Classification

Supervised machine learning techniques involve training a model to operate on a set of *features* and predict a *label* using a dataset that includes some already-known label values. You can think of this function like this, in which ***y*** represents the label we want to predict and ***X*** represents the vector of features the model uses to predict it.

$$y = f([x_1, x_2, x_3, \dots])$$

Classification is a form of supervised machine learning in which you train a model to use the features (the ***x*** values in our function) to predict a label (***y***) that calculates the probability of the observed case belonging to each of a number of possible classes, and predicting an appropriate label. The simplest form of classification is *binary* classification, in which the label is 0 or 1, representing one of two classes; for example, "True" or "False", "Internal" or "External", "Profitable" or "Non-Profitable", and so on.

Binary Classification

In this notebook, we'll focus on an example of *binary classification*, where the model must predict a label that belongs to one of two classes. We'll train a binary classifier to predict whether or not a patient should be tested for diabetes based on their medical data.

Explore the data

Run the following cell to load a CSV file of patent data into a **Pandas** dataframe.

In [1]:
```
import pandas as pd

# load the training dataset

diabetes = pd.read_csv('diabetes.csv')
diabetes.head()
```

Out[1]:

	PatientID	Pregnancies	PlasmaGlucose	DiastolicBloodPressure	TricepsThickness	SerumInsu
0	1354778	0	171	80	34	
1	1147438	8	92	93	47	
2	1640031	7	115	47	52	
3	1883350	9	103	78	25	3
4	1424119	1	85	59	27	

This data consists of diagnostic information about some patients who have been tested for diabetes. Scroll to the right if necessary, and note that the final column in the dataset (**Diabetic**) contains the value ***0*** for patients who tested negative for diabetes, and ***1*** for patients

who tested positive. This is the label that we will train our model to predict; most of the other columns (**Pregnancies**, **PlasmaGlucose**, **DiastolicBloodPressure**, and so on) are the features we will use to predict the **Diabetic** label.

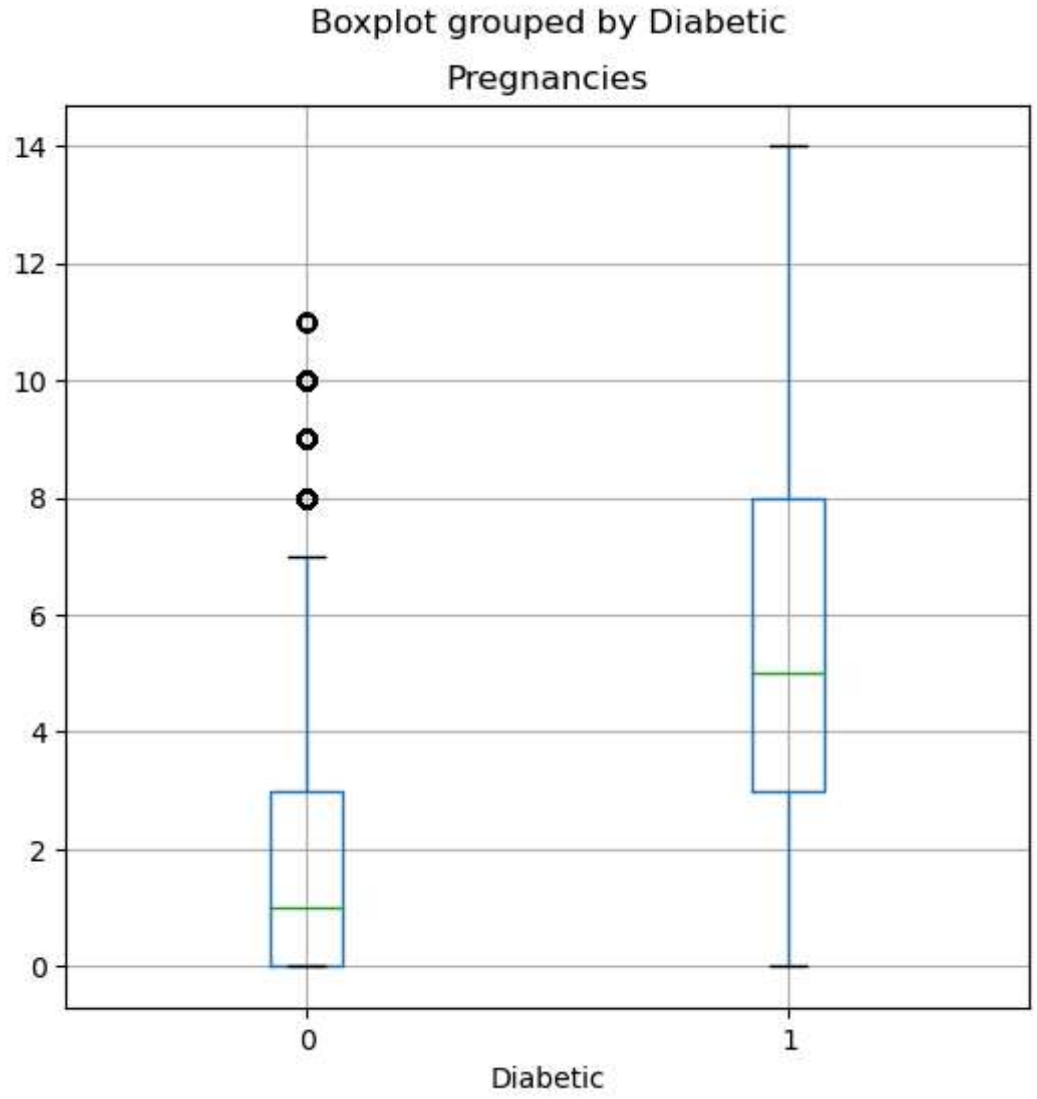
Let's separate the features from the labels - we'll call the features ***X*** and the label ***y***:

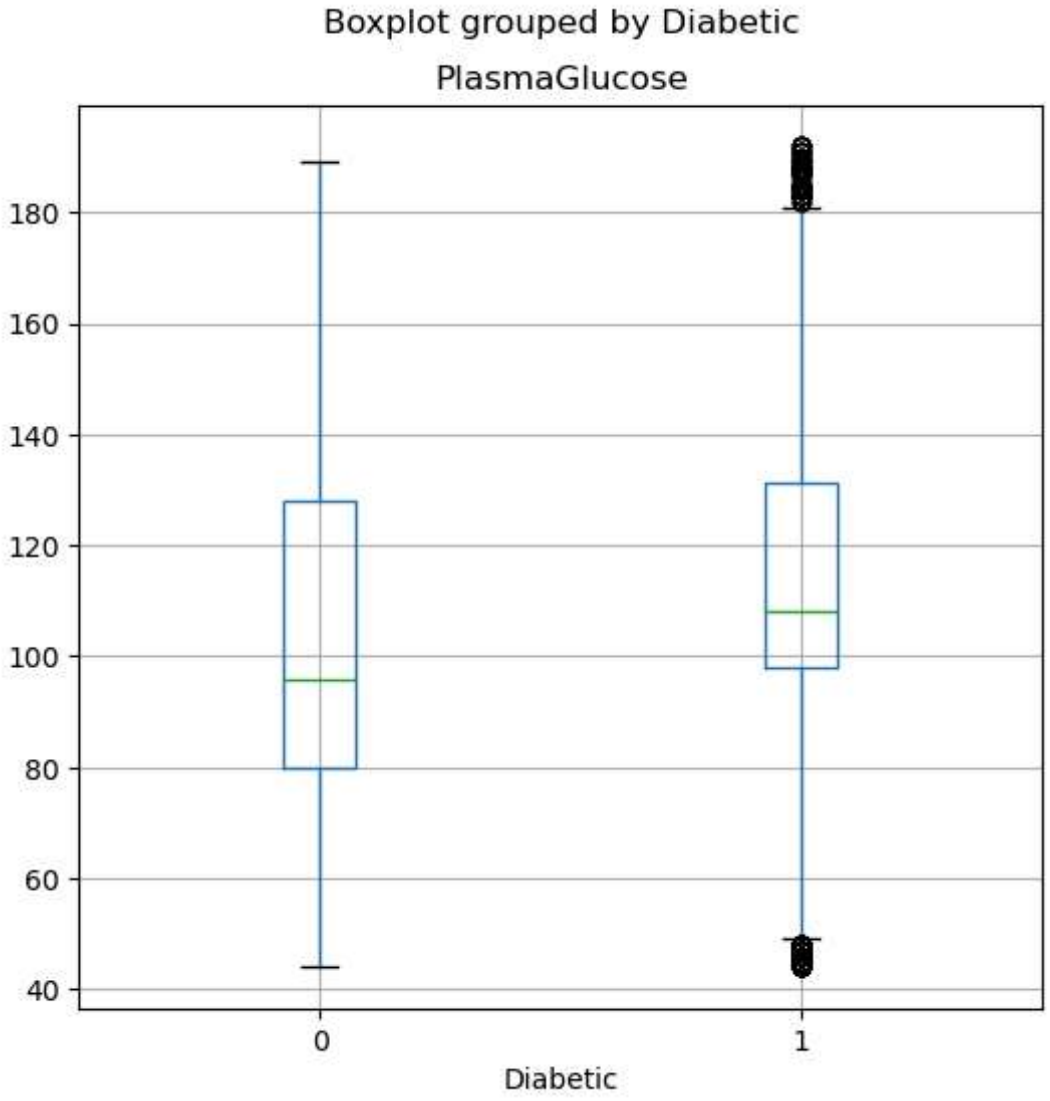
In [2]:
```
# Separate features and labels
features = ['Pregnancies','PlasmaGlucose','DiastolicBloodPressure','TricepsTh
label = 'Diabetic'
X, y = diabetes[features].values, diabetes[label].values

for n in range(0,4):
    print("Patient", str(n+1), "\n  Features:",list(X[n]), "\n  Label:", y[n]
```

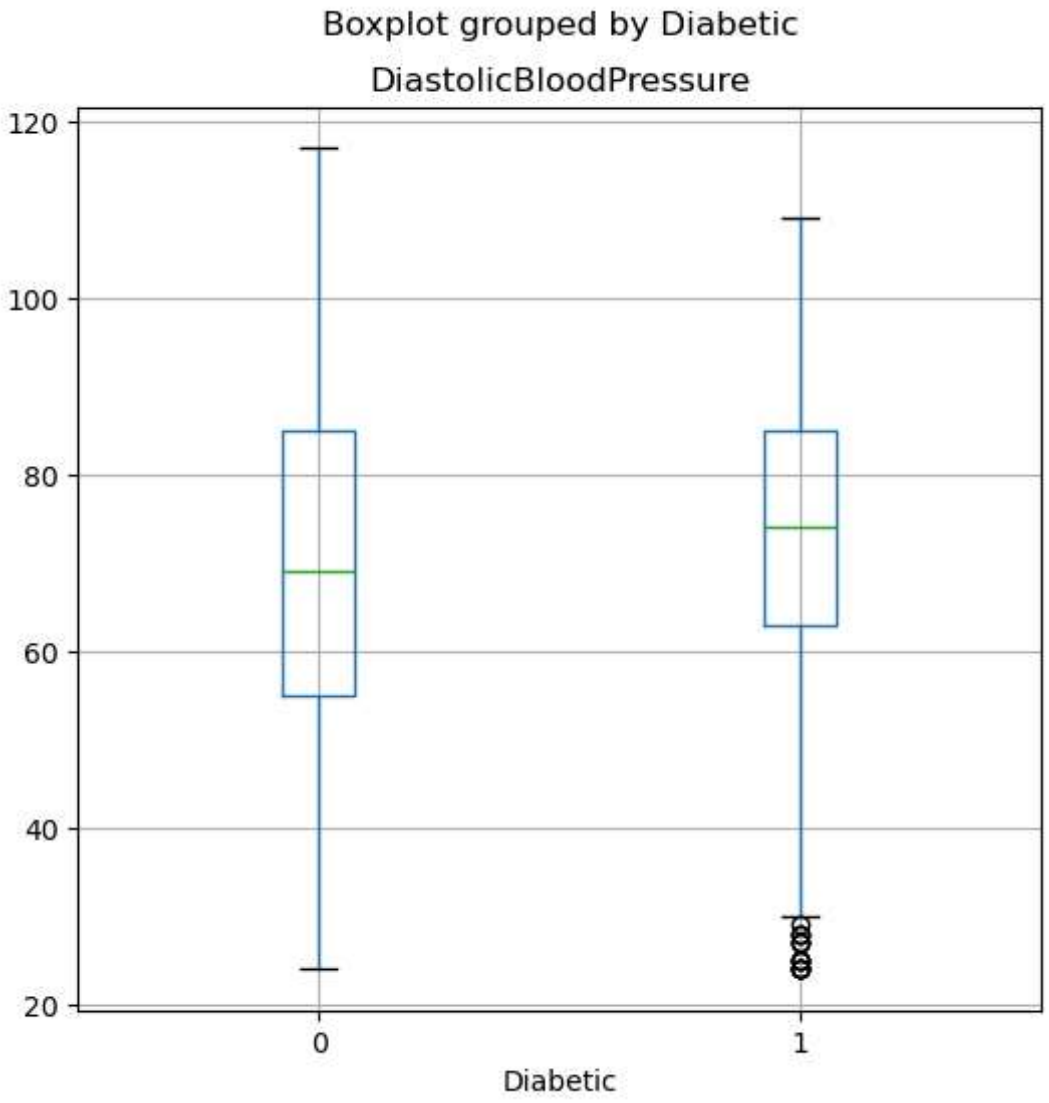
```
Patient 1
  Features: [0.0, 171.0, 80.0, 34.0, 23.0, 43.50972593, 1.213191354, 21.0]
  Label: 0
Patient 2
  Features: [8.0, 92.0, 93.0, 47.0, 36.0, 21.24057571, 0.158364981, 23.0]
  Label: 0
Patient 3
  Features: [7.0, 115.0, 47.0, 52.0, 35.0, 41.51152348, 0.079018568, 23.0]
  Label: 0
Patient 4
  Features: [9.0, 103.0, 78.0, 25.0, 304.0, 29.58219193, 1.282869847, 43.0]
  Label: 1
```

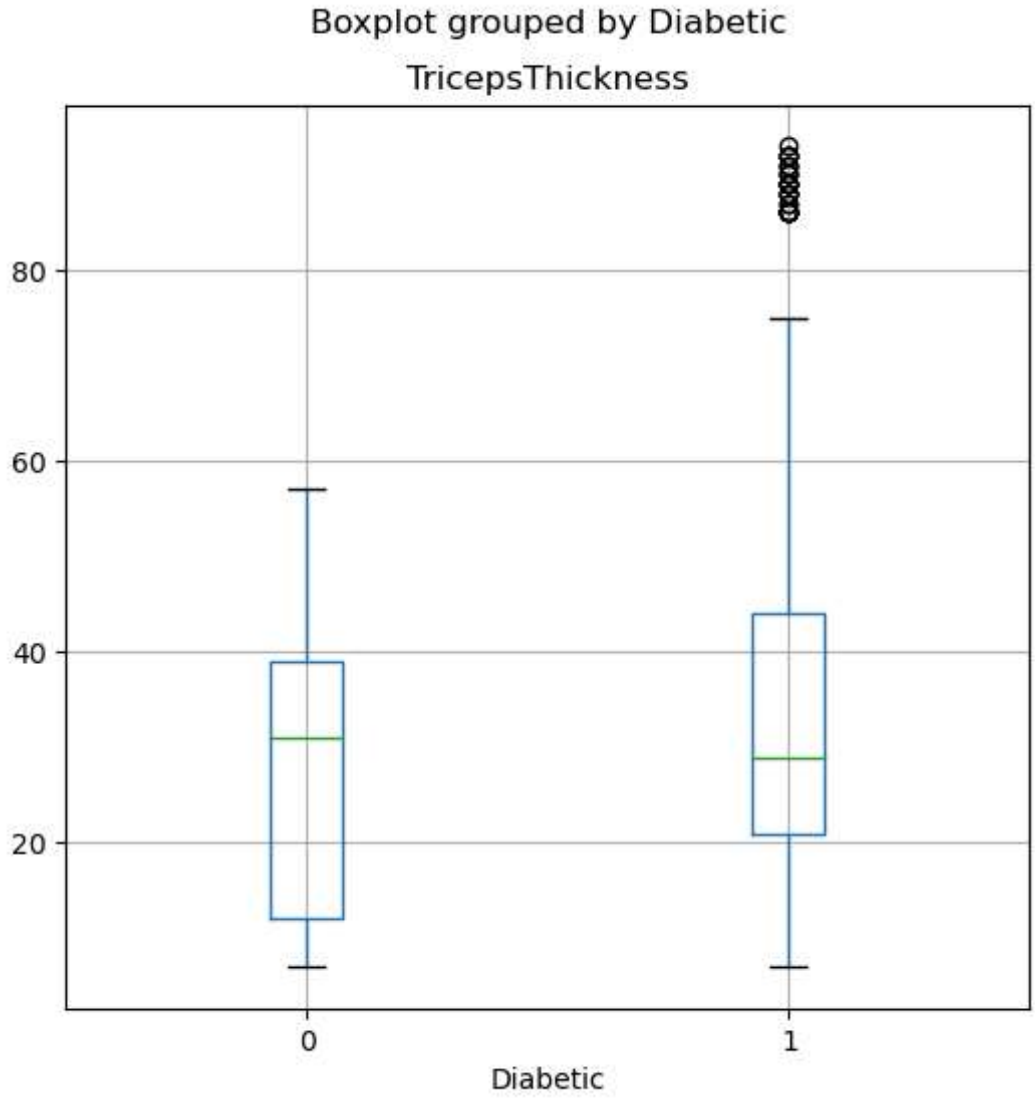
Now let's compare the feature distributions for each label value.

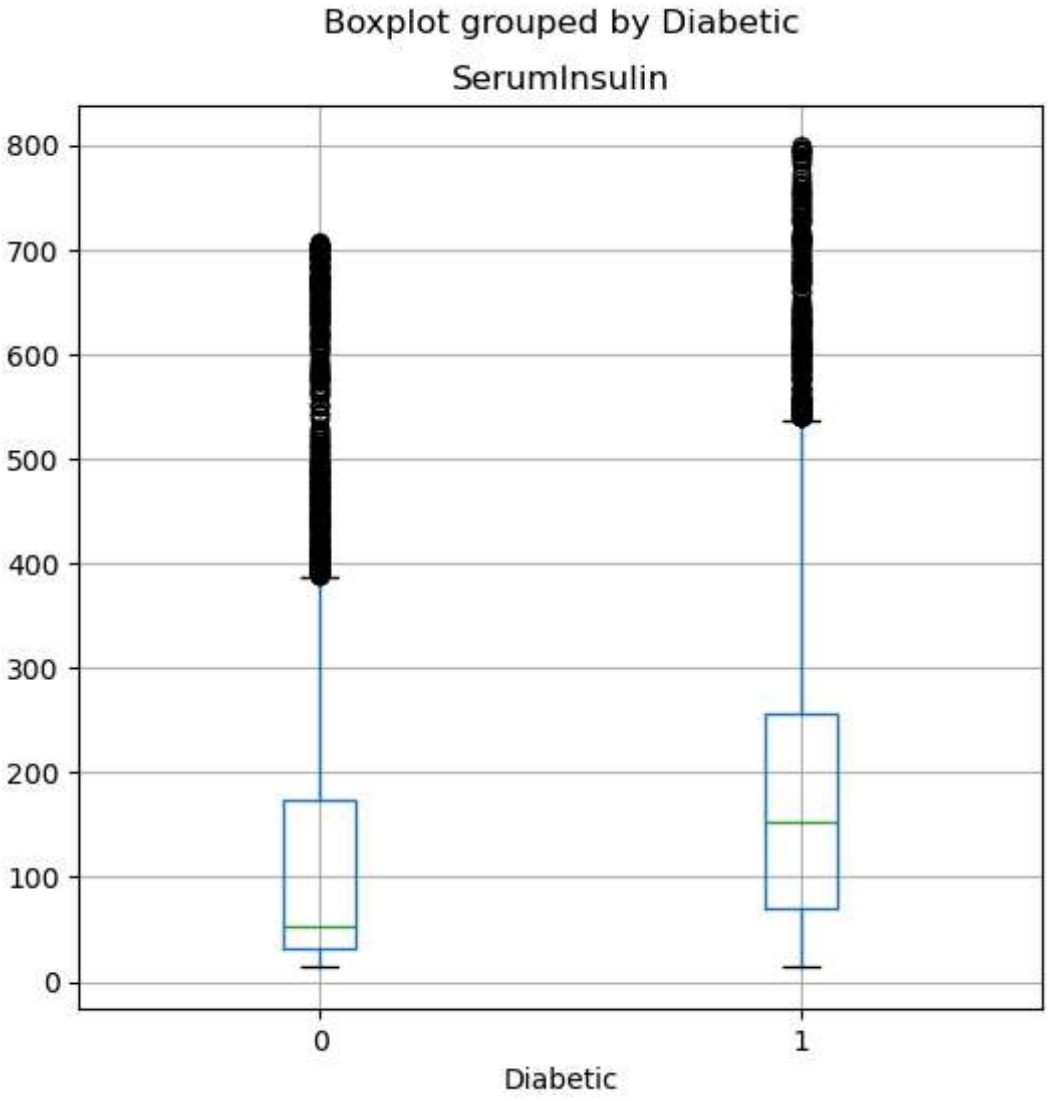
In [3]:
```
from matplotlib import pyplot as plt
%matplotlib inline

features = ['Pregnancies','PlasmaGlucose','DiastolicBloodPressure','TricepsTh
for col in features:
    diabetes.boxplot(column=col, by='Diabetic', figsize=(6,6))
    plt.title(col)
plt.show()
```

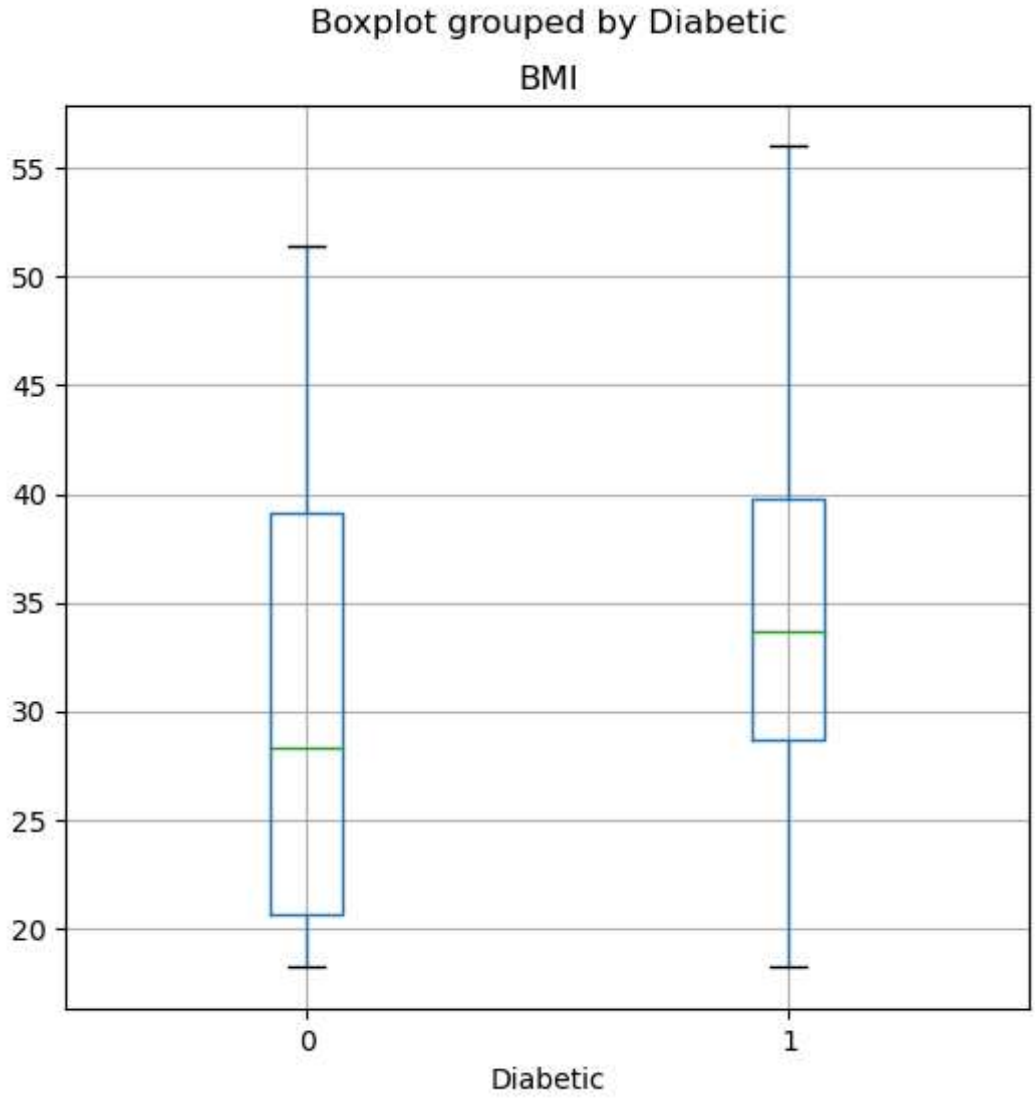


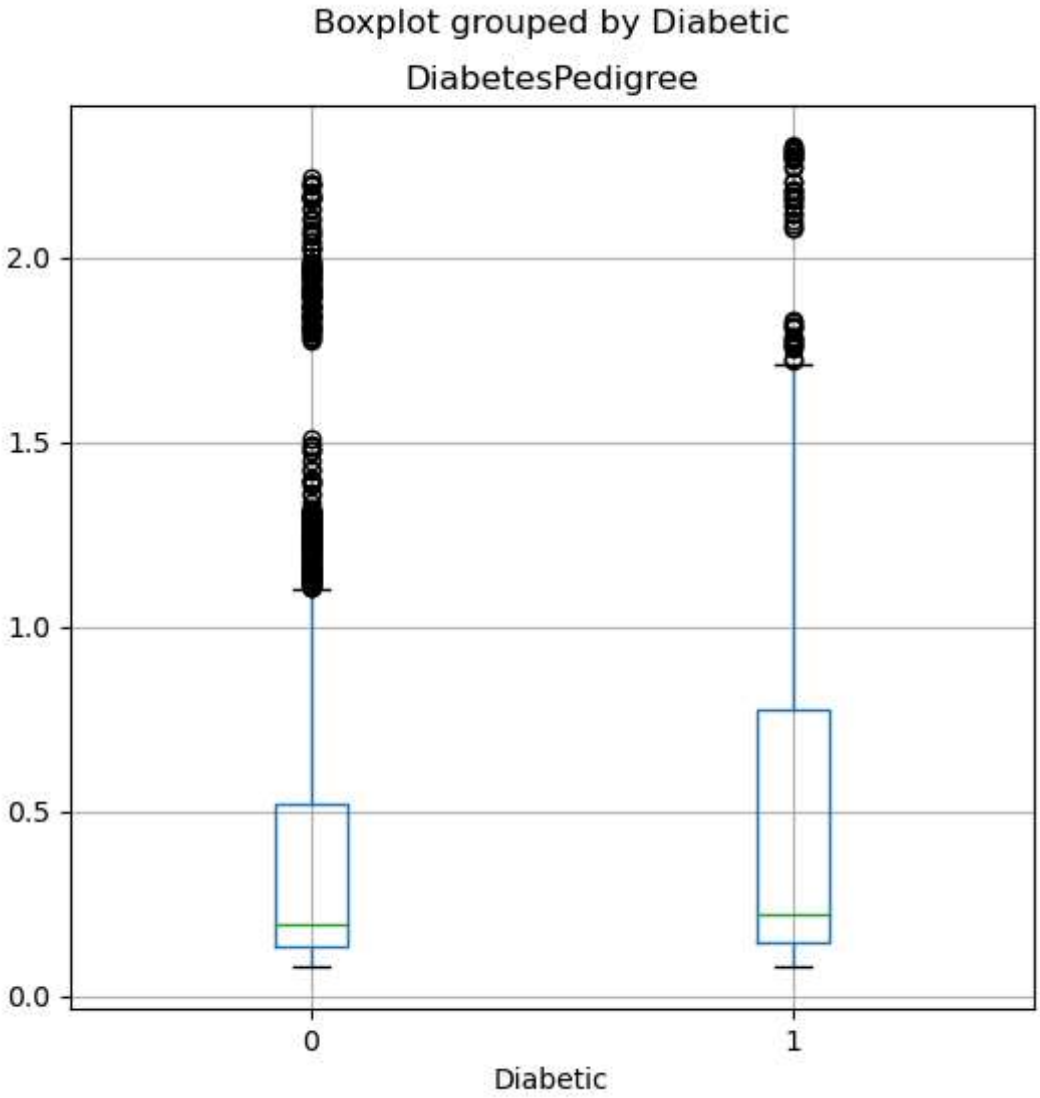

Boxplot grouped by Diabetic
PlasmaGlucose
180
160
140
120
100
80
60
40
0
1
Diabetic

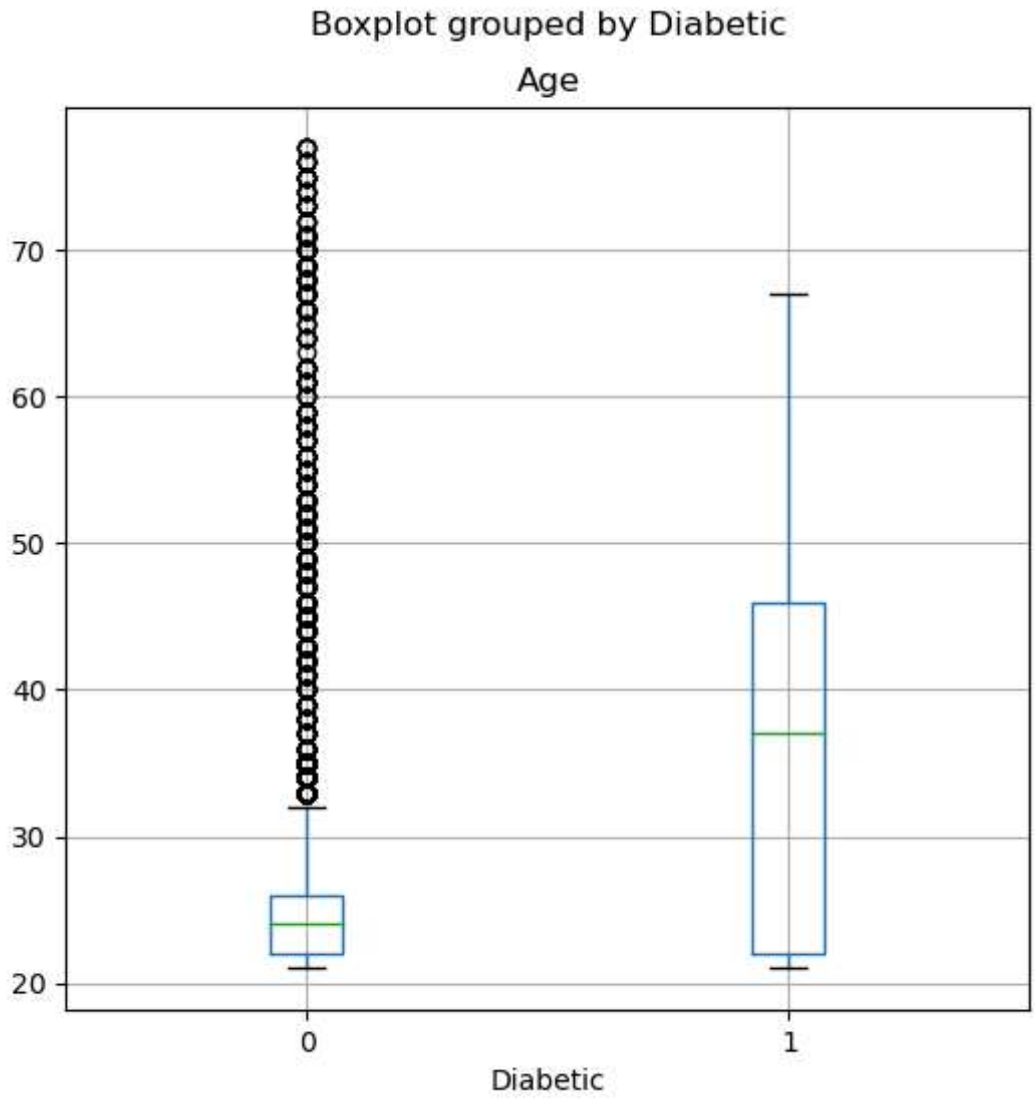

Boxplot grouped by Diabetic
DiastolicBloodPressure
120
100
80
60
40
20
0
1
Diabetic


Boxplot grouped by Diabetic
TricepsThickness
80
60
40
20
0
1
Diabetic


Boxplot grouped by Diabetic
SerumInsulin
800
700
600
500
400
300
200
100
0
0
1
Diabetic


Boxplot grouped by Diabetic
BMI
55
50
45
40
35
30
25
20
0
1
Diabetic


Boxplot grouped by Diabetic
DiabetesPedigree
2.0
1.5
1.0
0.5
0.0
0
1
Diabetic



For some of the features, there's a noticeable difference in the distribution for each label value. In particular, **Pregnancies** and **Age** show markedly different distributions for diabetic patients than for non-diabetic patients. These features may help predict whether or not a patient is diabetic.

Split the data

Our dataset includes known values for the label, so we can use this to train a classifier so that it finds a statistical relationship between the features and the label value - but how will we know if our model is any good? How do we know it will predict correctly when we use it with new data that it wasn't trained with? Well, we can take advantage of the fact we have a large dataset with known label values, use only some of it to train the model, and hold back some to test the trained model - enabling us to compare the predicted labels with the already known labels in the test set.

In Python, the **scikit-learn** package contains a large number of functions we can use to build a machine learning model - including a **train_test_split** function that ensures we get a statistically random split of training and test data. We'll use that to split the data into 70% for training and hold back 30% for testing.

In [4]:
```
from sklearn.model_selection import train_test_split

# Split data 70%-30% into training set and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30, ran

print ('Training cases: %d\nTest cases: %d' % (X_train.shape[0], X_test.shape
```

```
Training cases: 10500
Test cases: 4500
```

Train and Evaluate a Binary Classification Model

We're now ready to train our model by fitting the training features (**X_train**) to the training labels (**y_train**). There are various algorithms we can use to train the model. In this example, we'll use *Logistic Regression*, which (despite its name) is a well-established algorithm for classification. In addition to the training features and labels, we'll need to set a *regularization* parameter. This is used to counteract any bias in the sample, and help the model generalize well by avoiding *overfitting* the model to the training data.

> **Note**: Parameters for machine learning algorithms are generally referred to as *hyperparameters*. To a data scientist, *parameters* are values in the data itself - *hyperparameters* are defined externally from the data.

In [5]:
```
# Train the model
from sklearn.linear_model import LogisticRegression

# Set regularization rate
reg = 0.01

# train a logistic regression model on the training set
model = LogisticRegression(C=1/reg, solver="liblinear").fit(X_train, y_train)
print (model)
```

```
LogisticRegression(C=100.0, solver='liblinear')
```

Now that we've trained the model using the training data, we can use the test data we held back to evaluate how well it predicts. Again, **scikit-learn** can help us do this. Let's start by using the model to predict labels for our test set, and compare the predicted labels to the known labels:

In [6]:
```
predictions = model.predict(X_test)
print('Predicted labels: ', predictions)
print('Actual labels:    ', y_test)
```

```
Predicted labels:  [0 0 0 ... 0 1 0]
Actual labels:     [0 0 1 ... 1 1 1]
```

The arrays of labels are too long to be displayed in the notebook output, so we can only compare a few values. Even if we printed out all of the predicted and actual labels, there are too many of them to make this a sensible way to evaluate the model. Fortunately, **scikit-learn** has a few more tricks up its sleeve, and it provides some metrics that we can use to evaluate the model.

The first thing you might want to do is to check the *accuracy* of the predictions - that is, what proportion of the labels did the model predict correctly?

In [7]:
```
from sklearn.metrics import accuracy_score

print('Accuracy: ', accuracy_score(y_test, predictions))
```

```
Accuracy:  0.7891111111111111
```

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

print(classification_report(y_test, predictions))
```

```
              precision    recall  f1-score   support

           0       0.81      0.88      0.85      2986
           1       0.72      0.60      0.66      1514

    accuracy                           0.79      4500
   macro avg       0.77      0.74      0.75      4500
weighted avg       0.78      0.79      0.78      4500
```

The classification report includes the following metrics for each class (0 and 1):

Precision: Of the predictions the model made for this class, what proportion were correct?

Recall: Out of all of the instances of this class in the test dataset, how many did the model identify?

F1-Score: An average metric that takes both precision and recall into account.

Support: How many instances of this class are there in the test dataset?

The classification report also includes averages for these metrics, including a weighted average that allows for the imbalance in the number of cases of each class.

Because this is a binary classification problem, the 1 class is considered positive and its precision and recall are particularly interesting - these in effect answer the questions:

Of all the patients the model predicted are diabetic, how many are actually diabetic?

Of all the patients that are actually diabetic, how many did the model identify?

You can retrieve these values on their own by using the precision_score and recall_score metrics in Scikit-Learn (which by default assume a binary classification model).

In [9]:
```
from sklearn.metrics import precision_score, recall_score

print("Overall Precision:", precision_score(y_test, predictions))
print("Overall Recall:", recall_score(y_test, predictions))
```

```
Overall Precision: 0.723673792557403
Overall Recall: 0.6036988110964333
```

In [10]:
```
from sklearn.metrics import confusion_matrix

# Print the confusion matrix
cm = confusion_matrix(y_test, predictions)
print (cm)
```

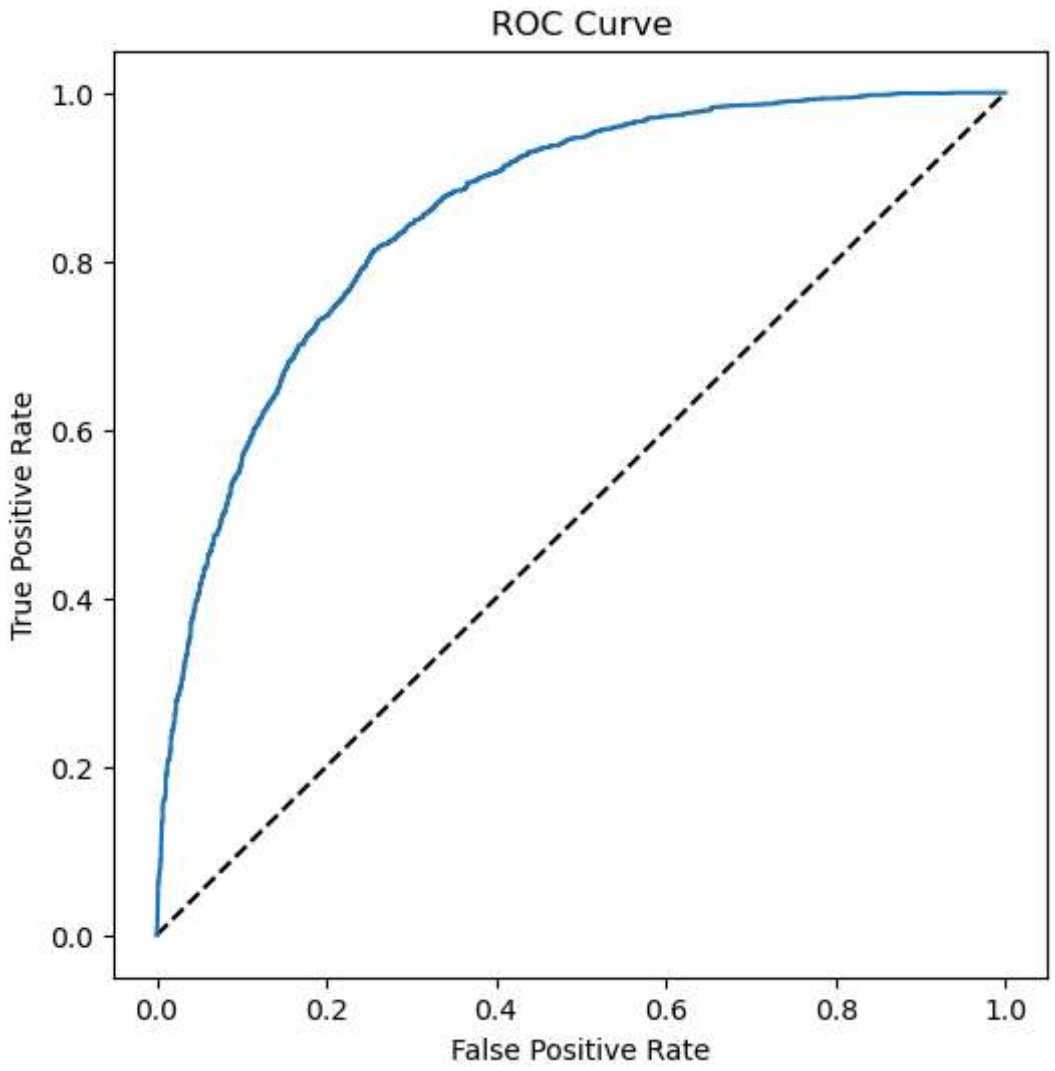
```
[[2637  349]
 [ 600  914]]
```

In [11]:
```
y_scores = model.predict_proba(X_test)
print(y_scores)
```

```
[[0.8157253  0.1842747 ]
 [0.96227714 0.03772286]
 [0.80784066 0.19215934]
 ...
 [0.61182181 0.38817819]
 [0.10960209 0.89039791]
 [0.64173872 0.35826128]]
```

In [12]:

```
from sklearn.metrics import roc_curve
from sklearn.metrics import confusion_matrix
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline

# calculate ROC curve
fpr, tpr, thresholds = roc_curve(y_test, y_scores[:,1])

# plot ROC curve
fig = plt.figure(figsize=(6, 6))
# Plot the diagonal 50% line
plt.plot([0, 1], [0, 1], 'k--')
# Plot the FPR and TPR achieved by our model
plt.plot(fpr, tpr)
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.show()
```



In [ ]: