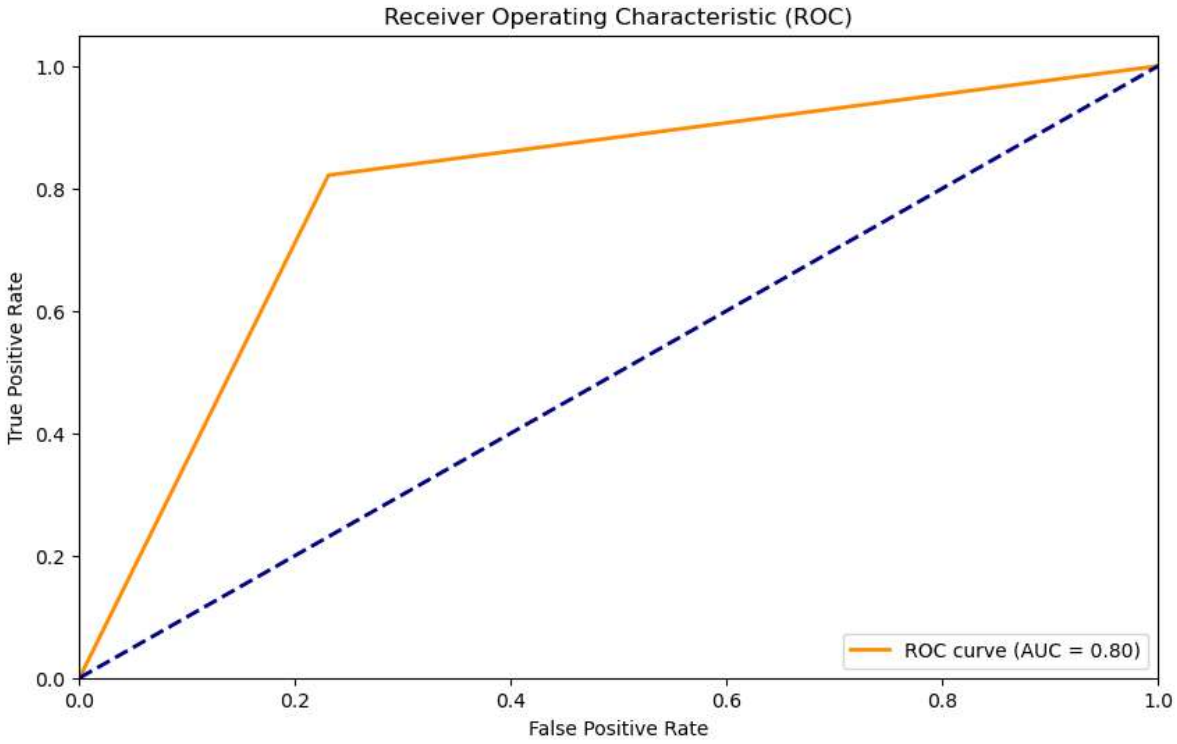
In [ ]:
```
randomized_fit = randomized.fit(X1_train,y1_train)
best_clf2 = randomized.best_estimator_
p = randomized.best_params_

print("Best score =", randomized.best_score_)
print("Parameters:")
for i in p:
    print(i,":",p[i])
```

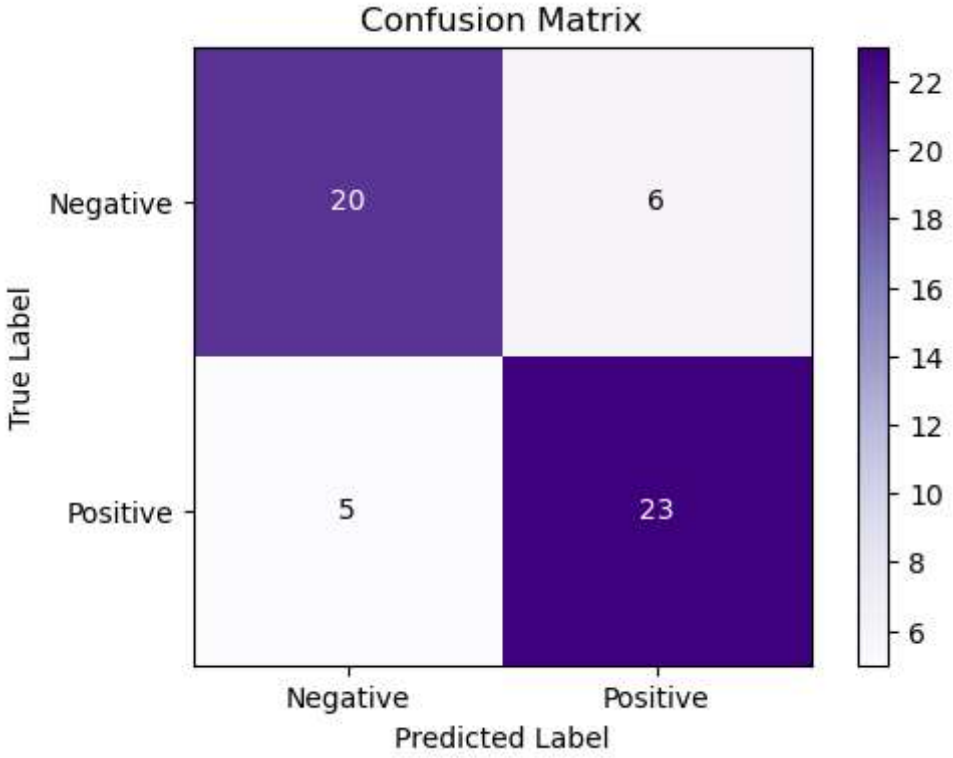
```
Best score = 0.7708333333333333
Parameters:
randomforestclassifier__n_estimators : 25
randomforestclassifier__min_samples_split : 5
randomforestclassifier__min_samples_leaf : 1
randomforestclassifier__max_features : sqrt
randomforestclassifier__max_depth : 4
randomforestclassifier__bootstrap : False
```

In [ ]:
```
y_pred = best_clf2.predict(X1_test)

print(classification_report(y1_test,y_pred))
```

```
              precision    recall  f1-score   support

           0       0.80      0.77      0.78        26
           1       0.79      0.82      0.81        28

    accuracy                           0.80        54
   macro avg       0.80      0.80      0.80        54
weighted avg       0.80      0.80      0.80        54
```

In [ ]:
```
plot(y1_test,y_pred)
```





6.2.2. Test Three: Ensemble Method #2 - AdaBoosting

AdaBoost or Adaptive Boosting is a boosting ensemble technique. It works by combining multiple weak models to form a strong model that can make more accurate predictions. The idea behind AdaBoost is to iteratively train the models, where each model is trained on the misclassified examples from the previous model. The predictions from each model are combined using a weighted vote to make a final prediction. In every iteration, the algorithm assigns higher weights to the misclassified examples, which forces the next model to focus more on these examples.

Parametres:

1. n_estimators: The number of base estimators to include in the ensemble.

2. learning_rate: The contribution of each model to the final prediction.
3. algorithm: The algorithm to use for boosting. Options are SAMME (discrete boosting) and SAMME.R (real boosting).

Once again we repeat this experiment

In [ ]:
```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier

model = AdaBoostClassifier(DecisionTreeClassifier(max_depth=2))
pipeline = make_pipeline(StandardScaler(), model)

parameters = {
    'n_estimators': [50, 100, 200, 500],
    'learning_rate': [0.001, 0.01, 0.1, 1],
    'algorithm': ['SAMME', 'SAMME.R']
}

randomized = RandomizedSearchCV(estimator=model, param_distributions = parameters,
```

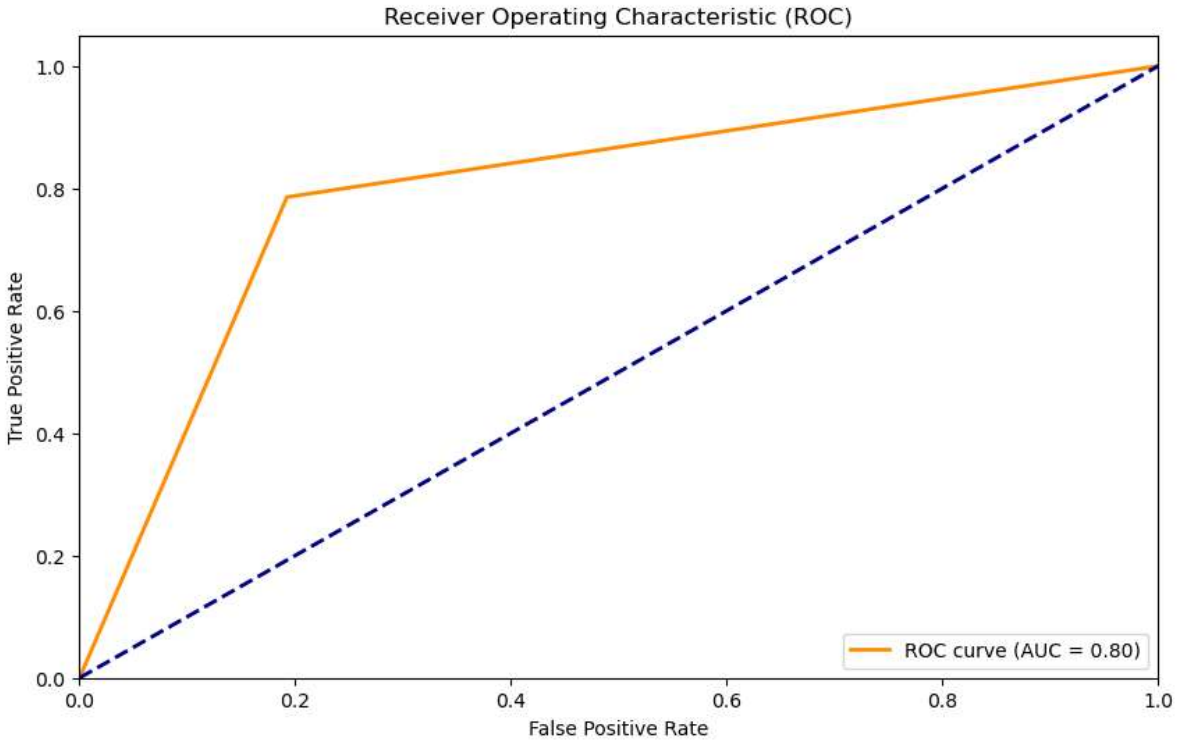
In [ ]:
```
randomized_fit = randomized.fit(X1_train,y1_train)
best_clf3 = randomized.best_estimator_
p = randomized.best_params_

print("Best score =", randomized.best_score_)
print("Parameters:")
for i in p:
    print(i,":",p[i])
```

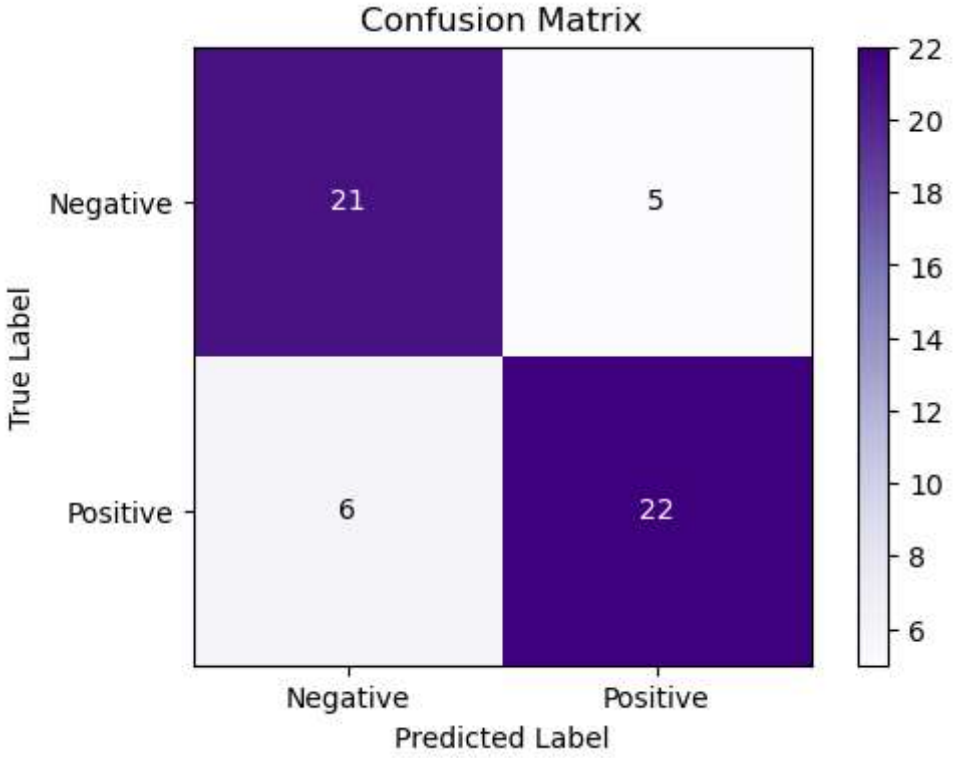
```
Best score = 0.7333333333333334
Parameters:
n_estimators : 100
learning_rate : 0.1
algorithm : SAMME
```

In [ ]:
```
y_pred = best_clf3.predict(X1_test)

print(classification_report(y1_test,y_pred))
```

```
              precision    recall  f1-score   support

           0       0.78      0.81      0.79        26
           1       0.81      0.79      0.80        28

    accuracy                           0.80        54
   macro avg       0.80      0.80      0.80        54
weighted avg       0.80      0.80      0.80        54
```

In [ ]:
```
plot(y1_test,y_pred)
```





6.3. Conclusions

These results tell us that the best set up was AdaBoost with these parameters: n_estimators : 100 learning_rate : 0.01 algorithm : SAMME.R

AdaBoost had the best scores overall, with the highest accuracy and best recall. RandomForest had and SVM were decent but below AdaBoost. AdaBoost makes sense for this algorithm, we do not have many features so this works well with the Decesion Trees set up with AdaBoost.

7. Further Tests

Now to test the variations in the dataset. It will be tested on AdaBoost with the configuration mentioned above.

In [ ]:
```
version1 = [{"trainX":X1_train,"trainy":y1_train,"testX":X1_train,"testy":y1_train
           {"trainX":X1_train_2,"trainy":y1_train_2,"testX":X1_train_2,"testy":y1_
           {"trainX":X1_train_3,"trainy":y1_train_3,"testX":X1_train_3,"testy":y1_

version2 = [{"trainX":X2_train,"trainy":y2_train,"testX":X2_train,"testy":y2_train
           {"trainX":X2_train_2,"trainy":y2_train_2,"testX":X2_train_2,"testy":y2_
           {"trainX":X2_train_3,"trainy":y2_train_3,"testX":X2_train_3,"testy":y2_

version3 = [{"trainX":X3_train,"trainy":y3_train,"testX":X3_train,"testy":y3_train
           {"trainX":X3_train_2,"trainy":y3_train_2,"testX":X3_train_2,"testy":y3_
           {"trainX":X3_train_3,"trainy":y3_train_3,"testX":X3_train_3,"testy":y3_

versions = [version1,version2,version3]

i = 0
j = 0
for version in versions:
    for combination in version:
        print("Version ",i+1)
        if j == 0:
            print("Undersampled")
        elif j == 1:
            print("Not sampled")
        else:
            print("Oversampled")

        model = AdaBoostClassifier(DecisionTreeClassifier(max_depth=2),n_estimators
        pipeline = make_pipeline(StandardScaler(), model)

        pipeline.fit(combination["trainX"],combination["trainy"])
        y_pred = pipeline.predict(combination["testX"])

        fpr, tpr, thresholds = roc_curve(combination["testy"], y_pred)
        roc_auc = auc(fpr, tpr)

        print(classification_report(combination["testy"],y_pred))

        cm = confusion_matrix(combination["testy"], y_pred)
        print(cm)

        j+=1

    i+=1
    j=0
```

```
Version  1
Undersampled
              precision    recall  f1-score   support

           0       0.78      0.80      0.79       242
           1       0.79      0.78      0.78       240

    accuracy                           0.79       482
   macro avg       0.79      0.79      0.79       482
weighted avg       0.79      0.79      0.79       482

[[194  48]
 [ 54 186]]
Version  1
Not sampled
              precision    recall  f1-score   support

           0       0.80      0.90      0.85       452
           1       0.75      0.58      0.65       239

    accuracy                           0.79       691
   macro avg       0.78      0.74      0.75       691
weighted avg       0.78      0.79      0.78       691

[[406  46]
 [101 138]]
Version  1
Oversampled
              precision    recall  f1-score   support

           0       0.78      0.80      0.79       242
           1       0.79      0.78      0.78       240

    accuracy                           0.79       482
   macro avg       0.79      0.79      0.79       482
weighted avg       0.79      0.79      0.79       482

[[194  48]
 [ 54 186]]
Version  2
Undersampled
              precision    recall  f1-score   support

           0       0.77      0.75      0.76       240
           1       0.75      0.77      0.76       235

    accuracy                           0.76       475
   macro avg       0.76      0.76      0.76       475
weighted avg       0.76      0.76      0.76       475

[[181  59]
 [ 55 180]]
Version  2
Not sampled
              precision    recall  f1-score   support

           0       0.80      0.87      0.84       432
           1       0.73      0.63      0.67       244

    accuracy                           0.78       676
   macro avg       0.77      0.75      0.75       676
weighted avg       0.78      0.78      0.78       676

[[375  57]
```

```
 [ 91 153]]
Version  2
Oversampled
              precision    recall  f1-score   support

           0       0.78      0.80      0.79       242
           1       0.79      0.78      0.78       240

    accuracy                           0.79       482
   macro avg       0.79      0.79      0.79       482
weighted avg       0.79      0.79      0.79       482

[[194  48]
 [ 54 186]]
Version  3
Undersampled
              precision    recall  f1-score   support

           0       0.77      0.79      0.78       240
           1       0.78      0.76      0.77       235

    accuracy                           0.78       475
   macro avg       0.78      0.78      0.78       475
weighted avg       0.78      0.78      0.78       475

[[190  50]
 [ 56 179]]
Version  3
Not sampled
              precision    recall  f1-score   support

           0       0.80      0.87      0.84       432
           1       0.73      0.63      0.67       244

    accuracy                           0.78       676
   macro avg       0.77      0.75      0.75       676
weighted avg       0.78      0.78      0.78       676

[[375  57]
 [ 91 153]]
Version  3
Oversampled
              precision    recall  f1-score   support

           0       0.78      0.80      0.79       242
           1       0.79      0.78      0.78       240

    accuracy                           0.79       482
   macro avg       0.79      0.79      0.79       482
weighted avg       0.79      0.79      0.79       482

[[194  48]
 [ 54 186]]
```

My hypothesis was correct...kind of. Overall close scores were seen by undersampling and oversampling but then again, factoring in the severity of the scenario undersampling is probably better. As for the imputation and cleaning of data while simply removing the colums got decent scores, imputation still takes the cake. It performed better.</h3>