cs188_project2-Blank

March 22, 2020

1 CS188 Project 2 - Binary Classification Comparative Methods

For this project we're going to attempt a binary classification of a dataset using multiple methods and compare results.

Our goals for this project will be to introduce you to several of the most common classification techniques, how to perform them and tweek parameters to optimize outcomes, how to produce and interpret results, and compare performance. You will be asked to analyze your findings and provide explanations for observed performance.

Specifically you will be asked to classify whether a patient is suffering from heart disease based on a host of potential medical factors.

DEFINITIONS

Binary Classification: In this case a complex dataset has an added 'target' label with one of two options. Your learning algorithm will try to assign one of these labels to the data.

Supervised Learning: This data is fully supervised, which means it's been fully labeled and we can trust the veracity of the labeling.

1.1 Background: The Dataset

For this exercise we will be using a subset of the UCI Heart Disease dataset, leveraging the fourteen most commonly used attributes. All identifying information about the patient has been scrubbed.

The dataset includes 14 columns. The information provided by each column is as follows:

age: Age in years

sex: (1 = male; 0 = female)

cp: Chest pain type (0 = asymptomatic; 1 = atypical angina; 2 = non-anginal pain; 3 = typical angina)

trestbps: Resting blood pressure (in mm Hg on admission to the hospital)

cholserum: Cholestoral in mg/dl

fbs Fasting blood sugar > 120 mg/dl (1 = true; 0 = false)

restecg: Resting electrocardiographic results (0= showing probable or definite left ventricular hypertrophy by Estes' criteria; 1 = normal; 2 = having ST-T wave abnormality (T wave inversions and/or ST elevation or depression of > 0.05 mV))

thalach: Maximum heart rate achieved

exang: Exercise induced angina (1 = yes; 0 = no)

oldpeakST: Depression induced by exercise relative to rest

slope: The slope of the peak exercise ST segment (0 = downsloping; 1 = flat; 2 = upsloping)

ca: Number of major vessels (0-3) colored by flourosopy

thal: 1 = normal; 2 = fixed defect; 7 = reversable defect

Sick: Indicates the presence of Heart disease (True = Disease; False = No disease)

1.2 Loading Essentials and Helper Functions

[1]:
```
#Here are a set of libraries we imported to complete this assignment.
#Feel free to use these or equivalent libraries for your implementation
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import matplotlib.pyplot as plt # this is used for the plot the graph
import os
import seaborn as sns # used for plot interactive graph.
from sklearn.model_selection import train_test_split, cross_val_score,␣
 ↪GridSearchCV
from sklearn import metrics
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.cluster import KMeans
from sklearn.metrics import confusion_matrix
import sklearn.metrics.cluster as smc
from sklearn.model_selection import KFold

# Added libraries for pre-processing the data
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, OneHotEncoder


from matplotlib import pyplot
import itertools

%matplotlib inline
import random

random.seed(42)
```

[2]:
```
# Helper function allowing you to export a graph
def save_fig(fig_id, tight_layout=True, fig_extension="png", resolution=300):
    path = os.path.join(fig_id + "." + fig_extension)
    print("Saving figure", fig_id)
    if tight_layout:
        plt.tight_layout()
    plt.savefig(path, format=fig_extension, dpi=resolution)
```

[3]:
```
# Helper function that allows you to draw nicely formatted confusion matrices
def draw_confusion_matrix(y, yhat, classes):
    '''
        Draws a confusion matrix for the given target and predictions
        Adapted from scikit-learn and discussion example.
    '''
    plt.cla()
    plt.clf()
    matrix = confusion_matrix(y, yhat)
    plt.imshow(matrix, interpolation='nearest', cmap=plt.cm.Blues)
    plt.title("Confusion Matrix")
    plt.colorbar()
    num_classes = len(classes)
    plt.xticks(np.arange(num_classes), classes, rotation=90)
    plt.yticks(np.arange(num_classes), classes)

    fmt = 'd'
    thresh = matrix.max() / 2.
    for i, j in itertools.product(range(matrix.shape[0]), range(matrix.
↪shape[1])):
        plt.text(j, i, format(matrix[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if matrix[i, j] > thresh else "black")

    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.tight_layout()
    plt.show()
```

1.3 [20 Points] Part 1. Load the Data and Analyze

Let's first load our dataset so we'll be able to work with it. (correct the relative path if your notebook is in a different directory than the csv file.)

[4]:
```
# Using Pandas' read_csv method to import data from the csv file
patient_data = pd.read_csv('heartdisease.csv')
```

1.3.1 Question 1.1 Now that our data is loaded, let's take a closer look at the dataset we're working with. Use the head method to display some of the rows so we can visualize the types of data fields we'll be working with, then use the describe method, along with any additional methods you'd like to call to better help you understand what you're working with and what issues you might face.

[5]:
```
# Displaying some of the rows so we can visualize the types of data fields
patient_data.head(5)
```

[5]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	sick
0	63	1	3	145	233	1	0	150	0	2.3	0	0	1	False
1	37	1	2	130	250	0	1	187	0	3.5	0	0	2	False
2	41	0	1	130	204	0	0	172	0	1.4	2	0	2	False
3	56	1	1	120	236	0	1	178	0	0.8	2	0	2	False
4	57	0	0	120	354	0	1	163	1	0.6	2	0	2	False

[6]:
```
# This tells us we have 303 datapoints, no nulls, and a majority of them are␣
 ↪represented as integers.
# Further, we can see that oldpeak is represented as a float, which 'sick' is␣
 ↪represented as a boolean.
patient_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 303 entries, 0 to 302
Data columns (total 14 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   age       303 non-null    int64
 1   sex       303 non-null    int64
 2   cp        303 non-null    int64
 3   trestbps  303 non-null    int64
 4   chol      303 non-null    int64
 5   fbs       303 non-null    int64
 6   restecg   303 non-null    int64
 7   thalach   303 non-null    int64
 8   exang     303 non-null    int64
 9   oldpeak   303 non-null    float64
 10  slope     303 non-null    int64
 11  ca        303 non-null    int64
 12  thal      303 non-null    int64
 13  sick      303 non-null    bool
```

```
dtypes: bool(1), float64(1), int64(12)
memory usage: 31.2 KB
```

[7]:
```
# Can use this to explore the mean and standard deviation, which helps provide␣
↪a better idea of the
# overall spread of values in the dataset for each feature.
patient_data.describe()
```

[7]:

	age	sex	cp	trestbps	chol	fbs	\
count	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	
mean	54.366337	0.683168	0.966997	131.623762	246.264026	0.148515	
std	9.082101	0.466011	1.032052	17.538143	51.830751	0.356198	
min	29.000000	0.000000	0.000000	94.000000	126.000000	0.000000	
25%	47.500000	0.000000	0.000000	120.000000	211.000000	0.000000	
50%	55.000000	1.000000	1.000000	130.000000	240.000000	0.000000	
75%	61.000000	1.000000	2.000000	140.000000	274.500000	0.000000	
max	77.000000	1.000000	3.000000	200.000000	564.000000	1.000000	

	restecg	thalach	exang	oldpeak	slope	ca	\
count	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	
mean	0.528053	149.646865	0.326733	1.039604	1.399340	0.729373	
std	0.525860	22.905161	0.469794	1.161075	0.616226	1.022606	
min	0.000000	71.000000	0.000000	0.000000	0.000000	0.000000	
25%	0.000000	133.500000	0.000000	0.000000	1.000000	0.000000	
50%	1.000000	153.000000	0.000000	0.800000	1.000000	0.000000	
75%	1.000000	166.000000	1.000000	1.600000	2.000000	1.000000	
max	2.000000	202.000000	1.000000	6.200000	2.000000	4.000000	

	thal
count	303.000000
mean	2.313531
std	0.612277
min	0.000000
25%	2.000000
50%	2.000000
75%	3.000000
max	3.000000

[8]:
```
# check for NaN values in dataset
patient_data.isna().sum()
```

[8]:
```
age         0
sex         0
cp          0
trestbps    0
chol        0
fbs         0
```

```
restecg     0
thalach     0
exang       0
oldpeak     0
slope       0
ca          0
thal        0
sick        0
dtype: int64
```

[9]:
```
# Check to see if there are any missing values in the dataset
patient_data.isnull().sum()
```

[9]:
```
age         0
sex         0
cp          0
trestbps    0
chol        0
fbs         0
restecg     0
thalach     0
exang       0
oldpeak     0
slope       0
ca          0
thal        0
sick        0
dtype: int64
```

1.3.2 Question 1.2 Discuss your data preprocessing strategy. Are their any datafield types that are problemmatic and why? Will there be any null values you will have to impute and how do you intend to do so? Finally, for your numeric and categorical features, what if any, additional preprocessing steps will you take on those data elements?

From the feature descriptions provided in the background and from 1.1 we can infer the following about our data

Features - Type of Variable age: Numeric

sex: Categorical

cp: Categorical

trestbps: Numeric

cholserum: Numeric

fbs: Categorical

restecg: Categorical

thalach: Numeric

exang: Categorical

oldpeakST: Numeric

slope: Categorical

ca: Categorical

thal: Categorical

sick (target): Categorical

Here are some data preprocessing strategies that will be used:

1. **Shuffle the dataset** - The dataset is organized such that all the 0s come before all the 1s. This could cause the model to learn trivial relationships since inputs are seen sequentially and weights are adjusted accordingly. This is usually considered good practice in general, but it especially applies to this case.
2. **sick**: This is a bit problematic. Given this is our label, it needs to be transformed from boolean (which tends to be binary in nature), to a binary 1 and 0, where 1 represents true and 0 represents false. Our models are not capable of predicting 'False' and 'True' as the boolean words themselves.
3. Since we have **no NaN values or null values**, we require no data imputation methods.
4. For our **numeric variables** (highlighted above), we want to scale the values for each feature such that all of the values from that feature have the mean value subtracted and have unit variance, allowing values to fall in the range of [-1,1] inclusive. This helps account for the variance in numeric features. Models don't just train better when this is done, but it also helps prevent given too much weightage to features that could propogate through the network.
5. For the **categorical variables**, it becomes important to one hot encode them so that we create fixed length sparse vectors for each of the variables (equal to the number of unique categories for that feature), where each column in the vector represents one unique value.

Most of these preprocessing steps can be done using sklearn mixins and pipelining.

While we may be tempted to mess around with `oldpeakST` which has float values rather than whole numbers, this should be left as is and **not** be converted to integers. We will apply standard scaling techniques to this as well to ensure that the entire range can be linearly transformed into a more suitable, manageable and model preferred range of [-1,1]

1.3.3 Question 1.3 Before we begin our analysis we need to fix the field(s) that will be problematic. Specifically convert our boolean sick variable into a binary numeric target variable (values of either '0' or '1'), and then drop the original sick datafield from the dataframe.

[10]:
```
patient_data['sick'].head()
```

[10]:
```
0    False
1    False
```

```
2    False
3    False
4    False
Name: sick, dtype: bool
```

[11]:
```
# Get distribution of true and false to ensure this does not change
patient_data['sick'].value_counts()
```

[11]:
```
False    165
True     138
Name: sick, dtype: int64
```

[12]:
```
# Using list comprehesion to do this in place instead of creating a new feature␣
 ↪and dropping the
# original target variable.
patient_data['sick'] = [1 if row else 0 for row in patient_data['sick']]
```

[13]:
```
patient_data['sick'].head()
```

[13]:
```
0    0
1    0
2    0
3    0
4    0
Name: sick, dtype: int64
```

[14]:
```
# Data Distribution stays the same.
patient_data['sick'].value_counts()
```

[14]:
```
0    165
1    138
Name: sick, dtype: int64
```

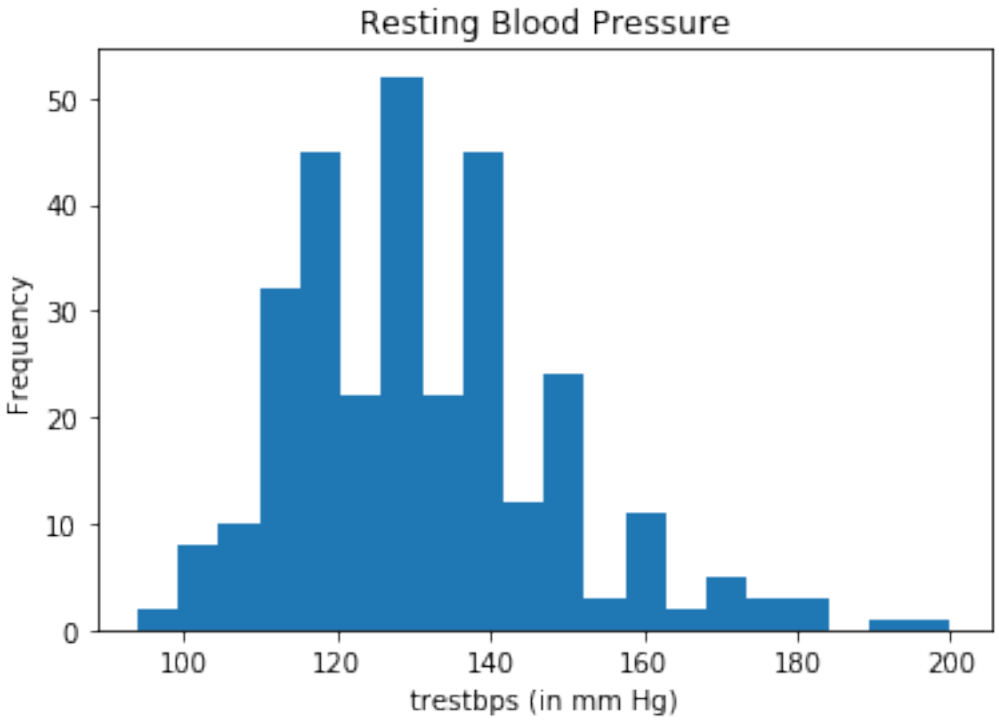
1.3.4 Question 1.4 Now that we have a feel for the data-types for each of the variables, plot histograms of each field and attempt to ascertain how each variable performs (is it a binary, or limited selection, or does it follow a gradient? (Note: No need to describe each variable, but pick out a few you wish to highlight)

[15]:
```
patient_data.columns
```

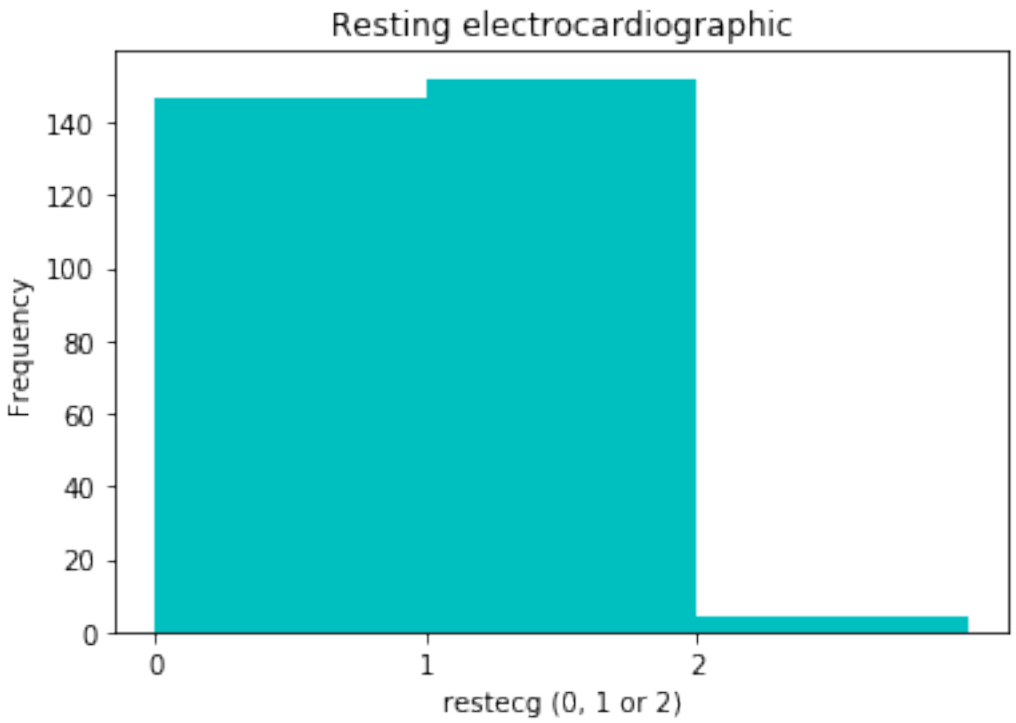
[15]:
```
Index(['age', 'sex', 'cp', 'trestbps', 'chol', 'fbs', 'restecg', 'thalach',
       'exang', 'oldpeak', 'slope', 'ca', 'thal', 'sick'],
      dtype='object')
```

[16]:
```
# trestbps is a numeric variable that is continuous in nature, and so it␣
 ↪follows a gradient.
fig, ax = plt.subplots()
```

```
ax.hist(patient_data['trestbps'], bins=20)
ax.set_ylabel('Frequency')
ax.set_xlabel('trestbps (in mm Hg)')
ax.set_title('Resting Blood Pressure')
plt.show()
```



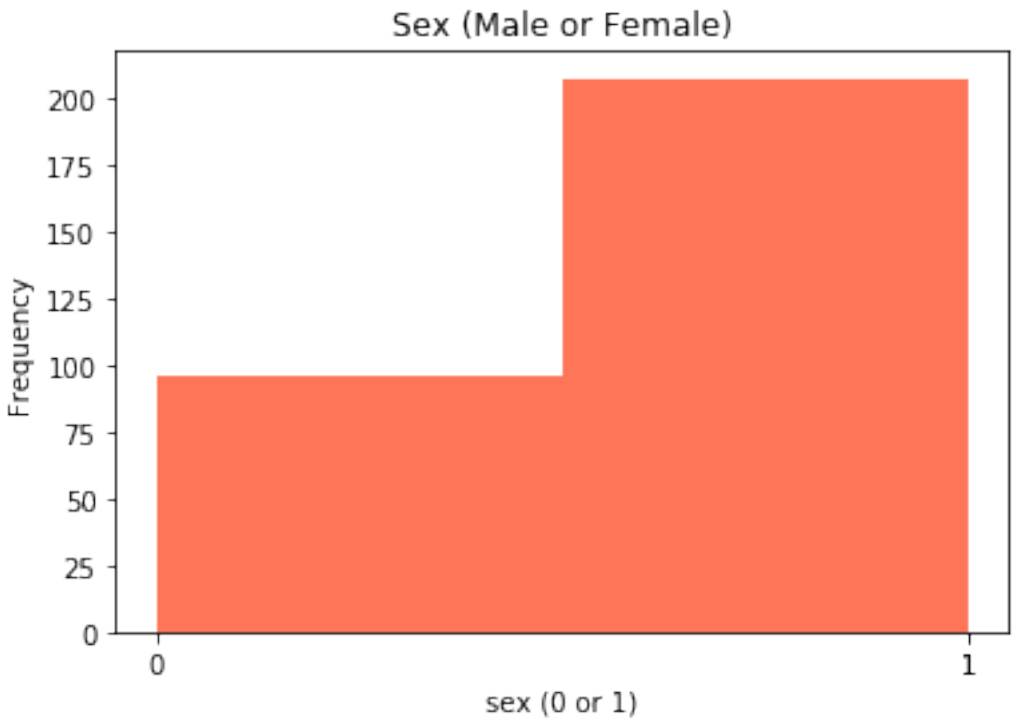
[17]:
```
# restecg is a categorical variable that appears to have more than 2␣
 ↪categories, so the
# variable is a limited selection.
fig, ax = plt.subplots()
ax.hist(patient_data['restecg'], color='c', bins=[0,1,2,3])
ax.set_xlabel('restecg (0, 1 or 2)')
ax.set_ylabel('Frequency')
ax.set_title('Resting electrocardiographic')
plt.xticks((0,2,1))
plt.show()
```



[18]:
```
patient_data['restecg'].value_counts()
```

[18]:
```
1    152
0    147
2      4
Name: restecg, dtype: int64
```

[19]:
```
# sex is a categorical variable that appears to have exactly 2 categories, so␣
↪the variable is binary in nature
fig, ax = plt.subplots()
ax.hist(patient_data['sex'], color='#FF7557', bins=[0,0.5,1])
ax.set_xlabel('sex (0 or 1)')
ax.set_ylabel('Frequency')
ax.set_title('Sex (Male or Female)')
plt.xticks((0,1,1))
plt.show()
```



1.3.5 Question 1.5 We also want to make sure we are dealing with a balanced dataset. In this case, we want to confirm whether or not we have an equitable number of sick and healthy individuals to ensure that our classifier will have a sufficiently balanced dataset to adequately classify the two. Plot a histogram specifically of the sick target, and conduct a count of the number of sick and healthy individuals and report on the results:

[20]:
```
patient_counts = patient_data['sick'].value_counts()
patient_counts
```

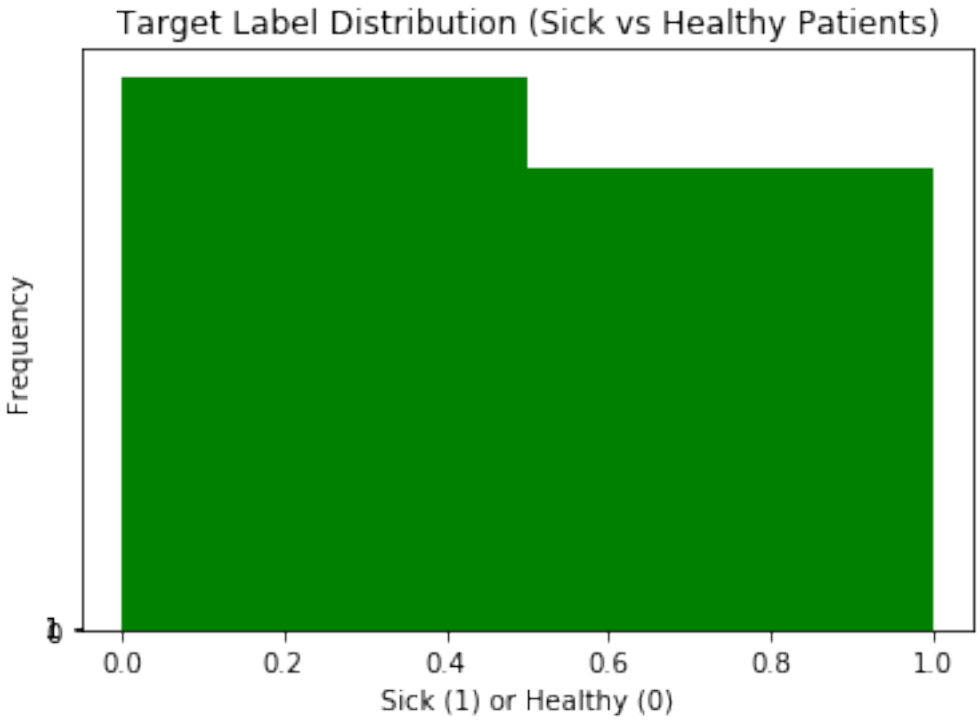
[20]:
```
0    165
1    138
Name: sick, dtype: int64
```

[21]:
```
# Using the values from value counts, we see that:
patient_counts = patient_counts.tolist()
total_patients = sum(patient_counts)
print("Percentage of sick patients: ", round((patient_counts[1]/
 ↪total_patients)*100, 2))
print("Percentage of healthy patients: ", round((patient_counts[0]/
 ↪total_patients)*100, 2))
```

```
Percentage of sick patients:  45.54
```

```
Percentage of healthy patients:  54.46
```

[22]:
```
fig, ax = plt.subplots()
ax.hist(patient_data['sick'], color='green', bins=[0,0.5,1])
ax.set_xlabel('Sick (1) or Healthy (0)')
ax.set_ylabel('Frequency')
ax.set_title('Target Label Distribution (Sick vs Healthy Patients)')
plt.yticks((0,1,1))
plt.show()
```



From the value counts (and the percentages determined from these values), as well as the histogram (rather, a bar plot) of the distribution of sick and healthy patients using the `sick` feature in the `patient_data` dataset, we can see that the dataset has a greater percentage of of healthy patients (54.5%) than it does sick patients (45.5%), which means it is slightly biased towards the healthy patient label. While this is not problematic as such because the distribution is very similar, it could affect models that perform better with equal splits of target variable.

It could be that the model tends to learn and adjust its weights to optimize for predicting healthy patients more accurately than sick patients, and we might be able to observe this if we create a correlation matrix and observe the `precision` and `recall` scores. This may also cause the model to have a slightly weakened accuracy on the predicition task, especially since our **goal** is to predict the `sick` patient more accurately, that is, to be able to ensure that we are able to get a high accuracy on this class since the consequences of a false negative prediction on this class can have very severe consequences.

The partial imbalance can also cause overfitting, which is a cause for concern since we want the model we train and select to be one that can generalize best across train, validation and test.

1.3.6 Question 1.6 Balanced datasets are important to ensure that classifiers train adequately and don't overfit, however arbitrary balancing of a dataset might introduce its own issues. Discuss some of the problems that might arise by artificially balancing a dataset.

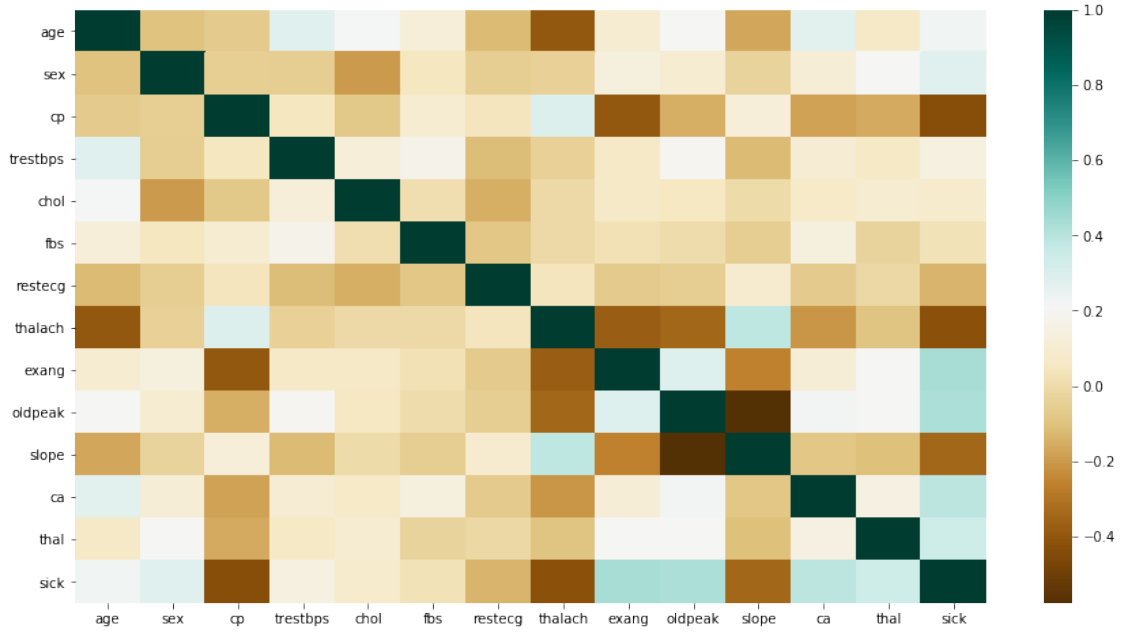
There are potentially many other pitfalls, however, they often might be more task specific. That being said, here are a few that may be generally applicable:

1. **Losing data**: You lose more of the majority class (assuming a binary classification task), which may cause a significant loss in vital information we didn't know about, i.e., the data points belonging to the majority class could hold information that would help the model train better. Therefore, the sample you pick may not represent the population well.
2. If you try and artificially/manually perform data reduction by obseration (especially for smaller datasets since this is actually humanly possible), you can introduce human bias which will propogate into the network. This becomes especially bad for this task.
3. Can potentially eliminate outliers if you use clustering-based dataset balancing methods (remove the outliers around each centroid), which will cause the model to overfit because the outliers often force the model to generalize by skewing the loss value on that iteration.
4. Reducing data can lead to lower accuracy scores, and also has the potential for overfitting.

1.3.7 Question 1.7 Now that we have our dataframe prepared let's start analyzing our data. For this next question let's look at the correlations of our variables to our target value. First, map out the correlations between the values, and then discuss the relationships you observe. Do some research on the variables to understand why they may relate to the observed corellations. Intuitively, why do you think some variables correlate more highly than others (hint: one possible approach you can use the sns heatmap function to map the corr() method)?

[23]:
```
corr = patient_data.corr()
fig, ax = plt.subplots(figsize=(15,8))
sns.heatmap(corr, ax=ax, cmap='BrBG')
```

[23]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1297bb450>
```



[24]:
```
corr['sick'].sort_values(ascending=False)[1:]
```

[24]:
```
exang       0.436757
oldpeak     0.430696
ca          0.391724
thal        0.344029
sex         0.280937
age         0.225439
trestbps    0.144931
chol        0.085239
fbs         0.028046
restecg    -0.137230
slope      -0.345877
thalach    -0.421741
cp         -0.433798
Name: sick, dtype: float64
```

From a very basic standpoint, using both the heatmap of correlation scores (and specifically focusing on `sick`) and the correlation values, we can see that the following features are **positively correlated**

exang 0.436757 (most positively correlated)

oldpeak 0.430696

ca 0.391724

thal 0.344029

sex 0.280937

age 0.225439

trestbps 0.144931

chol 0.085239

fbs 0.028046 (least positively correlated)

and the following features are **negatively correlated**

restecg -0.137230 (least negatively correlated)

slope -0.345877

thalach -0.421741

cp -0.433798 (most negatively correlated)

In particular, it makes sense to investigate the **most and least positively and negatively correlated features**.

1. **exang (most positively correlated)**: Exang is short for exercise induced angina. This is a binary categorical feature with 1 for yes and 0 for no. It intuitively makes sense that this has a strong positive correlation with the target label because seeing that someone has angina (insufficient supply of blood to the heart) clearly indicates that not enough blood is flowing in the blood vessels/arteries, indicating that they might indeed be clogged/blocked/narrow and the sign of a developing heart disease. This is also an easy label to measure since Angina is just marked by severe pain in the chest, which can easily be confirmed with a patient or by seeing their breathlessness. Since it is positively correlated, the higher/greater the amount of exercise induced angina, the greater the chance of a heart disease.
2. **fbs (least positively correlated)**: fbs stands for Fasting blood sugar, and is a binary categorical feature with > 120 mg/dl marked as 1 (true), and less than marked as 0 (false). A fbs > 120 is indicative of diabetes, and it intuitively makes sense that there is a partial correlation between diabetes and heart disease, since diabetic people tend to be overweight because they can't process all the glucose entering their body, which eventually gets stored as fat. There is an obvious relationship between heart disease and being overweight. However, I do find it surprising that this is the least positively correlated feature. Since it is positively correlated, the higher/greater the amount of blood sugar/diabetes, the greater the chance of a heart disease.
3. **restecg (least negatively correlated)**: This stands for Resting electrocardiographic. A label of 0 or 2 indicate some form of abnormality with the heart, while a value of 1 indicates a normal ECG. Negative correlation indicates that as one variable moves in one direction, the other moves in the opposite direction. This becomes harder to visualize when the target and feature are both categorical variables (I'm personally not sure how correlation is calculated between two categorical variables, since we cannot computer covariance). That being said, I believe that it is being measured using some sort of distance metric. However, since one is binary and the other has more than 2 classes, I believe that the interpretation here might be that the `0` class in the feature (showing probable or definite left ventricular hypertrophy by Estes' criteria) might be correlated to a `1` in the target variable, hence showing a `negative` correlation, and this intuitively makes quite a bit of sense.

4. **cp (most negatively correlated)**: stands for chest pain type. This is the most negatively correlated feature, meaning that as one variable (either `sick` or `cp` moves in one direction), the other moves in the opposite direction or vice-versa. It's interesting that this is the most negatively correlated feature, but from my understanding, this means that a cp of 0, 1 and possibly 2 result in a target class of 1, while a cp of 3 results in a target class of 0. [[If we assume that corr uses numeric data, and does not understand that it is a categorical variable]]. This could intuitively then make sense, because 1 and 2 stand for atypical-pain and non-angina based pain, which on a more abstract level, indicates that any form of persistent chest pain could be strongly correlated with some sort of coronary heart disease. I find it odd that the label 3 (angina pain) would then be negatively correlated. However, some investigation helped me figure out why this is reflected incorrectly. The value counts are highly skewed towards labeles 0, 1 and 2, with $< 10\%$ falling in category 3 (angina pain) [see below], while the labels still indicate 46% of the data samples to have heart disease. The disparity results in a strong negative correlation along with the fact that we're somehow finding correlation between two categorical variables.

[25]:
```
patient_data['cp'].value_counts()
```

[25]:
```
0    143
2     87
1     50
3     23
Name: cp, dtype: int64
```

1.4 [30 Points] Part 2. Prepare the Data

Before running our various learning methods, we need to do some additional prep to finalize our data. Specifically you'll have to cut the classification target from the data that will be used to classify, and then you'll have to divide the dataset into training and testing cohorts.

Specifically, we're going to ask you to prepare 2 batches of data: 1. Will simply be the raw numeric data that hasn't gone through any additional pre-processing. The other, will be data that you pipeline using your own selected methods. We will then feed both of these datasets into a classifier to showcase just how important this step can be!

1.4.1 Question 2.1 Save the target column as a separate array and then drop it from the dataframe.

[26]:
```
# Shuffling the dataset (part of the pre-processing step)
patient_data = patient_data.sample(frac=1, random_state=42)
# Copy labels
prediction_labels = patient_data['sick'].copy()
# Drop column inplace
patient_data.drop('sick', axis=1, inplace=True)
```

[27]:
```
patient_data.columns
```

[27]:
```
Index(['age', 'sex', 'cp', 'trestbps', 'chol', 'fbs', 'restecg', 'thalach',
       'exang', 'oldpeak', 'slope', 'ca', 'thal'],
      dtype='object')
```

1.4.2 Question 2.2 First Create your 'Raw' unprocessed training data by dividing your dataframe into training and testing cohorts, with your training cohort consisting of 70% of your total dataframe (hint: use the train_test_split method) Output the resulting shapes of your training and testing samples to confirm that your split was successful.

[28]:
```
X_train_raw, X_test_raw, y_train_raw, y_test_raw =␣
↪train_test_split(patient_data, prediction_labels, test_size=0.3,␣
↪random_state=42)
```

[29]:
```
X_train_raw.shape
```

[29]:
```
(212, 13)
```

[30]:
```
X_test_raw.shape
```

[30]:
```
(91, 13)
```

[31]:
```
y_train_raw.shape
```

[31]:
```
(212,)
```

[32]:
```
y_test_raw.shape
```

[32]:
```
(91,)
```

[33]:
```
print("Train percentage (approx): ", round(X_train_raw.shape[0]/patient_data.
↪shape[0] * 100,2))
```

```
Train percentage (approx):  69.97
```

1.4.3 Question 2.3 Now create a pipeline to conduct any additional preparation of the data you would like. Output the resulting array to ensure it was processed correctly.

Features - Type of Variable age: Numeric

sex: Categorical

cp: Categorical

trestbps: Numeric

cholserum: Numeric

fbs: Categorical

restecg: Categorical

thalach: Numeric

exang: Categorical

oldpeakST: Numeric

slope: Categorical

ca: Categorical

thal: Categorical

sick (target): Categorical

[34]:
```
categorical_features = ['sex','cp','fbs','restecg','exang','slope','ca','thal']
patient_data_no_categorical = patient_data.drop(categorical_features, axis=1)

num_pipeline = Pipeline([
        ('std_scaler', StandardScaler(with_std=True)),
])

numerical_features = list(patient_data_no_categorical)

full_pipeline = ColumnTransformer([
        ("num", num_pipeline, numerical_features),
        ("cat", OneHotEncoder(), categorical_features),
])

patient_data_prepared = full_pipeline.fit_transform(patient_data)
```

[35]:
```
num_unique_categories = 0
for category in patient_data[categorical_features]:
    num_unique_categories += len(patient_data[category].value_counts().tolist())
print("Number of numerical features: ", len(numerical_features))
print("Number of one-hot encoded categorical features: ", num_unique_categories)
print("Total number of input features (should equal patient_data_prepared.
 ↪shape[1]): ",
      len(numerical_features) + num_unique_categories)
patient_data_prepared.shape
```

```
Number of numerical features:  5
Number of one-hot encoded categorical features:  25
Total number of input features (should equal patient_data_prepared.shape[1]):
30
```

[35]:
```
(303, 30)
```

[36]:
```
patient_data_prepared[0]
```

[36]:
```
array([ 0.29046364,  1.04952029,  0.57466203, -1.6463164 , -0.37924438,
        0.        ,  1.        ,  1.        ,  0.        ,  0.        ,
        0.        ,  1.        ,  0.        ,  1.        ,  0.        ,
        0.        ,  0.        ,  1.        ,  0.        ,  1.        ,
        0.        ,  0.        ,  1.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  1.        ,  0.        ,  0.        ])
```

1.4.4 Question 2.4 Now create a separate, processed training data set by dividing your processed dataframe into training and testing cohorts, using the same settings as Q2.2 (REMEMBER TO USE DIFFERENT TRAINING AND TESTING VARIABLES SO AS NOT TO OVERWRITE YOUR PREVIOUS DATA). Output the resulting shapes of your training and testing samples to confirm that your split was successful, and describe what differences there are between your two training datasets.

[37]:
```
X_train_prepared, X_test_prepared, \
    y_train_prepared, y_test_prepared = train_test_split(patient_data_prepared,␣
↪prediction_labels,
                                                         test_size=0.3,␣
↪random_state=42)
```

[38]:
```
X_train_prepared.shape
```

[38]:
```
(212, 30)
```

[39]:
```
X_test_prepared.shape
```

[39]:
```
(91, 30)
```

[40]:
```
y_train_prepared.shape
```

[40]:
```
(212,)
```

[41]:
```
y_test_prepared.shape
```

[41]:
```
(91,)
```

The main differences between the raw and prepared dataset are as follows: 1. Each of the categorical variables have been transformed into one-hot encoded vectors in the prepared dataset, while they are still labelled as numbers in the raw data. This gives equal weightage to each unique category for each categorical variable, rather than biasing towards more importance for "higher-valued" categories (eg. $2 > 1$ although they are actually equally likely and equally important). 2. All numeric features have been scaled to unit variance in the prepared dataset, while they are still the raw values in the raw dataset and fall over a much larger distribution/range. 3. The number of input features have increased because of the one-hot encoded vectors, while they are the same in the raw dataset. 4. The row splits (train data, test data) are EXACTLY the same because we use the same order, same random state, and same test_size. This will allow for a fair comparison.

1.5 [50 Points] Part 3. Learning Methods

We're finally ready to actually begin classifying our data. To do so we'll employ multiple learning methods and compare result.

1.5.1 Linear Decision Boundary Methods

1.5.2 SVM (Support Vector Machine)

A Support Vector Machine (SVM) is a discriminative classifier formally defined by a separating hyperplane. In other words, given labeled training data (supervised learning), the algorithm outputs an optimal hyperplane which categorizes new examples. In two dimentional space this hyperplane is a line dividing a plane in two parts where in each class lay in either side.

1.5.3 Question 3.1.1 Implement a Support Vector Machine classifier on your RAW dataset. Review the SVM Documentation for how to implement a model. For this implementation you can simply use the default settings, but set probability = True.

[42]:
```
svm = SVC(probability=True)
svm.fit(X_train_raw, y_train_raw)
```

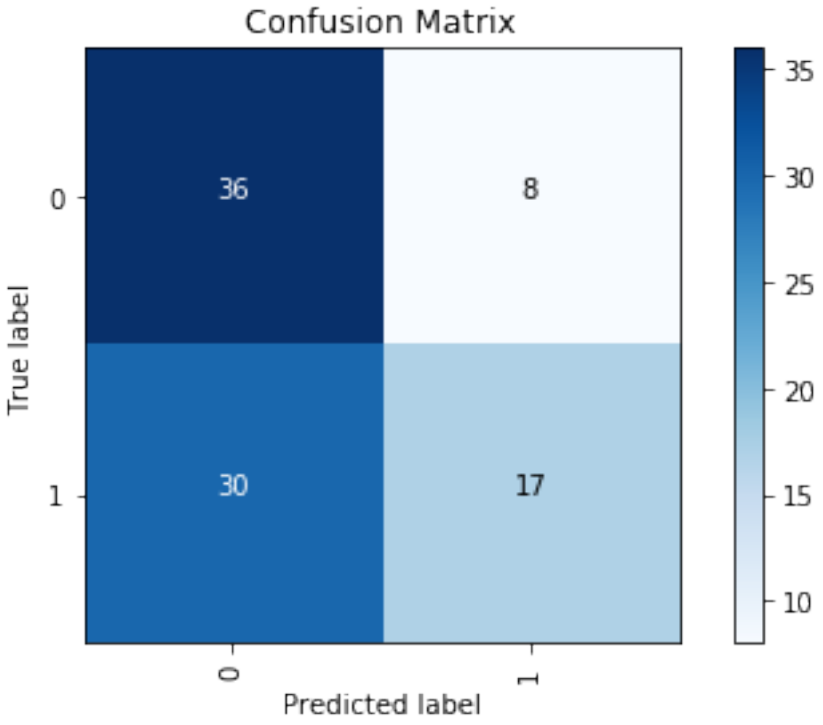
[42]:
```
SVC(C=1.0, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='scale', kernel='rbf',
    max_iter=-1, probability=True, random_state=None, shrinking=True, tol=0.001,
    verbose=False)
```

1.5.4 Question 3.1.2 Report the accuracy, precision, recall, F1 Score, and confusion matrix of the resulting model.

[43]:
```
predictions = svm.predict(X_test_raw)

accuracy = metrics.accuracy_score(y_test_raw, predictions)
precision = metrics.precision_score(y_test_raw, predictions)
recall = metrics.recall_score(y_test_raw, predictions)
f1_score = metrics.f1_score(y_test_raw, predictions)

print("Accuracy: ", round(accuracy,2))
print("Precision: ", round(precision, 2))
print("Recall: ", round(recall, 2))
print("F1 Score: ", round(f1_score,2))

# Confusion matrix
conf_matrix = confusion_matrix(y_test_raw, predictions)
print("Confusion Matrix: \n", conf_matrix)
draw_confusion_matrix(y_test_raw, predictions, [0,1])
```

```
Accuracy:  0.58
Precision:  0.68
```

```
Recall:  0.36
F1 Score:  0.47
Confusion Matrix:
 [[36  8]
 [30 17]]
```



1.5.5 Question 3.1.3 Discuss what each measure is reporting, why they are different, and why each of these measures is significant. Explore why we might choose to evaluate the performance of differing models differently based on these factors. Try to give some specific examples of scenarios in which you might value one of these measures over the others.

- **Accuracy**: Accuracy reflects the number of labels predicted correctly, that is, it is the ratio of the number of ground truth labels predicted correctly by our model / the total number of ground truth labels. This measure helps us get a better idea of the general prediction power of our model, as well as high level overview of how well the model has generalized its predicition capabilities (specifically for classification tasks). Accuracy works best if false positives and false negatives have similar cost. Accuracy of a model is generally pretty good, but is a terrible measurement in cases where the data is highly class imbalanced (like advertisements and cancer predicition), because a 99% accuracy might just reflect the accuracy of predicition the class with more data, but we care about the 1% class specific data and the model's accuracy on that data. Mathematically, this is (TP+TN)/(TP+FP+TN+FN)
- **Precision**: Precision measures the true positive rate, that is, by using the confusion matrix, we're able to calculate precision as TP/(TP+FP). Intuitively, this indicates that amongst all

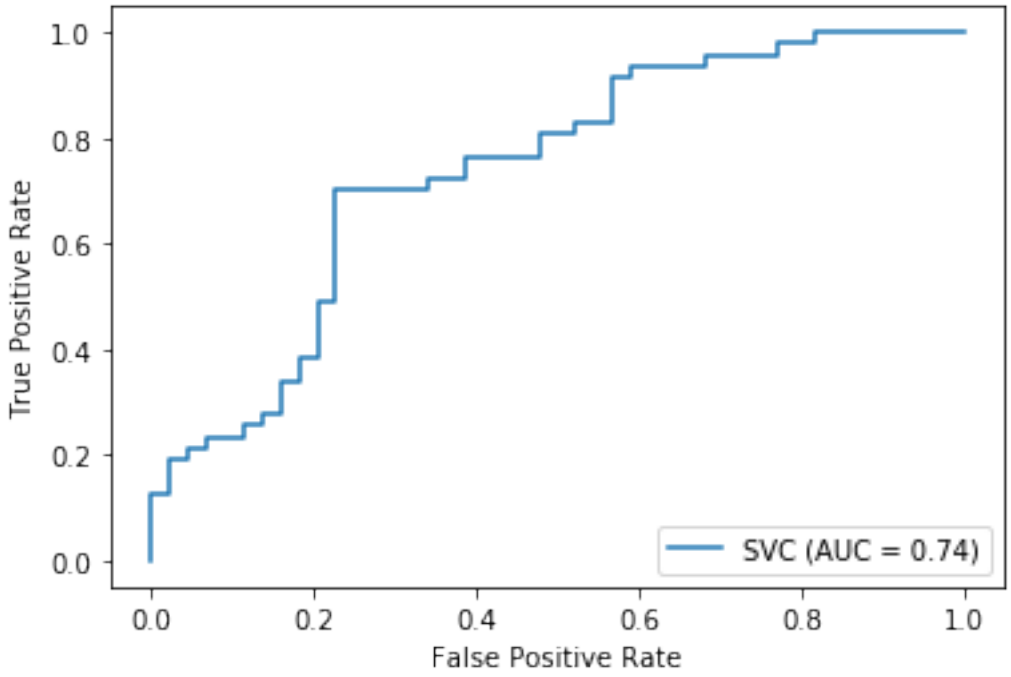
the true labels in the predictions, how many of the ground truths did we actually get correct, i.e., what proportion of true labels were actually correct? Precision is a great metric to use for things like YouTube recommendation predictions, because the consequence of a false positive is minimal, but it would still help to see how many true positives we got from all of the cases where the model predicts true.

- **Recall**: Recall is also calculated using the confusion matrix, and is calculated as (TP)/(TP+FN). In theory, this represents how many of the ground truths we actually got correct, given all the true ground labels in the dataset. Again, this is extremely important for project 2, because we want to maximize the number of positives we're actually able to predict. The problem is that precision and recall have a trade-off. As you maximize one, the other one decreases and vice-versa. Therefore, if we can ideally get a model with high precision and high recall, then we know the model is fairly robust in its ability to predict the correct positive labels. This is really important in this dataset for project 2, because we want to be able to maximize the number of true positives we're able to predict (can either increase TP or decrease FN). It is important for tasks where the consequence of an incorrect prediction is extremely costly.
- **F1-Score**: F1 Score is the weighted average of Precision and Recall ($2(Recall \quad Precision)$ / (Recall + Precision)). Therefore, this score takes both false positives and false negatives into account. F1-score is used when the False Negatives and False Positives are crucial. It tends to be a better representation of the misclassified data in comparison to accuracy. Great for class imbalanced datasets. In most real-life classification problems, imbalanced class distribution exists and thus F1-score is a better metric to evaluate our model on. Great metric for advertising industry with click-through rate, rare disease detection, etc.

1.5.6 Question 3.1.4 Plot a Receiver Operating Characteristic curve, or ROC curve, and describe what it is and what the results indicate

[44]:
```
metrics.plot_roc_curve(svm, X_test_raw, y_test_raw)
```

[44]:
```
<sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x1297a5c10>
```



ROC curves are a way of measuring the quality of predictions made by a classifier at **various thresholds settings**. ROC is a probability curve and AUC represents degree or measure of separability. It tells us how much the model is capable of distinguishing between classes. ROC curves typically feature TPR on the Y axis, and FPR - (FP/TN+FP) on the X axis. Since it is a probability curve, it essentially plots the distribution of TP,FP,TN,FN in the form of the TPR and FPR at various possible threshold values that split the prediction score (between 0.0 and 1.0) into True (1) or False (0) in a binary classification task. If TN and TP are completely separable (the model can distinguish this perfectly), then the probability distribution will be independent of each other, i.e., it has an "ideal" measure of separability. When two distributions overlap, we introduce type 1 and type 2 error. Depending upon the threshold, we can minimize or maximize them. When we decrease the threshold, we get more positive values and when we increase the threshold, we get more negative values. When we increase TPR, FPR also increases and vice versa. Overall it represents a tradeoff between "specificity" (FPR) and "sensitivity" (TPR). ROC curve: Optimizes to balance specificity and sensitivity. Lowering the classification threshold classifies more items as positive, increasing both False Positives and True Positives. [https://www.youtube.com/watch?v=egTNM8NSa2k]

The results indicate that the Area under the curve (AUC) is approximately 0.74, which means that at the optimization points of specificity and sensitivity, the distribution of TP and TN are somewhat separable. Although the model is not great, it is able to perform better than a random guess on the positive class and has a sufficiently high TPR at its peak (around 0.75).

1.5.7 Question 3.1.5 Rerun, using the exact same settings, only this time use your processed data as inputs.

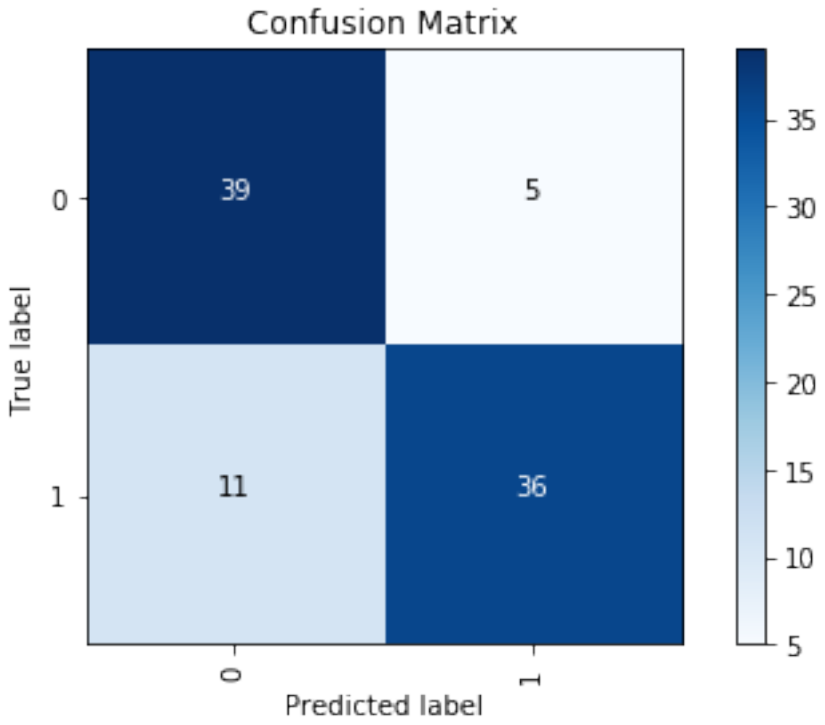
[45]:
```
svm = SVC(probability=True)
svm.fit(X_train_prepared, y_train_prepared)
```

[45]:
```
SVC(C=1.0, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='scale', kernel='rbf',
    max_iter=-1, probability=True, random_state=None, shrinking=True, tol=0.001,
    verbose=False)
```

1.5.8 Question 3.1.6 Report the accuracy, precision, recall, F1 Score, confusion matrix, and plot the ROC Curve of the resulting model.

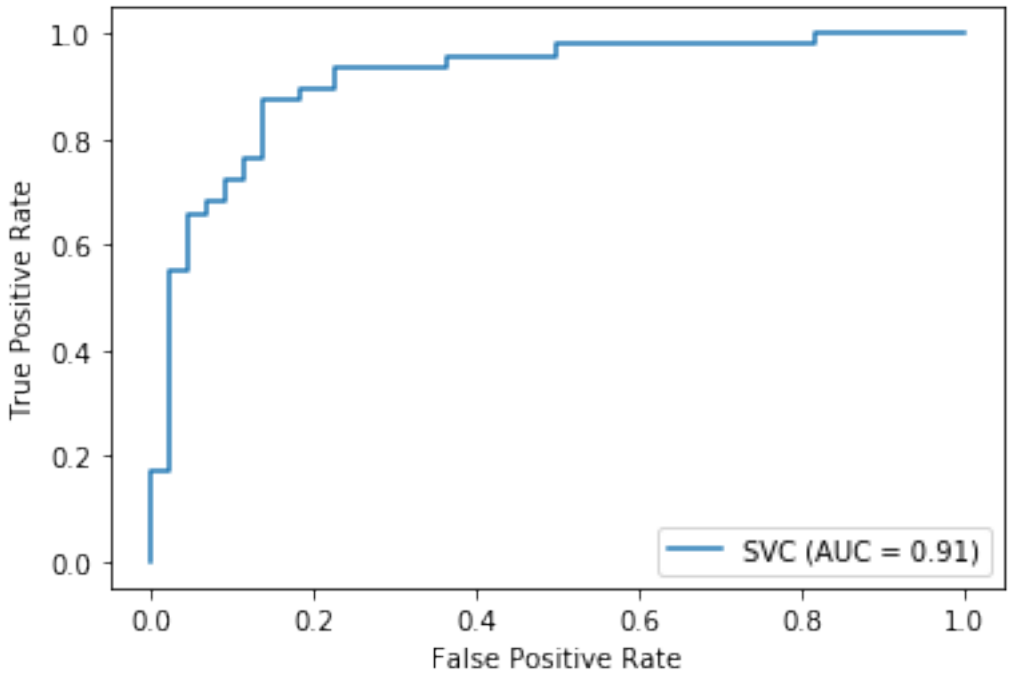
[46]:
```
predictions = svm.predict(X_test_prepared)

accuracy = metrics.accuracy_score(y_test_prepared, predictions)
precision = metrics.precision_score(y_test_prepared, predictions)
recall = metrics.recall_score(y_test_prepared, predictions)
f1_score = metrics.f1_score(y_test_prepared, predictions)

print("Accuracy: ", round(accuracy,2))
print("Precision: ", round(precision, 2))
print("Recall: ", round(recall, 2))
print("F1 Score: ", round(f1_score,2))

# Confusion matrix
conf_matrix = confusion_matrix(y_test_prepared, predictions)
print("Confusion Matrix: \n", conf_matrix)
draw_confusion_matrix(y_test_prepared, predictions, [0,1])
```

```
Accuracy:  0.82
Precision:  0.88
Recall:  0.77
F1 Score:  0.82
Confusion Matrix:
 [[39  5]
 [11 36]]
```



[47]:
```
metrics.plot_roc_curve(svm, X_test_prepared, y_test_prepared)
```

[47]: <sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x1273a0490>



1.5.9 Question 3.1.7 Hopefully you've noticed a dramatic change in performance. Discuss why you think your new data has had such a dramatic impact.

There is a pretty remarkable jump in performance, with almost a 25%-40% increase in all 5 metrics (Accuracy, Precision, Recall, F1-Score, ROC-AUC score). Since the goal of SVM is to find an optimized hyperplane in a n-dimensional feature space, by scaling the numeric features, we are now able to give it a smaller, more localized range of values in each dimension to work with that still hold the same relationship of importance without scaling. Therefore, it can much more easily find an optimal point for the hyperplane to pass through. In effect, it reduces the bias placed on any single numeric feature (based purely on numeric values) in its importance towards classification. Additionally, since we pass the entire dataset through the pipeline before splitting, we effectively ensure that the scaling factor remains the same for both training data and test data, which helps keep things consistent throughout. For the categorical variables, although the dimensionality does significantly increase when we use one-hot encoding, I still think that doesn't affect SVMs as such irrespective of what kernel choice we make. However, it once again ensures that the values are not given additional importance (Eg. category '2' is not greater than category '1') and it helps localize it to a more manageable space. Plus, since one-hot encoded vectors are sparse, in some way, although the number of dimensions are increasing, a lot of them just have 0s which makes it easier to work with since they don't make a signficant contribution to the overall result.

Furthermore, using scaled numeric features helps reduce the amount of variance in each of the numeric features (some of the numeric features have high variance as seen in patient_data.describe() from section 1 of this project), which helps the model train better since it becomes easier to find a hyperplane. Standardization of a dataset is a common requirement for many machine learning estimators: they might behave badly if the individual features do not more or less look like standard normally distributed data (e.g. Gaussian with 0 mean and unit variance).

1.5.10 Question 3.1.8 Rerun your SVM, but now modify your model parameter kernel to equal 'linear'. Again report your Accuracy, Precision, Recall, F1 scores, and Confusion matrix and plot the new ROC curve.

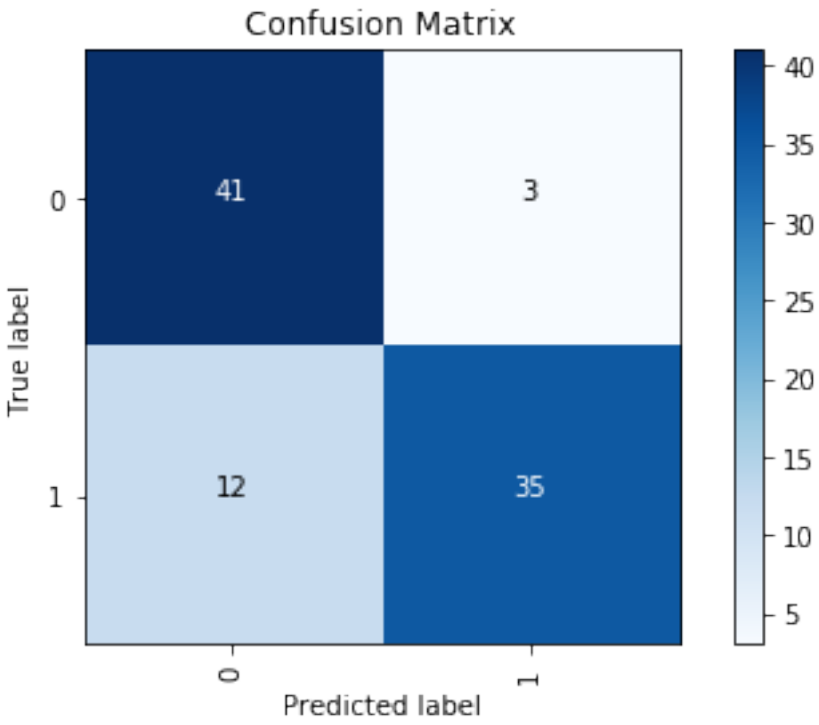
[48]:
```
svm = SVC(kernel='linear', probability=True)
svm.fit(X_train_prepared, y_train_prepared)
```

[48]:
```
SVC(C=1.0, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='scale', kernel='linear',
    max_iter=-1, probability=True, random_state=None, shrinking=True, tol=0.001,
    verbose=False)
```

[49]:
```
predictions = svm.predict(X_test_prepared)

accuracy = metrics.accuracy_score(y_test_prepared, predictions)
precision = metrics.precision_score(y_test_prepared, predictions)
recall = metrics.recall_score(y_test_prepared, predictions)
f1_score = metrics.f1_score(y_test_prepared, predictions)
```

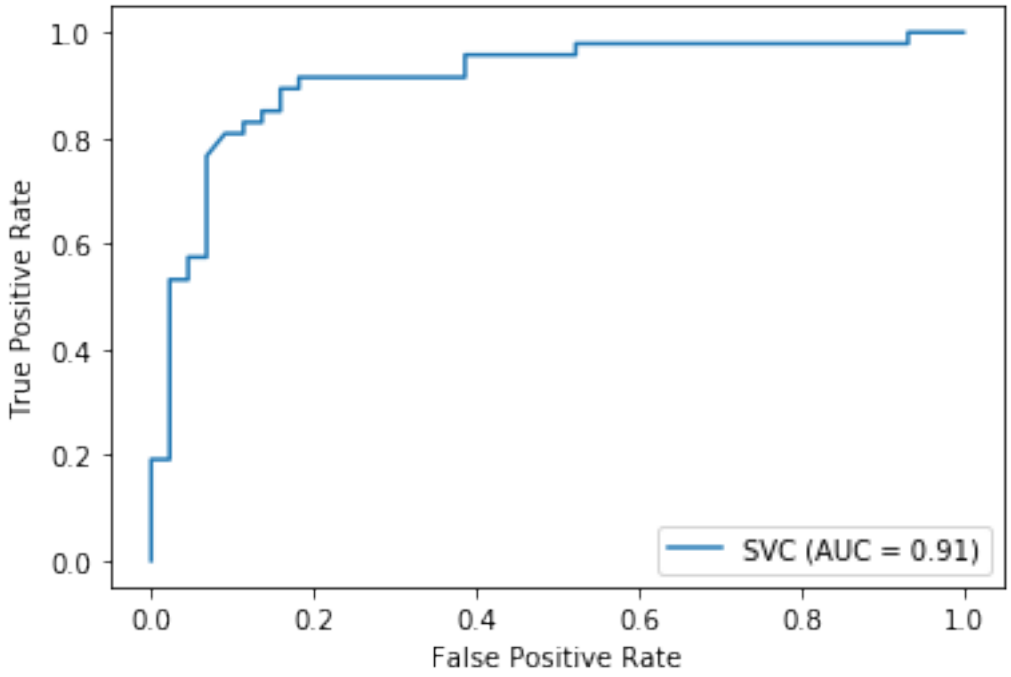
```
print("Accuracy: ", round(accuracy,2))
print("Precision: ", round(precision, 2))
print("Recall: ", round(recall, 2))
print("F1 Score: ", round(f1_score,2))

# Confusion matrix
conf_matrix = confusion_matrix(y_test_prepared, predictions)
print("Confusion Matrix: \n", conf_matrix)
draw_confusion_matrix(y_test_prepared, predictions, [0,1])
```

```
Accuracy:  0.84
Precision:  0.92
Recall:  0.74
F1 Score:  0.82
Confusion Matrix:
 [[41  3]
 [12 35]]
```



[50]:
```
metrics.plot_roc_curve(svm, X_test_prepared, y_test_prepared)
```

[50]: <sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x12725fd90>



1.5.11 Question 3.1.9 Explain what the new results you've achieved mean. Read the documentation to understand what you've changed about your model and explain why changing that input parameter might impact the results in the manner you've observed.

The main thing we change is the kernel. Linearly non-separable features often become linearly separable after they are mapped to a high dimensional feature space, and this mapping can be done via a `kernel` or `function`. From the documentation, it is apparent that the default kernel choice is `rbf`, or a radial-basis function. RBF kernels are great at determining non-linear boundaries. It is optimal for non-linear problems. However, since we know our data is linear in nature and is linearly separable, using a `linear` kernel helps map the data so that it becomes even more linearly-separable than the rbf kernel mapping in a higher dimensional space. SVMs are great at finding hyperplanes with linearly separable data, so that is why it excels in this case because the linear kernel makes the data more linearly separable than an rbf kernel because of the nature of our data (a fairly linear relationship between most attributes and the target attribute).

In this case, it is possible that using the `rbf` kernel caused overfitting, since it can create non-linear hyperplanes.

Although AUC remains the same, both accuracy and precision increase which is great for the nature of our problem. Plus, the AUC curve looks a bit more smooth in the case with the linear kernel.

Therefore, this model is arguably better than the previous one with a `rbf` kernel.

1.5.12 Logistic Regression

Knowing that we're dealing with a linearly configured dataset, let's now try another classifier that's well known for handling linear models: Logistic Regression. Logistic regression is a statistical model that in its basic form uses a logistic function to model a binary dependent variable.

1.5.13 Question 3.2.1 Implement a Logistical Regression Classifier. Review the Logistical Regression Documentation for how to implement the model. For this initial model set the solver = 'sag' and max_iter= 10). Report on the same four metrics as the SVM and graph the resulting ROC curve.

[51]:
```
# Logistic Regression
lr = LogisticRegression(solver='sag', max_iter=10)
lr.fit(X_train_prepared, y_train_prepared)
```

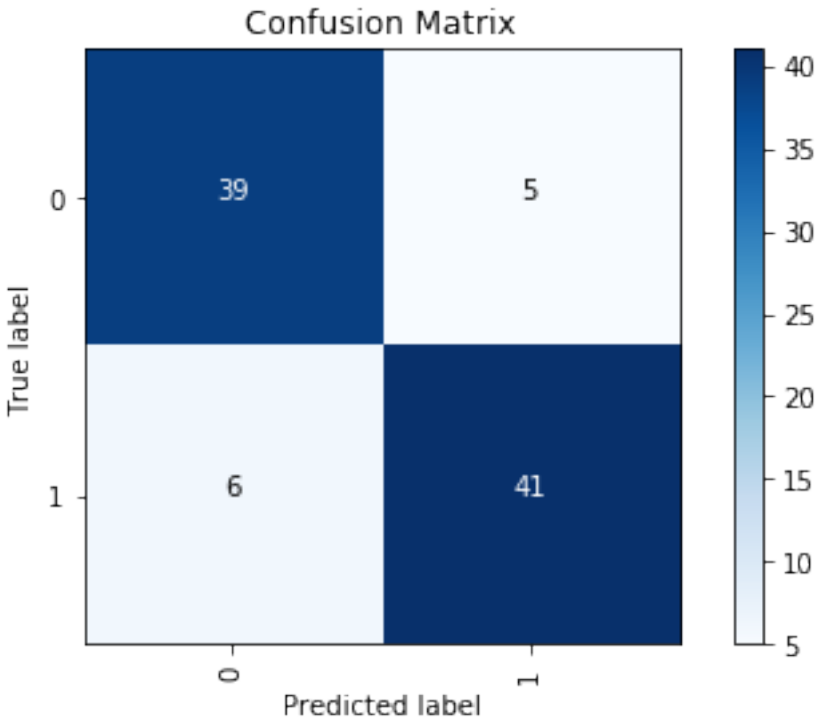
```
/usr/local/lib/python3.7/site-packages/sklearn/linear_model/_sag.py:330:
ConvergenceWarning: The max_iter was reached which means the coef_ did not
converge
  "the coef_ did not converge", ConvergenceWarning)
```

[51]:
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=10,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='sag', tol=0.0001, verbose=0,
                   warm_start=False)
```

[52]:
```
predictions = lr.predict(X_test_prepared)

accuracy = metrics.accuracy_score(y_test_prepared, predictions)
precision = metrics.precision_score(y_test_prepared, predictions)
recall = metrics.recall_score(y_test_prepared, predictions)
f1_score = metrics.f1_score(y_test_prepared, predictions)

print("Accuracy: ", round(accuracy,2))
print("Precision: ", round(precision, 2))
print("Recall: ", round(recall, 2))
print("F1 Score: ", round(f1_score,2))

# Confusion matrix
conf_matrix = confusion_matrix(y_test_prepared, predictions)
print("Confusion Matrix: \n", conf_matrix)
draw_confusion_matrix(y_test_prepared, predictions, [0,1])
```

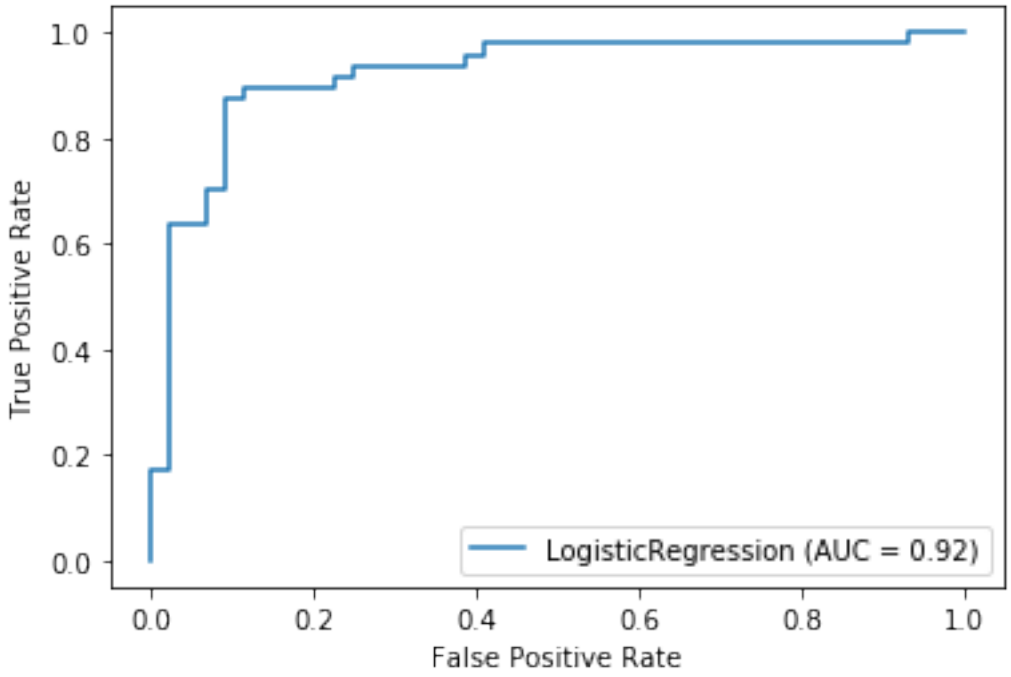
```
Accuracy:  0.88
Precision:  0.89
Recall:  0.87
F1 Score:  0.88
Confusion Matrix:
```

```
[[39  5]
 [ 6 41]]
```



[53]:
```
metrics.plot_roc_curve(lr, X_test_prepared, y_test_prepared)
```

[53]:
```
<sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x129ab8c50>
```



1.5.14 Question 3.2.2 Did you notice that when you ran the previous model you got the following warning: "ConvergenceWarning: The max_iter was reached which means the coef_ did not converge". Check the documentation and see if you can implement a fix for this problem, and again report your results.

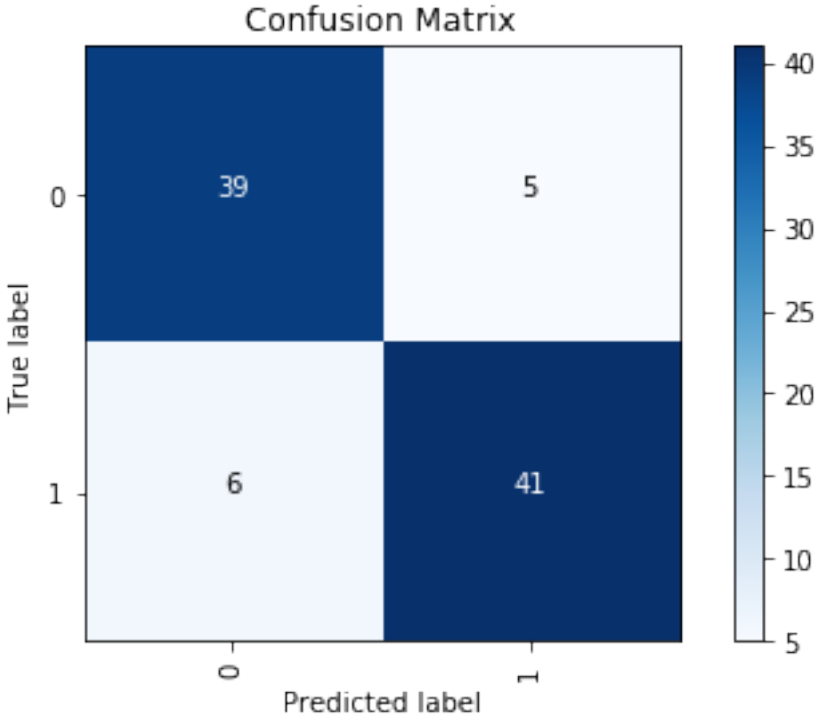
[54]:
```
# Logistic Regression
lr = LogisticRegression(solver='sag', max_iter=100)
lr.fit(X_train_prepared, y_train_prepared)
```

[54]:
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='sag', tol=0.0001, verbose=0,
                   warm_start=False)
```

[55]:
```
predictions = lr.predict(X_test_prepared)

accuracy = metrics.accuracy_score(y_test_prepared, predictions)
precision = metrics.precision_score(y_test_prepared, predictions)
recall = metrics.recall_score(y_test_prepared, predictions)
f1_score = metrics.f1_score(y_test_prepared, predictions)

print("Accuracy: ", round(accuracy,2))
print("Precision: ", round(precision, 2))
```

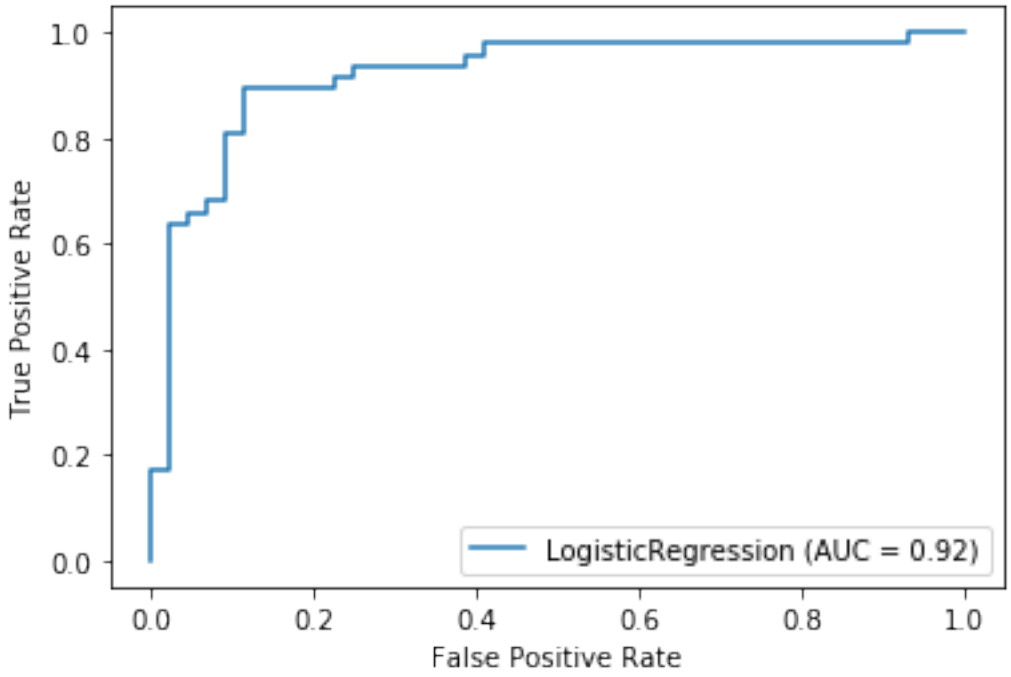
```
print("Recall: ", round(recall, 2))
print("F1 Score: ", round(f1_score,2))

# Confusion matrix
conf_matrix = confusion_matrix(y_test_prepared, predictions)
print("Confusion Matrix: \n", conf_matrix)
draw_confusion_matrix(y_test_prepared, predictions, [0,1])
```

```
Accuracy:  0.88
Precision:  0.89
Recall:  0.87
F1 Score:  0.88
Confusion Matrix:
 [[39  5]
 [ 6 41]]
```



[56]:
```
metrics.plot_roc_curve(lr, X_test_prepared, y_test_prepared)
```

[56]: <sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x129ef2dd0>



1.5.15 Question 3.2.3 Explain what you changed, and why that produced an improved outcome.

I changed max_iter from 10 to 100. This however, in my case, did not show any change in performance. I think this model just gets to a good performance really quickly (within 10 iterations) but doesn't fully converge until much later (with pretty much no improvement in accuracy) when max_iter is set to 100. I don't think this actually leads to an improved outcome overall since the accuracy, AUC and all other performance metrics have the same value, but this is more due to the nature of the problem itself and the way our data is processed. When the model converges to find the mimimum error, the accuracy still stays the same which is pretty remarkable. However, for other problems, this can make a big difference because it is important to let the model converge to maximize/minimize the objective function, because this is usually where the parameters/weights in the model are no longer adjusted and training/validation loss tends to be at its minimum.

1.5.16 Question 3.2.4 Rerun your logistic classifier, but modify the penalty = 'none', solver='sag' and again report the results.

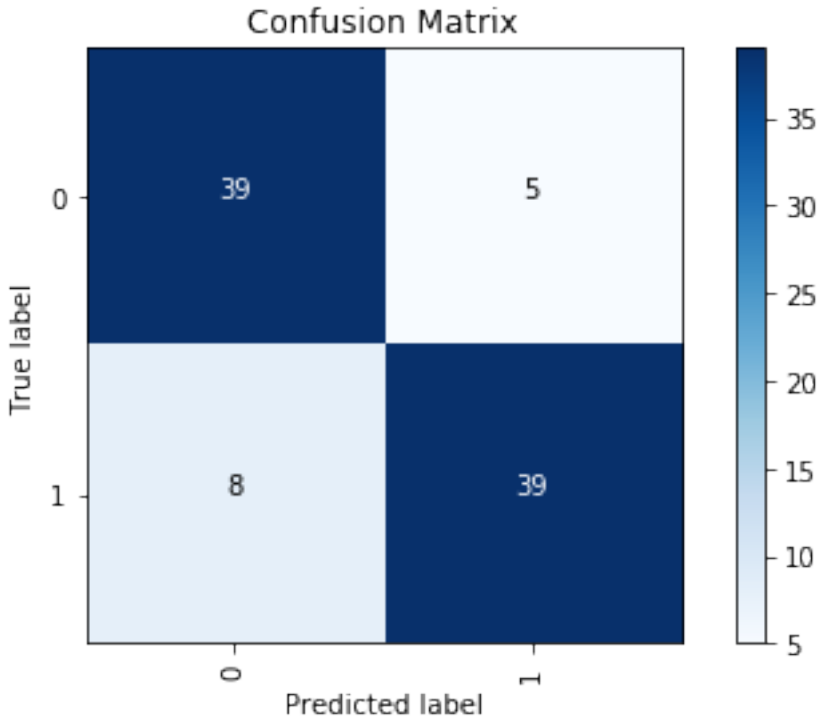
[57]:
```
lr = LogisticRegression(solver='sag', penalty='none', max_iter=3000,␣
↪random_state=42)
lr.fit(X_train_prepared, y_train_prepared)
```

[57]:
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=3000,
                   multi_class='auto', n_jobs=None, penalty='none',
                   random_state=42, solver='sag', tol=0.0001, verbose=0,
```

```
                   warm_start=False)
```

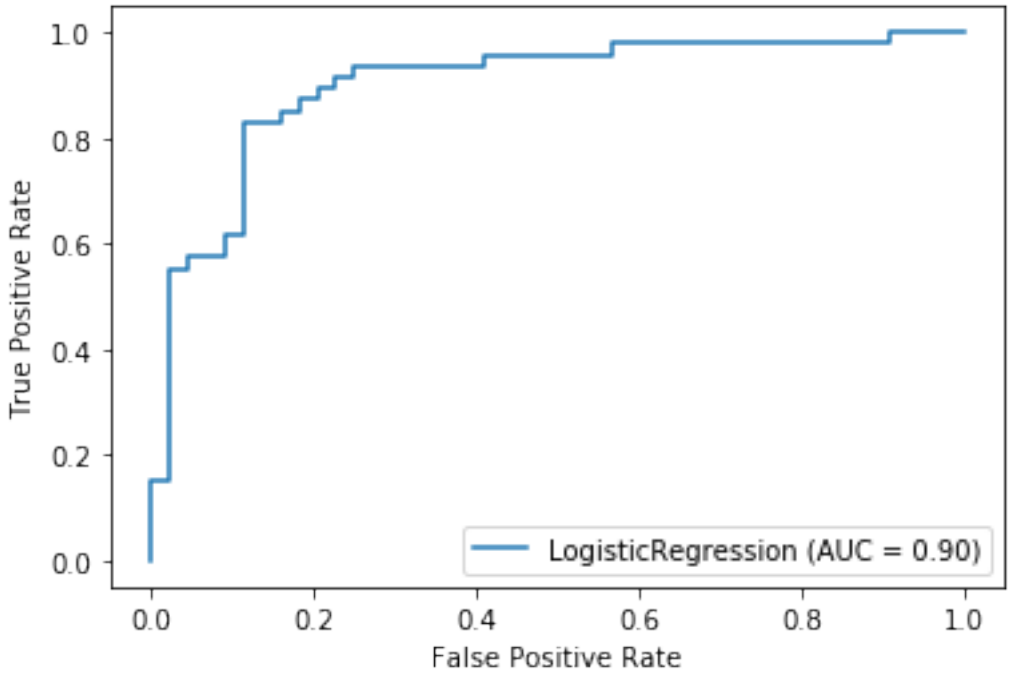
[58]:
```
predictions = lr.predict(X_test_prepared)

accuracy = metrics.accuracy_score(y_test_prepared, predictions)
precision = metrics.precision_score(y_test_prepared, predictions)
recall = metrics.recall_score(y_test_prepared, predictions)
f1_score = metrics.f1_score(y_test_prepared, predictions)

print("Accuracy: ", round(accuracy,2))
print("Precision: ", round(precision, 2))
print("Recall: ", round(recall, 2))
print("F1 Score: ", round(f1_score,2))

# Confusion matrix
conf_matrix = confusion_matrix(y_test_prepared, predictions)
print("Confusion Matrix: \n", conf_matrix)
draw_confusion_matrix(y_test_prepared, predictions, [0,1])
```

```
Accuracy:  0.86
Precision:  0.89
Recall:  0.83
F1 Score:  0.86
Confusion Matrix:
 [[39  5]
 [ 8 39]]
```



[59]:
```
metrics.plot_roc_curve(lr, X_test_prepared, y_test_prepared)
```

[59]:
```
<sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x129f60410>
```



1.5.17 Question 3.2.5 Explain what what the penalty parameter is doing in this function, what the solver method is, and why this combination likely produced a more optimal outcome.

The `penalty` parameter is 'Used to specify the norm used in the penalization'. This is essentially a way of adding on an additional term to the loss function and is usually used to help penalize bad predicitions more strictly/vigourously. The result is normally that it increases the loss for that training data point, causing the model to, say, adjust its 'weights' in a perceptron or in this case, adjust the theta values in the logistic regression equation a bit more strictly than it normally would. Removing penalty (setting penalty='none') essentially just forces the model to optimize the error normally by trying to find the average error across the entire training dataset without the norm term added to penalize the error.

I also believe that when `penalty` is set to `none`, the reason we need a significantly larger number of max_iterations for the model to converge is because the loss function no longer additionally penalizes predictions that are completely wrong, forcing the model to take longer to converge to the optimal theta values.

The `solver` method is 'Algorithm to use in the optimization problem'. Each solver tries to find the parameter weights that minimize a cost/loss function. Each solver optimizes the thetas in the

LR equation (before it is placed in a logistic function) differently, and scikit-learn appears to come with 5 different solvers: `newton-cg`, `lbfgs`, `liblinear`, `sag`, `saga`.

1.5.18 Question 3.2.6 Both logistic regression and linear SVM are trying to classify data points using a linear decision boundary, then what's the difference between their ways to find this boundary?

Logistic Regression: Logistic regression uses the concept of odds ratios to calculate the probability of a label. This is defined the ratio of the probability of an the label being true to its not. Logistic regression then models the data using the sigmoid or logistic function. The function gives an 'S' shaped curve to model the data. The curve is restricted between 0 and 1, so it is easy to apply when y is binary. From here, we apply a threshold to the probability prediction that emerges from the logistic function (usually set to 0.5 in binary classification tasks), classifying labels as 1 or 0 accordingly. Therefore, this essentially takes linear regression (the linear sum) and slaps on a logistic function so that the values can fall between a range of [0,1], and essentially reflect probabilities.

SVM: SVM's work by creating support vectors (you can imagine imaginary lines), which are the data points that lie closest to the decision surface (or hyperplane). You can also visualize this as the point from one of the labels that is closest to a point from the other label. They are the data points most difficult to classify, and have direct bearing on the optimum location of the decision surface. Once we have the support vectors, our goal is to create a line of best fit (the optimal hyperplane), and although there are virtually infinite lines (hyperplanes) that can pass through between both the support vectors that would separate label A and label B, we pick the one that maximizes the margin around the separating hyperplane, that is, the one which has maximum distance from both the support vectors.

Therefore, the approaches to finding the boundary of separation are different for logistic regression and SVM and result in different performance scores across all the metrics.

1.5.19 Clustering Approaches

Let us now try a different approach to classification using a clustering algorithm. Specifically, we're going to be using K-Nearest Neighbor, one of the most popular clustering approaches.

1.5.20 K-Nearest Neighbor

1.5.21 Question 3.3.1 Implement a K-Nearest Neighbor algorithm on our data and report the results. For this initial implementation simply use the default settings. Refer to the KNN Documentation for details on implementation. Report on the accuracy of the resulting model.

[60]:
```
knn = KNeighborsClassifier()
knn.fit(X_train_prepared, y_train_prepared)
```

[60]:
```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=5, p=2,
                     weights='uniform')
```

[61]:
```
predictions = knn.predict(X_test_prepared)
accuracy = metrics.accuracy_score(y_test_prepared, predictions)
print("Accuracy:", round(accuracy, 2))
```

```
Accuracy: 0.85
```

1.5.22 Question 3.3.2 For clustering algorithms, we use different measures to determine the effectiveness of the model. Specifically here, we're interested in the Homogeneity Score, Completeness Score, V-Measure, Adjusted Rand Score, and Adjusted Mutual Information. Calculate each score (hint review the SKlearn Metrics Clustering documentation for how to implement).

[62]:
```
homogeneity_score = metrics.homogeneity_score(y_test_prepared, predictions)
completeness_score = metrics.completeness_score(y_test_prepared, predictions)
v_measure = metrics.v_measure_score(y_test_prepared, predictions)
adjusted_rand_score = metrics.adjusted_rand_score(y_test_prepared, predictions)
adjusted_mi_score = metrics.adjusted_mutual_info_score(y_test_prepared,␣
↪predictions)

print("Homogeneity Score:", round(homogeneity_score,2))
print("Completeness Score:", round(completeness_score, 2))
print("V-measure:", round(v_measure, 2))
print("Adjusted Rand Score:", round(adjusted_rand_score,2))
print("Adjusted Mutual Information:", round(adjusted_mi_score,2))
```

```
Homogeneity Score: 0.4
Completeness Score: 0.4
V-measure: 0.4
Adjusted Rand Score: 0.47
Adjusted Mutual Information: 0.39
```

1.5.23 Question 3.3.3 Explain what each score means and interpret the results for this particular model.

Metrics

- **Homogeneity Score**: Homogeneity Score is the ratio for how uniform/pure the clustering is. It is a proportion between 0 and 1, with 1 indicating that clusters contain data points only from a single class. This is effictvely equivalent to a Precision Score in non-clustering based classification algorithms. This metric is independent of the absolute values of the labels: a permutation of the class or cluster label values won't change the score value in any way.
- **Completeness Score**: Completeness Score is the ratio for how comprehensive a cluster is in capturing elements from a single class. It is a proportion between 0 and 1, with 1 indicating that all data points in a given class are in the same cluster. This is equivalent to a Recall Score in non-clustering based classification algorithms. A good clustering algorithm should assign all samples with the same true label to the same cluster, and therefore, have a high completeness score.
- **V-Measure**: It is a weighted average of homogeneity score and completeness score (also

often referred to as "harmonic average"). It is similar to F1-scores for non-clustering based classification algorithms. The primary advantage of this evaluation metric is that it is independent of the number of class labels, the number of clusters, the size of the data and the clustering algorithm used and is a very reliable metric.

- **Adjusted Rand Score**: It is a measure of the similarity between two data clusterings. An adjusted rand score of 0 is bad because it is the equivalent of just putting all of your data in random buckets. In other words, it measures how similar the instances present in the respective clusters are. It has a range of values between [-1,1]. The Rand Index score however is highly dependent on the number of clusters and random chance clusterings, so the Adjusted Rand Index (ARI), corrects for that. Often, it is considered synonymous with accuracy, but it actually extends beyond that idea. Use ARI when the ground truth clustering has large equal sized clusters. Given the knowledge of the ground truth class assignments labels_true and our clustering algorithm assignments of the same samples labels_pred, the adjusted Rand index is a function that measures the similarity of the two assignments, ignoring permutations and with chance normalization.
- **Adjusted Mutual Information**: The Mutual Information (MI) score expresses the extent to which observed frequency of co-occurrence differs from what we would expect (statistically speaking). In statistically pure terms this is a measure of the strength of association between two variables (or two clusterings in our case). Adjusted MI (AMI) is an adjustment of MI to account for chance. It accounts for the fact that the MI is generally higher for two clusterings with a larger number of clusters, regardless of whether there is actually more information shared. Usa AMI when the ground truth clustering is unbalanced and there exist small clusters. Given the knowledge of the ground truth class assignments labels_true and our clustering algorithm assignments of the same samples labels_pred, the Mutual Information is a function that measures the agreement of the two assignments, ignoring permutations. AMI adjusts MI for chance.

Analysis of results – From the scores, we can see that all of the performance metrics lie between 0.39 and 0.47, with a majority equal to 0.4. These scores are very low, almost indicating that our clustering algorithm fails to assign a majority of the class labels into a single cluster, resulting in the creation of impure clusters that cannot distinguish the majority class correctly. From a slightly more visual perspective, since KNN picks the nearest 'n' neighbours and picks the majority class from that group as the label for that example, it could hint that our data points for both positive and negative classes are very close by, so much so that the linearly separable hyperplane is arguably very thin (with small margins), or that the data points are widely spread out so that the majority class often tends to be the opposite class, resulting in an incorrect classification. Another reason could be that our test and train data have different distributions, so while this works well and results in optimal clusters for our training data, the test data has a different distribution where this 'n' value for number of neighbours from our training dataset just results in incorrect predictions for the test dataset.

As we're beginning to see, the input parameters for your model can dramatically impact the performance of the model. How do you know which settings to choose? Studying the models and studying your datasets are critical as they can help you anticipate which models and settings are likely to produce optimal results. However sometimes that isn't enough, and a brute force method is necessary to determine which parameters to use. For this next question we'll attempt to optimize a parameter using a brute force approach.

1.5.24 Question 3.3.4 Parameter Optimization. The KNN Algorithm includes an n_neighbors attribute that specifies how many neighbors to use when developing the cluster. (The default value is 5, which is what your previous model used.) Lets now try n values of: 1, 2, 3, 5, 10, 20, 50, and 100. Run your model for each value and report the 6 measures (5 clustering specific plus accuracy) for each. Report on which n value produces the best accuracy and V-Measure. (HINT leverage python's ability to loop to run through the array and generate results without needing to manually code each iteration).

[63]:
```
def generate_scores(y_test_prepared, predictions):
    accuracy = metrics.accuracy_score(y_test_prepared, predictions)
    homogeneity_score = metrics.homogeneity_score(y_test_prepared, predictions)
    completeness_score = metrics.completeness_score(y_test_prepared,␣
 ↪predictions)
    v_measure = metrics.v_measure_score(y_test_prepared, predictions)
    adjusted_rand_score = metrics.adjusted_rand_score(y_test_prepared,␣
 ↪predictions)
    adjusted_mi_score = metrics.adjusted_mutual_info_score(y_test_prepared,␣
 ↪predictions)
    print("Accuracy:", round(accuracy, 2))
    print("Homogeneity Score:", round(homogeneity_score,2))
    print("Completeness Score:", round(completeness_score, 2))
    print("V-measure:", round(v_measure, 2))
    print("Adjusted Rand Score:", round(adjusted_rand_score,2))
    print("Adjusted Mutual Information:", round(adjusted_mi_score,2))
    print("\n")

for neigh_count in [1,2,3,5,10,20,50,100]:
    knn = KNeighborsClassifier(n_neighbors=neigh_count)
    knn.fit(X_train_prepared, y_train_prepared)
    predictions = knn.predict(X_test_prepared)
    print("For n_neighbours={0}:".format(neigh_count))
    generate_scores(y_test_prepared, predictions)
```

```
For n_neighbours=1:
Accuracy: 0.75
Homogeneity Score: 0.19
Completeness Score: 0.19
V-measure: 0.19
Adjusted Rand Score: 0.24
Adjusted Mutual Information: 0.18


For n_neighbours=2:
Accuracy: 0.79
Homogeneity Score: 0.34
Completeness Score: 0.38
```

```
V-measure: 0.36
Adjusted Rand Score: 0.33
Adjusted Mutual Information: 0.35


For n_neighbours=3:
Accuracy: 0.84
Homogeneity Score: 0.36
Completeness Score: 0.36
V-measure: 0.36
Adjusted Rand Score: 0.44
Adjusted Mutual Information: 0.36


For n_neighbours=5:
Accuracy: 0.85
Homogeneity Score: 0.4
Completeness Score: 0.4
V-measure: 0.4
Adjusted Rand Score: 0.47
Adjusted Mutual Information: 0.39


For n_neighbours=10:
Accuracy: 0.82
Homogeneity Score: 0.38
Completeness Score: 0.39
V-measure: 0.39
Adjusted Rand Score: 0.41
Adjusted Mutual Information: 0.38


For n_neighbours=20:
Accuracy: 0.81
Homogeneity Score: 0.33
Completeness Score: 0.33
V-measure: 0.33
Adjusted Rand Score: 0.39
Adjusted Mutual Information: 0.33


For n_neighbours=50:
Accuracy: 0.8
Homogeneity Score: 0.32
Completeness Score: 0.33
V-measure: 0.33
Adjusted Rand Score: 0.36
Adjusted Mutual Information: 0.32
```

```
For n_neighbours=100:
Accuracy: 0.77
Homogeneity Score: 0.31
Completeness Score: 0.34
V-measure: 0.32
Adjusted Rand Score: 0.28
Adjusted Mutual Information: 0.32
```

From the above results, we can see that the model with **`n_neighbours=5`** achieves the highest accuracy score and v-measure score.

1.5.25 Question 3.3.5 When are clustering algorithms most effective, and what do you think explains the comparative results we achieved?

Cluster analysis or clustering is the task of grouping a set of objects in such a way that objects in the same group (called a cluster) are more similar (in some sense) to each other than to those in other groups (clusters). Therefore, clustering algorithms are **most effective** when: 1. Understanding basic structures in the data, especially when we do not already know too much about the data. 2. When we have non-linear or multi-clustered data. 3. Datasets with fairly obvious different clusters, such as types of different diseases, different fraud types 4. Visualizing text data once you have embedding representations, to see what kind of sentences/phrases are similar to each other. Clustering might be great for unstructured data, because it actually adds some for of structure to it that can then be visualized.

In 3.3.4, we see that we get the most optimal value for `n_neighbours=5`. We can also see that as n_neighbours increases from 1 to 5, both accuracy and v-measure increase. Both of these peak at 5. After 5, both of these start to decrease. This poses that near every datapoint, we have a majority where 3 belong to the correct class and 2 belong to the wrong class, and so the majority appears to win in this case every time. The fact that our scores increase with an increasing number of n_neighbours at the beginning suggests that the optimal clusters are a bit spaced out (not all together), giving us a great 85% on this task. However, what is concerning is that even at the highest accuracy, our v-measure and all other scores are extremely poor, suggesting that clustering may not be as good of a algorithm choice for this dataset as linear regression or SVM.

In Summary To wrap up, it appears that our dataset is indeed fairly linearly separable, encouraging the use and creation of models that work best on linearly separable data. Although SVM and logistic regression scores look comparable, I would personally pick logistic regression as the model of choice for its extremely high scores across the board (accuracy, precision, recall and F1-score). Given that our task pertains to predicting the risk of a heart disease, we want to have high accuracy and high recall score, and the logistic regression model appears to have the highest score. Intuitively, this makes sense since most of the features have a linear relationship with the target variable. The other advantage is that we can actually extract and use the coefficient scores to map out the importance of each individual feature (according to the model), which would help tremendously with explainability, which once again, becomes very important for this task in particular since we want to be able to explain our results and not leave it entirely up to a machine

learning model.

The model's scores could probably be improved even further by dropping a few redundant features, and by playing around with other parameters that we can configure for the logistic regression model.