Comparing and Contrasting the Basic Principles and Characteristics of a range of Contemporary Machine Learning Algorithms

Arthur Milner - 21035478

Diabetes Dataset

To prepare the dataset for training on a model I first imported the diabetes.csv file into a DataFrame using pandas, the DataFrame pandas provides also greatly improves the readability of the data in the notebook. Consequently making it easier to get a better grasp on the structure of the dataset and what details are stored where, DataFrames also include built in features such as .head(x)/.tail(x) and .index to aid in analysing the structure of the dataset.

In [1]:

```python
#importing libraries needed

import pandas as pd
import numpy as np
import seaborn as sns #for statistics ploting
import matplotlib.pyplot as plt
import math
import sklearn
from sklearn import datasets, linear_model
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import RobustScaler
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from scipy.stats import pearsonr
from sklearn import svm
from sklearn.metrics import confusion_matrix,classification_report
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import AdaBoostClassifier
%matplotlib inline
```

In [2]:

```python
diabetesData = pd.read_csv('./datasets/diabetes.csv') # Reads dataset into a DataFrame from the diabetes.csv file
diabetesData.head(5) #First 5 rows of the DataFrame, checking it has imported correctly
```

Out[2]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

In [3]:

```python
diabetesData.tail(5) #Last 5 rows of the DataFrame, checking it has imported correctly
```

Out[3]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
763	10	101	76	48	180	32.9	0.171	63	0
764	2	122	70	27	0	36.8	0.340	27	0
765	5	121	72	23	112	26.2	0.245	30	0
766	1	126	60	0	0	30.1	0.349	47	1
767	1	93	70	31	0	30.4	0.315	23	0

In [4]:

```python
diabetesData.index #Shows number of rows in the dataset
```

Out[4]:

```
RangeIndex(start=0, stop=768, step=1)
```

Upon viewing the columns and values within the DataFrame, we can see the problem involves two classes, these classes being 0 and 1 to represent either a positive (1) or a negative (0) diagnosis of diabetes. Also noticeable is there are many missing values within some of the columns, such as SkinThickness, in order to get a better picture of the severity of this I will replace the necessary 0 values with null and use pandas built in functions to detect null values. Also notable about the dataset is that each column appears valuable in predicting diabetes as they can all have an effect on the development of the condition.

Analysing the Dataset Before Processing:

Firstly I will replace the 0 values in the DataFrame with null, from there I will decide how to manage these null values once I gather the frequency of them in each respective column using diabetesData.isNull().sum().

In [5]:

```python
#Code to replace the 0 values with N/A
columnsToChange = ['Glucose', 'BloodPressure', 'SkinThickness', 'Insulin','BMI','Age']
for i in range(len(columnsToChange)-1):
    diabetesData[columnsToChange[i]].replace(to_replace = 0, value = pd.NA, inplace=True)
```

In [6]:

```python
diabetesData.isnull().sum()
```

Out[6]:

```
Pregnancies                   0
Glucose                       5
BloodPressure                35
SkinThickness               227
Insulin                     374
BMI                          11
DiabetesPedigreeFunction      0
Age                           0
Outcome                       0
dtype: int64
```

In [7]:

```python
diabetesData.head(20) #Verifying replacement
```

Out[7]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	<NA>	33.6	0.627	50	1
1	1	85	66	29	<NA>	26.6	0.351	31	0
2	8	183	64	<NA>	<NA>	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1
5	5	116	74	<NA>	<NA>	25.6	0.201	30	0
6	3	78	50	32	88	31.0	0.248	26	1
7	10	115	<NA>	<NA>	<NA>	35.3	0.134	29	0
8	2	197	70	45	543	30.5	0.158	53	1
9	8	125	96	<NA>	<NA>	<NA>	0.232	54	1
10	4	110	92	<NA>	<NA>	37.6	0.191	30	0
11	10	168	74	<NA>	<NA>	38.0	0.537	34	1
12	10	139	80	<NA>	<NA>	27.1	1.441	57	0
13	1	189	60	23	846	30.1	0.398	59	1
14	5	166	72	19	175	25.8	0.587	51	1
15	7	100	<NA>	<NA>	<NA>	30.0	0.484	32	1
16	0	118	84	47	230	45.8	0.551	31	1
17	7	107	74	<NA>	<NA>	29.6	0.254	31	1
18	1	103	30	38	83	43.3	0.183	33	0
19	1	115	70	30	96	34.6	0.529	32	1

Following the result of diabetesData.isnull().sum() we can see there are many missing values, .head(20) shows even the first lot of values contain multiple null results.

Pre-Processing the Data:

Now that I have identified the fact all columns are necessary for the dataset and the rows with missing values I have some pre-processing to do to ensure those missing values do not harm the effeciveness of the model. I have decided for the rows containing null values, if the frequency of these null values is above one I will delete the row, if the row contains just one null value I will replace it with the appropriate for that row.

In [8]:
```
for index, row in diabetesData.iterrows():
    countOfNull = diabetesData.loc[[index]].isna().sum().sum() #Gets number of null values for current row
    if countOfNull > 1:
        diabetesData = diabetesData.drop([index])
```

In [9]:
```
diabetesData.head(20) #Verify rows with multiple null values have been removed
```

Out[9]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	<NA>	33.6	0.627	50	1
1	1	85	66	29	<NA>	26.6	0.351	31	0
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1
6	3	78	50	32	88	31.0	0.248	26	1
8	2	197	70	45	543	30.5	0.158	53	1
13	1	189	60	23	846	30.1	0.398	59	1
14	5	166	72	19	175	25.8	0.587	51	1
16	0	118	84	47	230	45.8	0.551	31	1
18	1	103	30	38	83	43.3	0.183	33	0
19	1	115	70	30	96	34.6	0.529	32	1
20	3	126	88	41	235	39.3	0.704	27	0
23	9	119	80	35	<NA>	29.0	0.263	29	1
24	11	143	94	33	146	36.6	0.254	51	1
25	10	125	70	26	115	31.1	0.205	41	1
27	1	97	66	15	140	23.2	0.487	22	0
28	13	145	82	19	110	22.2	0.245	57	0
30	5	109	75	26	<NA>	36.0	0.546	60	0
31	3	158	76	36	245	31.6	0.851	28	1
32	3	88	58	11	54	24.8	0.267	22	0

In [10]:
```
diabetesData.isnull().sum() #Checking what values are still null
```

Out[10]:
```
Pregnancies                   0
Glucose                       1
BloodPressure                 0
SkinThickness                 0
Insulin                     140
BMI                           1
DiabetesPedigreeFunction      0
Age                           0
Outcome                       0
dtype: int64
```

Upon reviewing which rows still contain null values after removing those with multiple nulls it appears only insulin remains, with two outliers being BMI and Glucose. Considering this it might make sense to drop the two rows with BMI and Glucose missing and then predict the insulin for the remaining 140 rows using a linear regression line.

In [11]:
```
diabetesData = diabetesData.dropna(subset=['Glucose'])
diabetesData = diabetesData.dropna(subset=['BMI'])
diabetesData.isnull().sum() #Checking what values are still null
```

Out[11]:
```
Pregnancies                   0
Glucose                       0
BloodPressure                 0
SkinThickness                 0
Insulin                     140
BMI                           0
DiabetesPedigreeFunction      0
Age                           0
```

```
Outcome                       0
dtype: int64
```

Using Graphs to Identify Relationships Between Variables

Where I have identified insulin to be the column missing the most values after removing rows with mulitple null values I will use matplotlib to visualise how the other variables might effect the insulin value, drawing upon real world knowledge it is expected columns such as BMI and Glucose will be the greatest indicators. In identifying the best variable to use I can draw a linear regression line which returns a prediction of these null values for insertion into the dataset. The goal of this is to give more accurate values compared to the approach of simply inserting the average insulin value into all null values.

In [12]:
```
#Creating a copy of the current DataFrame without any null insulin values
diabetesData1 = diabetesData.dropna()
```

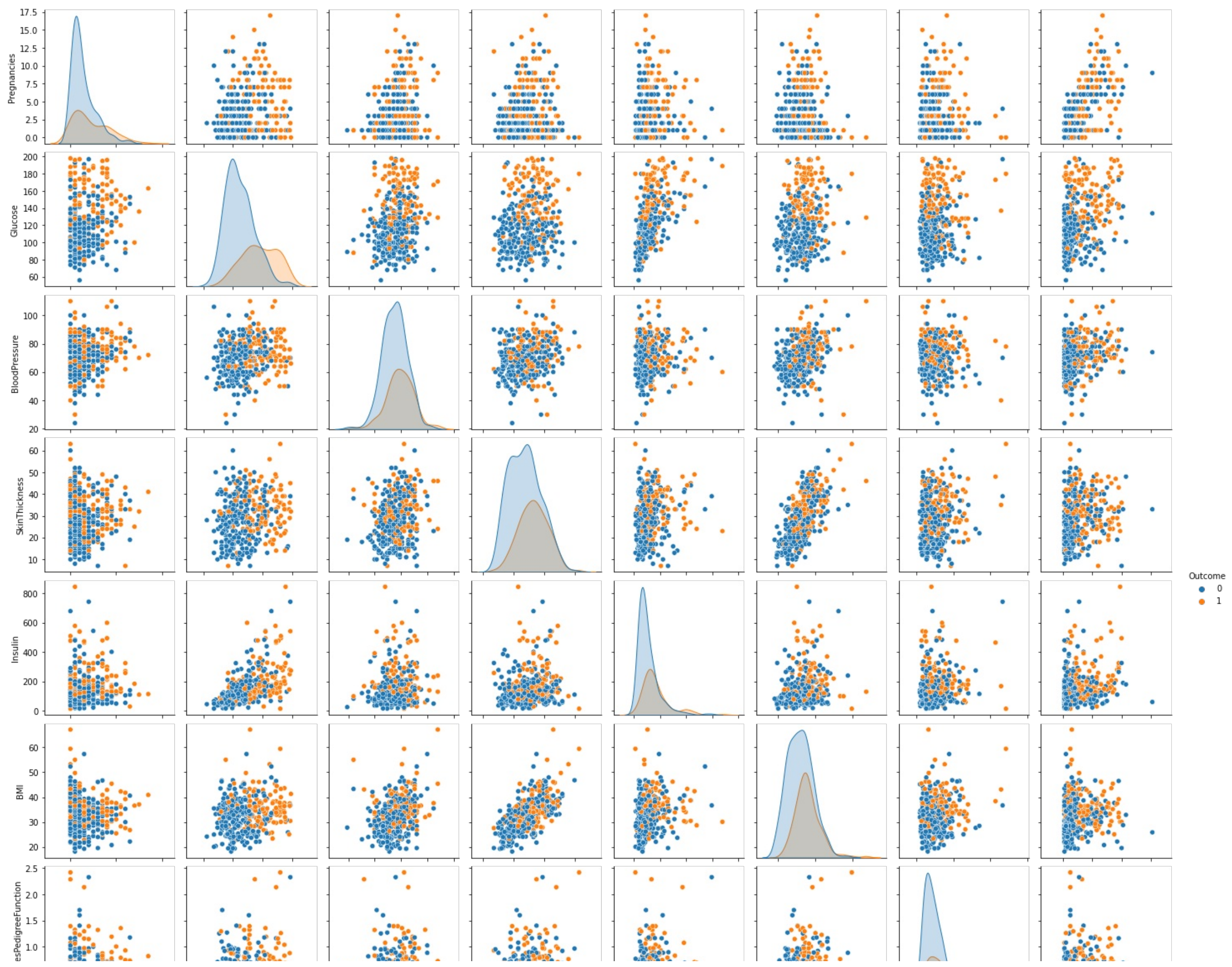
In [13]:
```
diabetesData1.isnull().sum() #Checking what values are still null
```

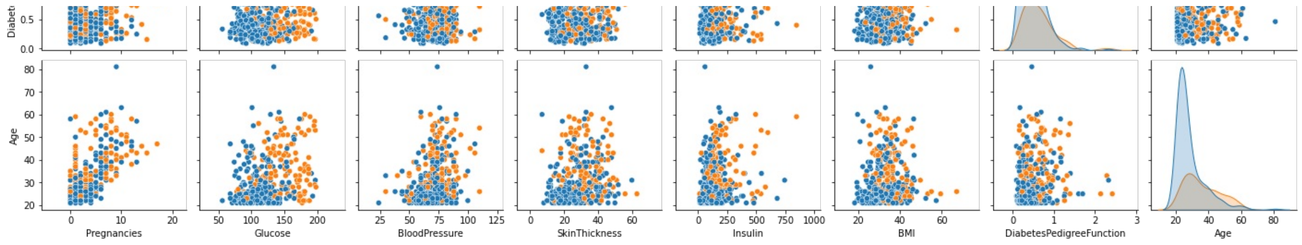
Out[13]:
```
Pregnancies                 0
Glucose                     0
BloodPressure               0
SkinThickness               0
Insulin                     0
BMI                         0
DiabetesPedigreeFunction    0
Age                         0
Outcome                     0
dtype: int64
```

Using seaborn pairplot it allows us to see the relationship between all variables in the dataset, including insulin.

In [14]:
```
sns.pairplot(diabetesData1, hue="Outcome")
```

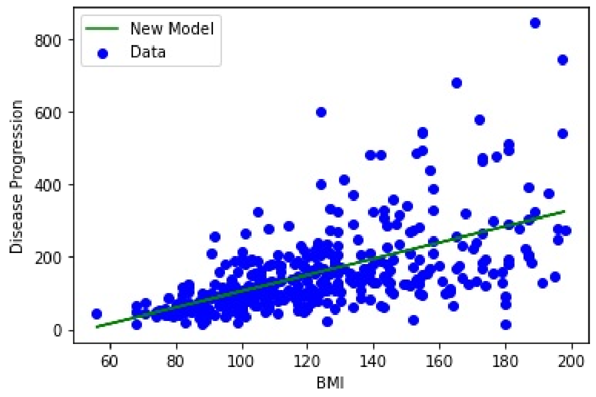
Out[14]:
```
<seaborn.axisgrid.PairGrid at 0x7fb2d04bd040>
```





Upon viewing the graphs above, it appears glucose is the best variable for predicting insulin levels, with this knowledge I will now plot a linear regression line in order to predict the values which will fill in the null values present in the dataset. This also makes sense as in real life those with high glucose may get insulin injections. Also notable about the graphs is that some measures appear to be much better indicators of diabetes than others, for instance glucose has a clear link between the outcome.

In [15]:
```python
glucoseX = diabetesData1['Glucose'].to_numpy()
insulinY = diabetesData1['Insulin'].to_numpy()

regr = LinearRegression().fit(glucoseX.reshape(-1, 1), insulinY)

insulinYHat = regr.predict(glucoseX.reshape(-1, 1))

plt.scatter(glucoseX, insulinY, c='b', label='Data')
plt.plot(glucoseX, insulinYHat, c='g', label='New Model')

plt.legend(loc='best')

plt.xlabel('BMI')
plt.ylabel('Disease Progression')
plt.show()
```



With the linear regression line plotted I must now use it to insert the null values.

In [16]:
```python
for index, row in diabetesData.iterrows():
    countOfNull = diabetesData.loc[[index]].isna().sum().sum() #Gets number of null values for current row
    if countOfNull == 1:
        glucoseNewX = np.array([[diabetesData.loc[index, 'Glucose']]])
        insulinNewYHat = int(regr.predict(glucoseNewX.reshape(-1, 1)))
        diabetesData.loc[index, 'Insulin'] = insulinNewYHat
```

In [17]:
```python
diabetesData.head(5) #Verifying values have changed
```

Out[17]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	212	33.6	0.627	50	1
1	1	85	66	29	71	26.6	0.351	31	0
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1
6	3	78	50	32	88	31.0	0.248	26	1

In [18]:
```python
diabetesData.isnull().sum() #Checking what values are still null
```

Out[18]:
```
Pregnancies                 0
Glucose                     0
```

```
BloodPressure               0
SkinThickness               0
Insulin                     0
BMI                         0
DiabetesPedigreeFunction    0
Age                         0
Outcome                     0
dtype: int64
```

Feature Selection

Using the pearson correlation below I am checking to see how relevant each feature is in detecting diabetes.

In [19]:

```
columns = ["Pregnancies", "Glucose", "BloodPressure", "SkinThickness", "Insulin", "BMI", "DiabetesPedigreeFunctic
for i in columns:
    correlation, na = pearsonr(diabetesData[i], diabetesData["Outcome"])
    print(i, correlation)
```

```
Pregnancies 0.2525855109647306
Glucose 0.5036139374861698
BloodPressure 0.1834318739109298
SkinThickness 0.254873710725995
Insulin 0.324336516507137
BMI 0.300900747668014
DiabetesPedigreeFunction 0.2330738977685281
Age 0.3150968324412516
```

The above results denotes Glucose, BMI, Age and Insulin to be the biggest detectors, as blood pressure has a low correlation (under 0.2 is considered weak) it may be worth removing it from the dataset entirely.

In [20]:

```
diabetesData = diabetesData.drop(columns=['BloodPressure'])
diabetesData.head(5)
```

Out[20]:

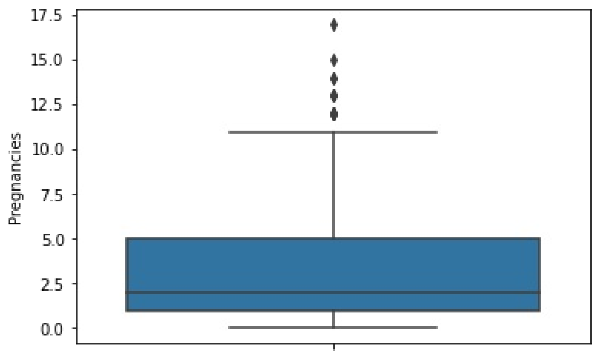
	Pregnancies	Glucose	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	35	212	33.6	0.627	50	1
1	1	85	29	71	26.6	0.351	31	0
3	1	89	23	94	28.1	0.167	21	0
4	0	137	35	168	43.1	2.288	33	1
6	3	78	32	88	31.0	0.248	26	1

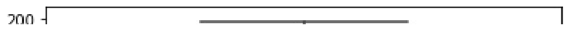
Scaling the Dataset

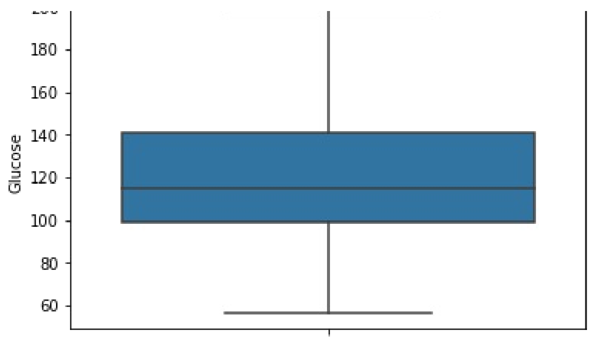
Because some algorithms I will be using are effected by whether or not data is scaled it is important I scale the dataset. Scaling a dataset is done because data is often stored in different measures, which when using something such as euclidian distance can severely alter results.

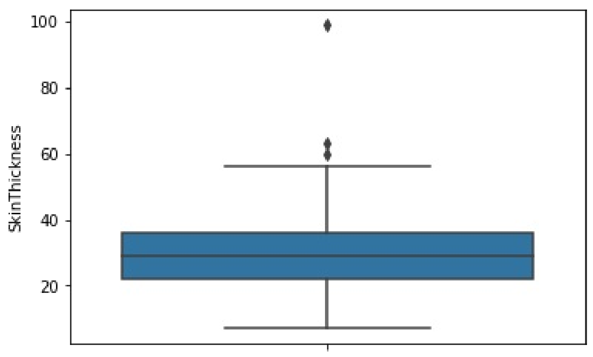
In [21]:

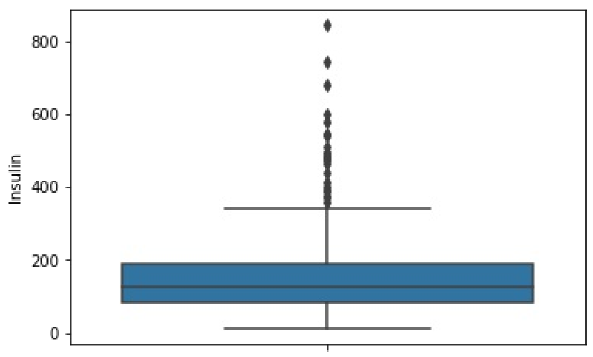
```
#Checking the outliers within the dataset, if there are a large amount min max scaler
# might not be viable as it is sensitive to outliers
for column in diabetesData.columns:
    plt.figure()
    sns.boxplot(y = column, data = diabetesData, orient = "v")
```

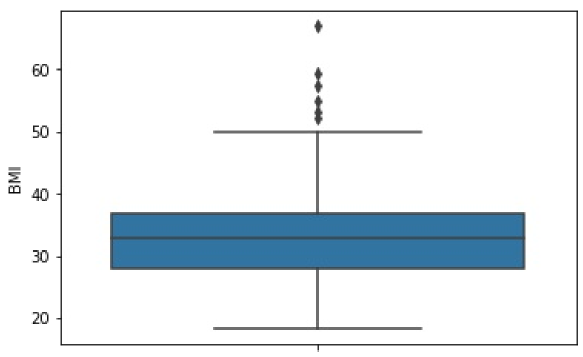


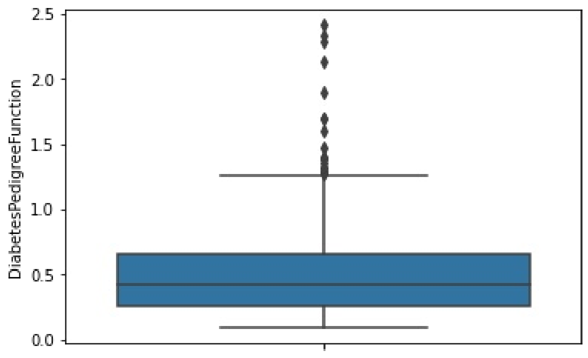


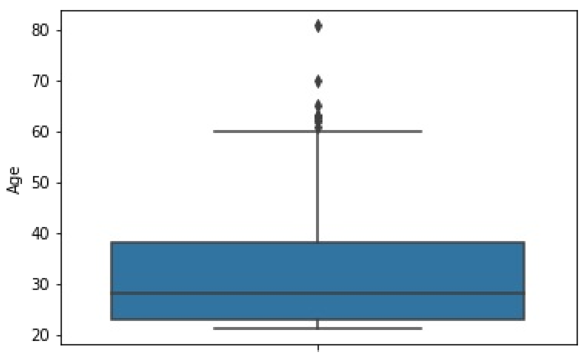

Glucose
180
160
140
120
100
80
60

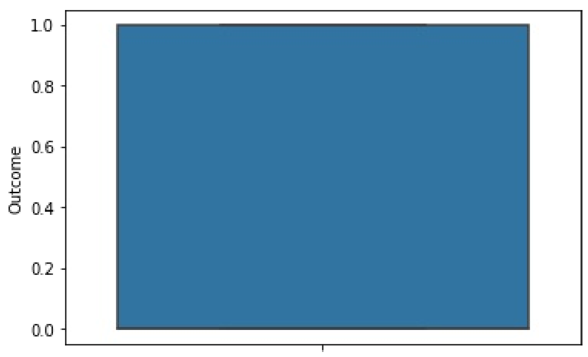

SkinThickness
100
80
60
40
20


Insulin
800
600
400
200
0


BMI
60
50
40
30
20


DiabetesPedigreeFunction
2.5
2.0
1.5
1.0
0.5
0.0


Age
80
70
60
50
40
30
20



Looking at the box plots it is clear the data includes quite a few outliers, consequently a robust scaler might be the best option as it is much less sensitive to outliers due to it making a much wider range from the values, thus reducing the relative distance.

In [22]:
```
#Get dataset into x and y variables, y for the outcome
X = diabetesData.iloc[:,:-1].values
y = diabetesData.iloc[:,-1].values
```

In [23]:
```
#Splitting the dataset into train and test, 80% train, 30% test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)
```

In [24]:
```
#Showing the amount in each array
print('X_train: ', X_train.shape)
print('X_test: ', X_test.shape)
print('y_train: ', y_train.shape)
print('y_test: ', y_test.shape)
```

```
X_train:  (372, 7)
X_test:  (160, 7)
y_train:  (372,)
y_test:  (160,)
```

Note: For the grid searches below I often did experimentation in order to decide on a nice range of values.

SVM Classifier

The first model I will train will be an SVM classifier, the hyper-parameters I will need to tune will be:

- Kernel
- C
- Degree (Polynomial)
- Gamma (RBF)

I have chosen SVM because it is widely used for classification problems and generally performs well on a wide variety of datasets. The size of the dataset is also further reasoning for choosing SVM classifier as with large datasets SVM can be very time consuming, which should not be a problem here. The many kernels you can use with SVM also appeal to me as it makes it quite versatile for many dimensions and various datasets.

To do this I will use grid search CV and justify why certain parameters perform better than others as I go along. I decided on grid search as it is very efficient in testing multiple parameters/values with minimal code, meaning I can try low, medium and high values all at once, I am also using cross validation with 5 folds to further the accuracy of my results through averaging performance.

- Perhaps the most important hyper-parameter for SVM is the kernel, consequently it is very important that I spend a majority of time deciding on the kernel most appopriate for the dataset. I predict the polynomial/RBF kernel will outperform the linear kernel as this data is not linearly seperable.
- For values of C I will base my range around the recommendation of "trying exponentially growing sequences of C" to identify good parameters. **(Hsu, C., Chang, C. and Lin, C., 2016)**

Linear Kernel

Below are the scores returned by the linear kernel using a wide range of values for C.

In [25]:
```
linearKernel = Pipeline([("scaler", RobustScaler()), ("svc", svm.SVC(kernel="linear"))])

params = {
    'svc__C': [0.1, 1, 10, 100, 1000]
}

linearGrid = GridSearchCV(linearKernel, params, cv=5,verbose=True,n_jobs=-1,scoring='f1')
linearGrid.fit(X_train,y_train)
```

```
Fitting 5 folds for each of 5 candidates, totalling 25 fits
```

Out[25]:
```
GridSearchCV(cv=5,
             estimator=Pipeline(steps=[('scaler', RobustScaler()),
                                       ('svc', SVC(kernel='linear'))]),
             n_jobs=-1, param_grid={'svc__C': [0.1, 1, 10, 100, 1000]},
             scoring='f1', verbose=True)
```

In [26]:
```
print("The parameters for the best score the grid search returned were: " + str(linearGrid.best_params_))
print("Best score the grid search returned was: "+ str(linearGrid.best_score_))
print("\n\n")
print("Scores on training data:")
y_pred = linearGrid.predict(X_train)
print(confusion_matrix(y_train, y_pred))
print(classification_report(y_train, y_pred))
print("\n\n")
print("Scores on testing data:")
y_pred = linearGrid.predict(X_test)
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test,y_pred))
```

```
The parameters for the best score the grid search returned were: {'svc__C': 10}
Best score the grid search returned was: 0.6207261292166952



Scores on training data:
[[221  29]
 [ 48  74]]
              precision    recall  f1-score   support

           0       0.82      0.88      0.85       250
           1       0.72      0.61      0.66       122

    accuracy                           0.79       372
   macro avg       0.77      0.75      0.75       372
weighted avg       0.79      0.79      0.79       372




Scores on testing data:
[[98  7]
 [24 31]]
              precision    recall  f1-score   support

           0       0.80      0.93      0.86       105
           1       0.82      0.56      0.67        55

    accuracy                           0.81       160
   macro avg       0.81      0.75      0.77       160
weighted avg       0.81      0.81      0.80       160
```

Polynomial Kernel

Below are the scores returned by the polynomial kernel using a wide range of values for C and degree. The degree will be searched between 1-10, this gives a nice range of values allowing for both straighter and more flexible decision boundaries.

In [27]:
```
polynomialKernel = Pipeline([("scaler", RobustScaler()), ("svc", svm.SVC(kernel="poly"))])

params = {
    'svc__C': [0.1, 1, 10, 100, 1000],
    'svc__degree' : [2,3,4,5,6,7,8,9,10]
}

polynomialGrid = GridSearchCV(polynomialKernel, params, cv=5,verbose=True,n_jobs=-1,scoring='f1')
```

In [28]:
```
polynomialGrid.fit(X_train,y_train)
```

```
Fitting 5 folds for each of 45 candidates, totalling 225 fits
```

Out[28]:
```
GridSearchCV(cv=5,
             estimator=Pipeline(steps=[('scaler', RobustScaler()),
                                       ('svc', SVC(kernel='poly'))]),
             n_jobs=-1,
             param_grid={'svc__C': [0.1, 1, 10, 100, 1000],
                         'svc__degree': [2, 3, 4, 5, 6, 7, 8, 9, 10]},
             scoring='f1', verbose=True)
```

In [29]:
```
print("Best parameters the grid search returned were: " + str(polynomialGrid.best_params_))
print("Best score the grid search returned was: "+ str(polynomialGrid.best_score_))
print("\n\n")
print("Scores on training data:")
y_pred = polynomialGrid.predict(X_train)
print(confusion_matrix(y_train, y_pred))
print(classification_report(y_train, y_pred))
print("\n\n")
print("Scores on testing data:")
y_pred = polynomialGrid.predict(X_test)
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test,y_pred))
```

```
Best parameters the grid search returned were: {'svc__C': 100, 'svc__degree': 3}
Best score the grid search returned was: 0.5161171177128623



Scores on training data:
[[243   7]
 [ 37  85]]
              precision    recall  f1-score   support

           0       0.87      0.97      0.92       250
           1       0.92      0.70      0.79       122

    accuracy                           0.88       372
   macro avg       0.90      0.83      0.86       372
weighted avg       0.89      0.88      0.88       372




Scores on testing data:
[[90 15]
 [29 26]]
              precision    recall  f1-score   support

           0       0.76      0.86      0.80       105
           1       0.63      0.47      0.54        55

    accuracy                           0.73       160
   macro avg       0.70      0.66      0.67       160
weighted avg       0.71      0.72      0.71       160
```

RBF Kernel

Below are the scores returned by the RBF kernel using a wide range of values for C and gamma.

In [30]:
```
RBFKernel = Pipeline([("scaler", RobustScaler()), ("svc", svm.SVC(kernel="rbf"))])

params = {
    'svc__C': [0.1, 1, 10, 100, 1000],
    'svc__gamma' : [0.001, 0.01, 0.1, 0.5, 1.0, 10.0, 50, 100, 'scale', 'auto']
}

RBFGrid = GridSearchCV(RBFKernel, params, cv=5,verbose=True,n_jobs=-1,scoring='f1')
```

In [31]:
```
RBFGrid.fit(X_train,y_train)
```

```
Fitting 5 folds for each of 50 candidates, totalling 250 fits
```

Out[31]:
```
GridSearchCV(cv=5,
             estimator=Pipeline(steps=[('scaler', RobustScaler()),
                                       ('svc', SVC())]),
             n_jobs=-1,
             param_grid={'svc__C': [0.1, 1, 10, 100, 1000],
                         'svc__gamma': [0.001, 0.01, 0.1, 0.5, 1.0, 10.0, 50,
                                        100, 'scale', 'auto']},
             scoring='f1', verbose=True)
```

In [32]:
```
print("Best parameters the grid search returned were: " + str(RBFGrid.best_params_))
print("Best score the grid search returned was: "+ str(RBFGrid.best_score_))
print("\n\n")
print("Scores on training data:")
y_pred = RBFGrid.predict(X_train)
print(confusion_matrix(y_train, y_pred))
print(classification_report(y_train, y_pred))
print("\n\n")
print("Scores on testing data:")
y_pred = RBFGrid.predict(X_test)
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test,y_pred))
```

```
Best parameters the grid search returned were: {'svc__C': 100, 'svc__gamma': 0.001}
Best score the grid search returned was: 0.607120887201777



Scores on training data:
[[223  27]
 [ 49  73]]
              precision    recall  f1-score   support

           0       0.82      0.89      0.85       250
           1       0.73      0.60      0.66       122

    accuracy                           0.80       372
   macro avg       0.77      0.75      0.76       372
weighted avg       0.79      0.80      0.79       372




Scores on testing data:
[[96  9]
 [24 31]]
              precision    recall  f1-score   support

           0       0.80      0.91      0.85       105
           1       0.78      0.56      0.65        55

    accuracy                           0.79       160
   macro avg       0.79      0.74      0.75       160
weighted avg       0.79      0.79      0.78       160
```

Kernel Comparison and Selection:

Comparing the C Values and Considering the Degree of the Polynomial kernel:

- Linear and RBF:
  - The grid search returned a C of 10 to be the best performing for both the linear and RBF kernel, meaning with a C of ten the trade-off between margin size and misclassification is the most even compared to the other C values I fed into the grid search. The C value controls the penalty for misclassification. This suggests the higher C values suffered from potentially overfitting on the training set due to having too small a margin, leading to poor performance on the test set. It also suggests the lower values of C lead to too many misclassifications, meaning the margin was too large to avoid a higher volume of misclassification.
- Polynomial:
  - Interestingly, the polynomial kernel found a C of 100 to be most appropriate. This is perhaps due to the fact the degree the grid search had returned was 3, meaning the decision boundary can be a little more flexible and potentially requires a bit of a bigger

margin to maintain the margin/misclassification trade-off. It is also worth noting when I allowed a degree of 1 in the grid search it would choose this degree, meaning it would essentially be acting as a linear kernel and suggests the polynomial kernel is not a good fit for the problem because the data is somewhat linearly separable before applying the kernel trick of translating to a higher degree.

- Considering the gamma for RBF:
  - The ideal gamma value chosen by the grid search for RBF was 0.001, which can be considered a very low value for gamma, meaning the kernel is considering points both close and very far away in the creation of its decision boundary. This suggests that the ideal decision boundary will be somewhat straight, which also further explains why the linear kernel performs so well on the dataset, contrary to my predictions.

Comparison and Analysis of the Kernels Performance:

After utilising grid search with a wide range of values and comparing their results I think the optimal kernel for this dataset would be, to my surprise, the linear kernel. The precision, recall and f1-score of the linear kernel outperform both the RBF and polynomial kernels in every category. It is because of this I believe the linear kernel to perhaps be the optimal kernel for the problem as it has the lowest computational cost and training speed and delivers consistently better results than the polynomial and RBF kernel with the tested parameters.

To compare the scores of the kernels against the test and train data, it is clear the polynomial kernel suffers from overfitting to a much higher degree than the other two, which appear to have a somewhat even performance between train and test scores. The polynomial generally has an f1-score over 20 points higher on the train than the test, a clear indicator of overfitting a model onto the training data.

To take a more in-depth look at the general scores of the three kernels you can see predicting a positive outcome is considerably less accurate than predicting a negative one. Generally all three kernels manage an f1-score of at least 79 for predicting a negative outcome, where as predicting a positive outcome the f1-score can go into the low 50s for the polynomial kernel and mid 60s for RBF/linear kernel. To consider the real world this is perhaps because diabetes can be heavily influenced by genetics, such as type 2 diabetes being linked to family history which is not measured within the dataset.

To conclude I will be choosing linear kernel for the SVC on this dataset due to its low performance cost and relatively high performance metrics.

Further Optimisation with Linear Kernel:

I will now quickly run the linear kernel with a closer ranges of values in hopes to narrow down the most ideal C value, I am basing this off the current ideal C of 10.

In [33]:
```
linearKernel = Pipeline([("scaler", RobustScaler()), ("svc", svm.SVC(kernel="linear"))])

params = {
    'svc__C': [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
}

linearGrid = GridSearchCV(linearKernel, params, cv=5,verbose=True,n_jobs=-1,scoring='f1')
linearGrid.fit(X_train,y_train)
```

```
Fitting 5 folds for each of 11 candidates, totalling 55 fits
```

Out[33]:
```
GridSearchCV(cv=5,
             estimator=Pipeline(steps=[('scaler', RobustScaler()),
                                       ('svc', SVC(kernel='linear'))]),
             n_jobs=-1,
             param_grid={'svc__C': [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]},
             scoring='f1', verbose=True)
```

In [34]:
```
print("The parameters for the best score the grid search returned were: " + str(linearGrid.best_params_))
print("Best score the grid search returned was: "+ str(linearGrid.best_score_))
print("\n\n")
print("Scores on training data:")
y_pred = linearGrid.predict(X_train)
print(confusion_matrix(y_train, y_pred))
print(classification_report(y_train, y_pred))
print("\n\n")
print("Scores on testing data:")
y_pred = linearGrid.predict(X_test)
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test,y_pred))
```

```
The parameters for the best score the grid search returned were: {'svc__C': 5}
Best score the grid search returned was: 0.632392795883362
```

```
Scores on training data:
[[221  29]
 [ 47  75]]
              precision    recall  f1-score   support

           0       0.82      0.88      0.85       250
           1       0.72      0.61      0.66       122

    accuracy                           0.80       372
   macro avg       0.77      0.75      0.76       372
weighted avg       0.79      0.80      0.79       372



Scores on testing data:
[[97  8]
 [24 31]]
              precision    recall  f1-score   support

           0       0.80      0.92      0.86       105
           1       0.79      0.56      0.66        55

    accuracy                           0.80       160
   macro avg       0.80      0.74      0.76       160
weighted avg       0.80      0.80      0.79       160
```

In [35]:

```
linearKernel = Pipeline([("scaler", RobustScaler()), ("svc", svm.SVC(kernel="linear"))])

params = {
    'svc__C': [4, 4.25, 4.5, 4.75, 5, 5.25, 5.5, 5.75, 6]
}

linearGrid = GridSearchCV(linearKernel, params, cv=5,verbose=True,n_jobs=-1,scoring='f1')
linearGrid.fit(X_train,y_train)
```

```
Fitting 5 folds for each of 9 candidates, totalling 45 fits
```

Out[35]:

```
GridSearchCV(cv=5,
             estimator=Pipeline(steps=[('scaler', RobustScaler()),
                                       ('svc', SVC(kernel='linear'))]),
             n_jobs=-1,
             param_grid={'svc__C': [4, 4.25, 4.5, 4.75, 5, 5.25, 5.5, 5.75, 6]},
             scoring='f1', verbose=True)
```

In [36]:

```
print("The parameters for the best score the grid search returned were: " + str(linearGrid.best_params_))
print("Best score the grid search returned was: "+ str(linearGrid.best_score_))
print("\n\n")
print("Scores on training data:")
y_pred = linearGrid.predict(X_train)
print(confusion_matrix(y_train, y_pred))
print(classification_report(y_train, y_pred))
print("\n\n")
print("Scores on testing data:")
y_pred = linearGrid.predict(X_test)
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test,y_pred))
```

```
The parameters for the best score the grid search returned were: {'svc__C': 4.25}
Best score the grid search returned was: 0.6354069802805251



Scores on training data:
[[221  29]
 [ 47  75]]
              precision    recall  f1-score   support

           0       0.82      0.88      0.85       250
           1       0.72      0.61      0.66       122

    accuracy                           0.80       372
   macro avg       0.77      0.75      0.76       372
weighted avg       0.79      0.80      0.79       372
```

```
Scores on testing data:
[[97  8]
 [24 31]]
              precision    recall  f1-score   support

           0       0.80      0.92      0.86       105
           1       0.79      0.56      0.66        55

    accuracy                           0.80       160
   macro avg       0.80      0.74      0.76       160
weighted avg       0.80      0.80      0.79       160
```

In [37]:

```python
linearKernel = Pipeline([("scaler", RobustScaler()), ("svc", svm.SVC(kernel="linear"))])

params = {
    'svc__C': [4.10, 4.15, 4.2, 4.25, 4.3]
}

linearGrid = GridSearchCV(linearKernel, params, cv=5,verbose=True,n_jobs=-1,scoring='f1')
linearGrid.fit(X_train,y_train)
```

```
Fitting 5 folds for each of 5 candidates, totalling 25 fits
```

Out[37]:

```
GridSearchCV(cv=5,
             estimator=Pipeline(steps=[('scaler', RobustScaler()),
                                       ('svc', SVC(kernel='linear'))]),
             n_jobs=-1, param_grid={'svc__C': [4.1, 4.15, 4.2, 4.25, 4.3]},
             scoring='f1', verbose=True)
```

In [38]:

```python
print("The parameters for the best score the grid search returned were: " + str(linearGrid.best_params_))
print("Best score the grid search returned was: "+ str(linearGrid.best_score_))
print("\n\n")
print("Scores on training data:")
y_pred = linearGrid.predict(X_train)
print(confusion_matrix(y_train, y_pred))
print(classification_report(y_train, y_pred))
print("\n\n")
print("Scores on testing data:")
y_pred = linearGrid.predict(X_test)
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test,y_pred))
```

```
The parameters for the best score the grid search returned were: {'svc__C': 4.1}
Best score the grid search returned was: 0.6354069802805251



Scores on training data:
[[221  29]
 [ 47  75]]
              precision    recall  f1-score   support

           0       0.82      0.88      0.85       250
           1       0.72      0.61      0.66       122

    accuracy                           0.80       372
   macro avg       0.77      0.75      0.76       372
weighted avg       0.79      0.80      0.79       372



Scores on testing data:
[[97  8]
 [24 31]]
              precision    recall  f1-score   support

           0       0.80      0.92      0.86       105
           1       0.79      0.56      0.66        55

    accuracy                           0.80       160
   macro avg       0.80      0.74      0.76       160
weighted avg       0.80      0.80      0.79       160
```

After a quick operation closing in on a more specific C value with grid search I arrived at the C value of 4.1.

Final SVM Model

Below is the final model I arrived at through my experimentation with SVM on the diabetes dataset:

In [39]:
```
optimalSVMModel = Pipeline([("scaler", RobustScaler()), ("svc", svm.SVC(kernel="linear", C=4.1))])
optimalSVMModel.fit(X_train,y_train)
print("Scores on training data:")
y_pred = optimalSVMModel.predict(X_train)
print(confusion_matrix(y_train, y_pred))
print(classification_report(y_train, y_pred))
y_pred = optimalSVMModel.predict(X_test)
print("Scores on testing data:")
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

```
Scores on training data:
[[221  29]
 [ 47  75]]
              precision    recall  f1-score   support

           0       0.82      0.88      0.85       250
           1       0.72      0.61      0.66       122

    accuracy                           0.80       372
   macro avg       0.77      0.75      0.76       372
weighted avg       0.79      0.80      0.79       372

Scores on testing data:
[[97  8]
 [24 31]]
              precision    recall  f1-score   support

           0       0.80      0.92      0.86       105
           1       0.79      0.56      0.66        55

    accuracy                           0.80       160
   macro avg       0.80      0.74      0.76       160
weighted avg       0.80      0.80      0.79       160
```

Random Forest (Ensemble)

The second model I will train will be random forest, the hyper-parameters I will need to tune will be:

- Max Features (The number of features to consider)
- Max Depth (The maximum height the trees within the random forest can grow)
- Number of Estimators (The number of trees inside the random forest)
- Min Samples Split (The number of samples in a node that allow it to split into other nodes)
- Max Samples (The maximum ratio of samples each tree can use)

I have decided upon Random Forest for a similar reason to SVM, it is a widely used model in classification problems and again generally produces good results due to its versatility. For the criterion I am using gini as I am familiar with how it works and it is also commonly used. I will again use cross validation with 5 folds.

Also notable is the fact Random Forest does not require feature scaling as it is a tree based model.

In [40]:
```
randomForestModel = Pipeline([("randomForest", RandomForestClassifier(criterion="gini", random_state=1, max_sampl

params = {
    'randomForest__max_features': ['sqrt', 'log2', 2],
    'randomForest__max_depth' : [1, 2, 3, 4, 5],
    'randomForest__n_estimators': [25, 50, 75, 100, 150, 200]
}

randomForestGrid = GridSearchCV(randomForestModel, params, cv=5,verbose=True,n_jobs=-1,scoring='f1')
randomForestGrid.fit(X_train,y_train)
```

```
Fitting 5 folds for each of 90 candidates, totalling 450 fits
```

Out[40]:
```
GridSearchCV(cv=5,
             estimator=Pipeline(steps=[('randomForest',
                                        RandomForestClassifier(max_samples=0.6,
                                                               random_state=1))]),
             n_jobs=-1,
             param_grid={'randomForest__max_depth': [1, 2, 3, 4, 5],
                         'randomForest__max_features': ['sqrt', 'log2', 2],
                         'randomForest__n_estimators': [25, 50, 75, 100, 150,
                                                        200]},
             scoring='f1', verbose=True)
```

In [41]:
```python
print("The parameters for the best score the grid search returned were: " + str(randomForestGrid.best_params_))
print("Best score the grid search returned was: "+ str(randomForestGrid.best_score_))
print("\n\n")
print("Scores on training data:")
y_pred = randomForestGrid.predict(X_train)
print(confusion_matrix(y_train, y_pred))
print(classification_report(y_train, y_pred))
print("\n\n")
print("Scores on testing data:")
y_pred = randomForestGrid.predict(X_test)
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test,y_pred))
```

```
The parameters for the best score the grid search returned were: {'randomForest__max_depth': 5, 'randomForest__ma
x_features': 'sqrt', 'randomForest__n_estimators': 25}
Best score the grid search returned was: 0.639952380952381



Scores on training data:
[[233  17]
 [ 28  94]]
              precision    recall  f1-score   support

           0       0.89      0.93      0.91       250
           1       0.85      0.77      0.81       122

    accuracy                           0.88       372
   macro avg       0.87      0.85      0.86       372
weighted avg       0.88      0.88      0.88       372




Scores on testing data:
[[95 10]
 [26 29]]
              precision    recall  f1-score   support

           0       0.79      0.90      0.84       105
           1       0.74      0.53      0.62        55

    accuracy                           0.78       160
   macro avg       0.76      0.72      0.73       160
weighted avg       0.77      0.78      0.76       160
```

Straight away I can see the Random Forest is delivering a decent performance with the current values. I played around with max_samples and decided on 0.6 as it is a good amount for training on a single tree to get results reflective of the dataset.

The grid search returning the highest available max-depth is perhaps the reason for the model overfitting, the bigger the depth the more chance you have of overfitting since the model will be highly tuned towards the boundaries of the training data. It is also notable that max_depth might be considered one of the most important parameters in a Random Forest as it has the potential to vastly improve performance, but also to overfit the model, so a good balance is important. The model does perform worse on the testing data compared to the training data, and is consequently overfitting.

The sqrt (√number of features) for max features is common in Random Forest classifiers, so it is not surprising it has been selected here, having a higher number of features for each tree could require them to need a lot of depth to ensure leaves maintain a low gini index. This high depth can in turn lead to overfitting.

For number of estimators I chose higher numbers as the trade-off of an increase in the computation time of the model for real-time classifications is not important for what I am doing. It does appear, however, the model requires a small amount of estimators regardless and any more will offer little improvement.

I will now further tune max depth and number of estimators, along with the min samples split parameter.

Tuning the min number of samples parameter can aid the performance of the model as it manages overfitting much like max depth as it prevents too many splits occuring. I will search between 4 and 8 as too high a value can actually cause underfitting as the final node can still contain a considerable amount of samples of multiple classes, a low value of course overfits as it will create too many splits, in a similar fashion to a large max depth. I am searching higher values than the default 2 as the model is currently overfitting.

In [42]:
```
randomForestModel = Pipeline([("randomForest", RandomForestClassifier(criterion="gini", random_state=1, max_sampl

params = {
    'randomForest__max_depth' : [2,3,4,5],
    'randomForest__n_estimators': [25, 50, 75, 100],
    'randomForest__min_samples_split' : [2, 5, 8, 10, 12]
}

randomForestGrid = GridSearchCV(randomForestModel, params, cv=5,verbose=True,n_jobs=-1,scoring='f1')
randomForestGrid.fit(X_train,y_train)
```

```
Fitting 5 folds for each of 80 candidates, totalling 400 fits
```

Out[42]:
```
GridSearchCV(cv=5,
             estimator=Pipeline(steps=[('randomForest',
                                        RandomForestClassifier(max_features='sqrt',
                                                               max_samples=0.6,
                                                               random_state=1))]),
             n_jobs=-1,
             param_grid={'randomForest__max_depth': [2, 3, 4, 5],
                         'randomForest__min_samples_split': [2, 5, 8, 10, 12],
                         'randomForest__n_estimators': [25, 50, 75, 100]},
             scoring='f1', verbose=True)
```

In [43]:
```
print("The parameters for the best score the grid search returned were: " + str(randomForestGrid.best_params_))
print("Best score the grid search returned was: "+ str(randomForestGrid.best_score_))
print("\n\n")
print("Scores on training data:")
y_pred = randomForestGrid.predict(X_train)
print(confusion_matrix(y_train, y_pred))
print(classification_report(y_train, y_pred))
print("\n\n")
print("Scores on testing data:")
y_pred = randomForestGrid.predict(X_test)
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test,y_pred))
```

```
The parameters for the best score the grid search returned were: {'randomForest__max_depth': 5, 'randomForest__mi
n_samples_split': 2, 'randomForest__n_estimators': 25}
Best score the grid search returned was: 0.639952380952381



Scores on training data:
[[233  17]
 [ 28  94]]
              precision    recall  f1-score   support

           0       0.89      0.93      0.91       250
           1       0.85      0.77      0.81       122

    accuracy                           0.88       372
   macro avg       0.87      0.85      0.86       372
weighted avg       0.88      0.88      0.88       372




Scores on testing data:
[[95 10]
 [26 29]]
              precision    recall  f1-score   support

           0       0.79      0.90      0.84       105
           1       0.74      0.53      0.62        55

    accuracy                           0.78       160
   macro avg       0.76      0.72      0.73       160
weighted avg       0.77      0.78      0.76       160
```

Following these results, the performance appears to have not improved, it still favours the default min sample split. This is unexpected as it could help combat the overfitting of the model by reducing the number of splits in some trees and consequently their depth. It is possible the Random Forest's weak classifiers are causing this overfitting, and since all the trees in a Random Forest have the same say towards the final prediction they hold just as much value as the strong classifiers.

I will consider a quick look at AdaBoost in hopes it can deliver improved performance as I have considered many parameters of the Random Forest classifier with little luck in improving accuracy.

Final Random Forest Model

Below is the final model I arrived at through my experimentation with Random Forest on the diabetes dataset:

In [44]:
```
finalRandomForestModel = Pipeline([("randomForest", RandomForestClassifier(criterion="gini", max_depth=5, min_sam
finalRandomForestModel.fit(X_train,y_train)
y_pred = finalRandomForestModel.predict(X_train)
print("Scores on testing data:")
print(confusion_matrix(y_train, y_pred))
print(classification_report(y_train, y_pred))
y_pred = finalRandomForestModel.predict(X_test)
print("Scores on testing data:")
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

```
Scores on testing data:
[[233  17]
 [ 28  94]]
              precision    recall  f1-score   support

           0       0.89      0.93      0.91       250
           1       0.85      0.77      0.81       122

    accuracy                           0.88       372
   macro avg       0.87      0.85      0.86       372
weighted avg       0.88      0.88      0.88       372

Scores on testing data:
[[95 10]
 [26 29]]
              precision    recall  f1-score   support

           0       0.79      0.90      0.84       105
           1       0.74      0.53      0.62        55

    accuracy                           0.78       160
   macro avg       0.76      0.72      0.73       160
weighted avg       0.77      0.78      0.76       160
```

AdaBoost

I hope AdaBoost can find improved scores from my Random Forest model, the two models are similar however AdaBoost allows for different decision trees to have a different amount of influence on the training of the model based on their effectiveness. The parameters I will be tuning here will be the learning rate and number of estimators, the learning rate decides how much the next version of the model will change from the previous version. Number of estimators is the same as seen in Random Forest. I will simply use the recommended base estimator of the 1 depth decision tree/stump, this is because AdaBoost works best with many weak learners, which a depth of 1 would theoretically achieve.

In [45]:
```
adaBoostModel = Pipeline([("adaBoost", AdaBoostClassifier(random_state=1))])

params = {
    'adaBoost__learning_rate' : [0.0001, 0.001, 0.01, 0.1, 1, 2],
    'adaBoost__n_estimators' : [10, 50, 100, 200, 500, 1000]
}

adaBoostGrid = GridSearchCV(adaBoostModel, params, cv=5,verbose=True,n_jobs=-1,scoring='f1')
adaBoostGrid.fit(X_train,y_train)
```

```
Fitting 5 folds for each of 36 candidates, totalling 180 fits
```

Out[45]:
```
GridSearchCV(cv=5,
             estimator=Pipeline(steps=[('adaBoost',
                                   AdaBoostClassifier(random_state=1))]),
             n_jobs=-1,
             param_grid={'adaBoost__learning_rate': [0.0001, 0.001, 0.01, 0.1,
                                                      1, 2],
                         'adaBoost__n_estimators': [10, 50, 100, 200, 500,
                                                     1000]},
             scoring='f1', verbose=True)
```

In [46]:

```
print("The parameters for the best score the grid search returned were: " + str(adaBoostGrid.best_params_))
print("Best score the grid search returned was: "+ str(adaBoostGrid.best_score_))
print("\n\n")
print("Scores on training data:")
y_pred = adaBoostGrid.predict(X_train)
print(confusion_matrix(y_train, y_pred))
print(classification_report(y_train, y_pred))
print("\n\n")
print("Scores on testing data:")
y_pred = adaBoostGrid.predict(X_test)
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test,y_pred))
```

```
The parameters for the best score the grid search returned were: {'adaBoost__learning_rate': 1, 'adaBoost__n_esti
mators': 500}
Best score the grid search returned was: 0.6238302277432712



Scores on training data:
[[250   0]
 [  0 122]]
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       250
           1       1.00      1.00      1.00       122

    accuracy                           1.00       372
   macro avg       1.00      1.00      1.00       372
weighted avg       1.00      1.00      1.00       372




Scores on testing data:
[[86 19]
 [27 28]]
              precision    recall  f1-score   support

           0       0.76      0.82      0.79       105
           1       0.60      0.51      0.55        55

    accuracy                           0.71       160
   macro avg       0.68      0.66      0.67       160
weighted avg       0.70      0.71      0.71       160
```

The model is overfitting horribly, this perhaps suggests the dataset has a lot of noise and the model is focusing on those noisy features/values over the important sections of the dataset. The model may not be appropriate for the problem without applying further performance improving techniques/alogrithms or modifications to the dataset. A potential cause of this severe overfitting could be the fact that predicting a negative is generally a lot easier to predict than a positive using the dataset, this has also been the case for all 3 models I have trained thus far. Consequently, some weak learners may actually perform well on predicting the negative cases which could harm AdaBoost's effectiveness. This could be further evidenced in the almost equal number of false and positive negatives within the confusion matrix for the test data, the model is essentially as accurate as a 50/50 guess on predicting negative outcomes.

I chose my selection of potential learning rates with the risk of a model which changes too greatly at each interval in mind, resulting in inaccuracy, hence why they are mostly somewhat low values. This of course has not worked when looking at the results, and the best learning rate returned by the grid search is the value of 1, which is a somewhat standard learning rate. This is good as it is a nice middle ground between too small and too large which means the model can tune itself at a comfortable rate, too low a learning rate requires more computing power for a similar outcome as the optimal learning rate.

The 500 estimators being selected suggests this is around the cut off point where any added estimators add very little to the model, in a similar fashion to Random Forest.

The final scores on the test data from the model are the lowest seen yet, whilst the scores on the training data are the highest. Again, this shows an extreme case of overfitting and suggests AdaBoost might not be optimal for the dataset.

Comparison of the SVM and Random Forest Models

SVM Final Model's Performance on Train and Test Data:

In [47]:

```
print("Scores on training data:")
y_pred = optimalSVMModel.predict(X_train)
print(confusion_matrix(y_train, y_pred))
print(classification_report(y_train, y_pred))
y_pred = optimalSVMModel.predict(X_test)
print("\n\n")
print("Scores on testing data:")
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

```
Scores on training data:
[[221  29]
 [ 47  75]]
              precision    recall  f1-score   support

           0       0.82      0.88      0.85       250
           1       0.72      0.61      0.66       122

    accuracy                           0.80       372
   macro avg       0.77      0.75      0.76       372
weighted avg       0.79      0.80      0.79       372




Scores on testing data:
[[97  8]
 [24 31]]
              precision    recall  f1-score   support

           0       0.80      0.92      0.86       105
           1       0.79      0.56      0.66        55

    accuracy                           0.80       160
   macro avg       0.80      0.74      0.76       160
weighted avg       0.80      0.80      0.79       160
```

Random Forest Final Model's Performance on Train and Test Data:

In [48]:

```
y_pred = finalRandomForestModel.predict(X_train)
print("Scores on testing data:")
print(confusion_matrix(y_train, y_pred))
print(classification_report(y_train, y_pred))
y_pred = finalRandomForestModel.predict(X_test)
print("\n\n")
print("Scores on testing data:")
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

```
Scores on testing data:
[[233  17]
 [ 28  94]]
              precision    recall  f1-score   support

           0       0.89      0.93      0.91       250
           1       0.85      0.77      0.81       122

    accuracy                           0.88       372
   macro avg       0.87      0.85      0.86       372
weighted avg       0.88      0.88      0.88       372




Scores on testing data:
[[95 10]
 [26 29]]
              precision    recall  f1-score   support

           0       0.79      0.90      0.84       105
           1       0.74      0.53      0.62        55
    accuracy                           0.78       160
   macro avg       0.76      0.72      0.73       160
weighted avg       0.77      0.78      0.76       160
```

Comparison:

Comparing the Scores of the Two Models:

I could perhaps begin my comparison by looking at the performance of the two tuned models. The linear SVM has a better average performance on the test data in precision, recall and f1-score. Consequently, it would not be unfair to state it performs better on the given dataset in classifying unseen data accurately. Both the models performance, however, is somewhat unimpressive as they both get around 80 for all 3 performance metrics of precision, recall and f1-score. Interesting to note is the Random Forest suffers severely from overfitting, its scores on the training data come in at around a very acceptable 90 but drop to the 70s on the test data. If I managed to reduce this overfitting its very possible the Random Forest would outperform the SVM so it is a shame I was unable to do so.

It could be valid to say that the SVM model is capturing more of a macro perspective on the problem whilst the Random Forest is capturing more of a micro perspective by overfitting on the training data.

Tuning Process of Both Models:

One benefit you might consider of the SVM model was that the linear kernel only requires one parameter to be tuned, consequently its much easier to realise the optimal performance of that model, on the other hand, Random Forest has many parameters which have an effect on model performance meaning it can be a lot more of an exhaustive search for the optimal parameters. Another perspective here however could be that the many parameters of Random Forest makes it a much more adaptive algorithm for different datasets, of course, the many kernels for SVM means it also has this adaptivity but for linear kernel it can be considered somewhat constricted. It is worth noting even when using polynomial or RBF kernel for SVM it has much less important parameters you must tune for performance improvement compared to Random Forest.

Advantages of SVM:

- The model is very versatile and can work well with a lot of datasets.
- Relatively simple model to understand.
- Easy to tune due to the small number of important parameters.
- Linear SVM is fast to train, especially compared to other kernel options.
- SVM can handle many dimensions of data relatively well depending on the kernel. (For high dimensions RBF generally considered)

Disadvantages of SVM:

- Struggles with large datasets for reasons such as the time complexity.
- Sensitive to noise within the dataset. (Such as in the diabetes dataset where positive and negative outcomes often overlap.)

Advantages of Random Forest:

- There are ways to measure how effective each feature is in improving the overall impurity in the forest, meaning you can effectively dispose of irrelevant features and helping in feature selection.
- The model is very versatile and can work well with a lot of datasets.
- Relatively simple model to understand.
- Can work for both classification and regression.

Disadvantages of Random Forest:

- If the model requires a lot of estimators to be effective, it can become ineffective for real time classification due to the time complexity.
- With the diabetes dataset it appears to suffer from overfitting, perhaps this can be overcome with better feature selection.
- Sensitive to noise within the dataset.

Conclusion:

To conclude both models have their uses, many of these uses overlap, such as they are both often used to gain a better insight into the data rather than as a final model. In this case the SVM outperformed the Random Forest and should theroretically perform faster due to it using the linear kernel. That being said, Random Forest was perhaps focusing on the noise in the dataset, so with some further pre-processing it could very well outperform the SVM. Something such as making sure there were even rows of negative and positive outcomes could be an example of further pre-processing.

Bibliography:

- Hsu, B., Chang, C. and Lin, C. (2016) A Practical Guide to Support Vector Classification. *Department of Computer Science, National Taiwan University* **[online]**. [Accessed 15/03/2023].