CSM148 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</u>
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.

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 (0-4) colored by flourosopy
- **thal:** 1 = normal; 2 = fixed defect; 3 = reversable defect
- **Sick:** Indicates the presence of Heart disease (True = Disease; False = No disease)

Loading Essentials and Helper Functions

In [4]:

```
#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
from sklearn.metrics import confusion_matrix, precision_score, recall_score, f1_sco
re, accuracy_score
import sklearn.metrics.cluster as smc
from sklearn.model_selection import KFold


from matplotlib import pyplot
import itertools

%matplotlib inline

import random 
 
random.seed(42) 
```

In [5]:

```
# 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)
```

In [6]:

```
# 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()
```

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.)

In [7]:

```
data = pd.read_csv(os.getcwd() + "/heartdisease.csv")
```

Now that our data is loaded, let's take a closer look at the dataset we're working with. Use the head method, the describe method, and the info method to display some of the rows so we can visualize the types of data fields we'll be working with.

In [8]:

```
data.head()
```

Out[8]:

	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

Sometimes data will be stored in different formats (e.g., string, date, boolean), but many learning methods work strictly on numeric inputs. Call the info method to determine the datafield type for each column. Are there any that are problemmatic and why?

In [9]:

```
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
```

Certain variables that are boolean in nature (such as sex) are being encoded as int64 while other boolean variables such as sick are actually bool. We may need to convert all variables to int64 for consistency.

Determine if we're dealing with any null values. If so, report on which columns?

In [10]:

```
data.isna().values.any()
```

Out[10]:

```
False
```

There are no null values in the dataframe, so we will not need to impute any cells.

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. (hint: try label encoder or .astype()

In [11]:

```
data['sick'] = data['sick'].astype(int)
raw_data = data.copy()
data = data.drop(['sick'], axis=1)
data.head()
```

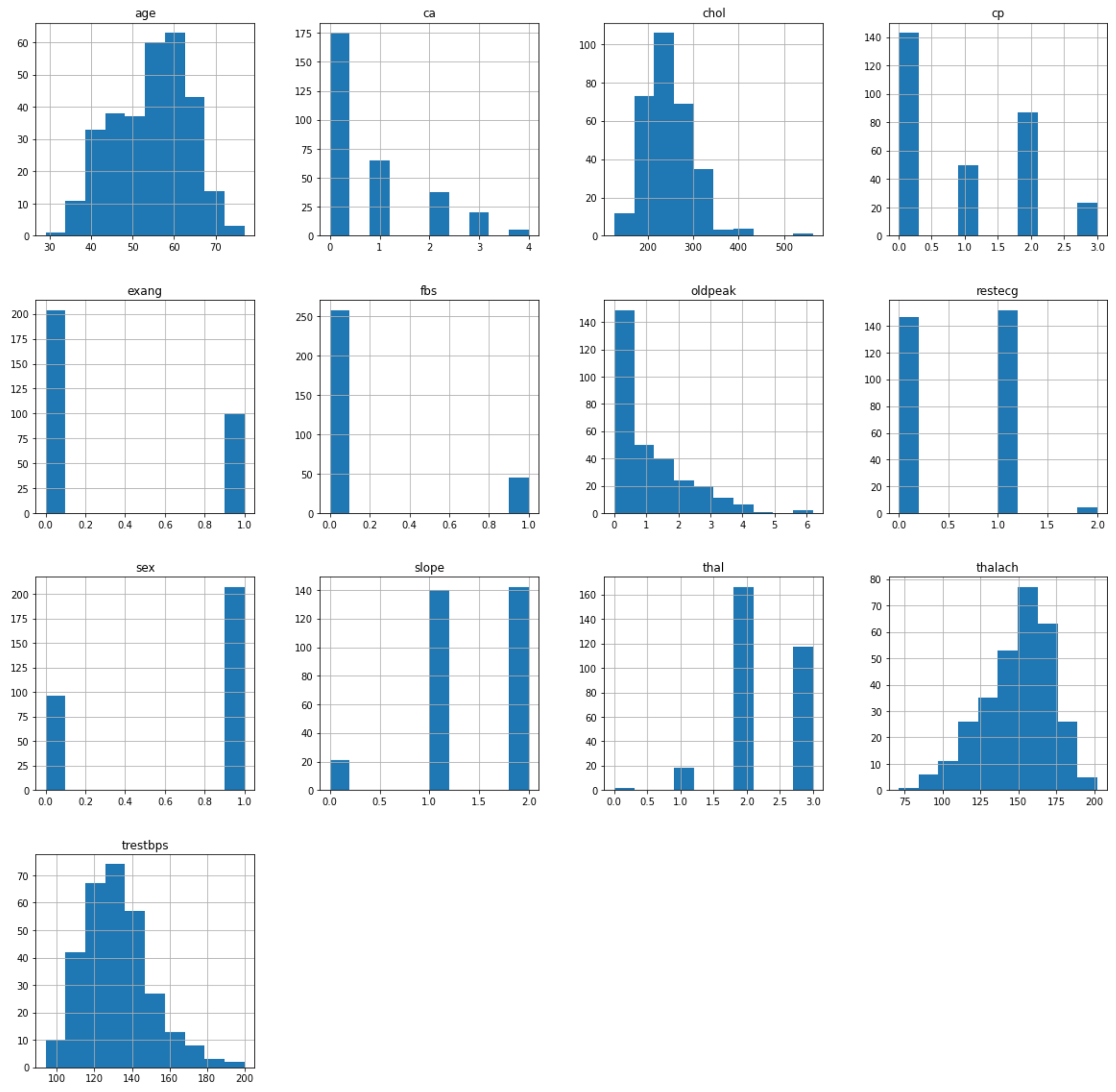
Out[11]:

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

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?

In [12]:

```
data.hist(figsize = (20,20))
plt.show()
```



- Binary - sex, exang, fbs
- Limited selection - slope, ca, cp
- Normal - thalach (skew left), tretsbps (skew right), age, oldpeak (skew right)

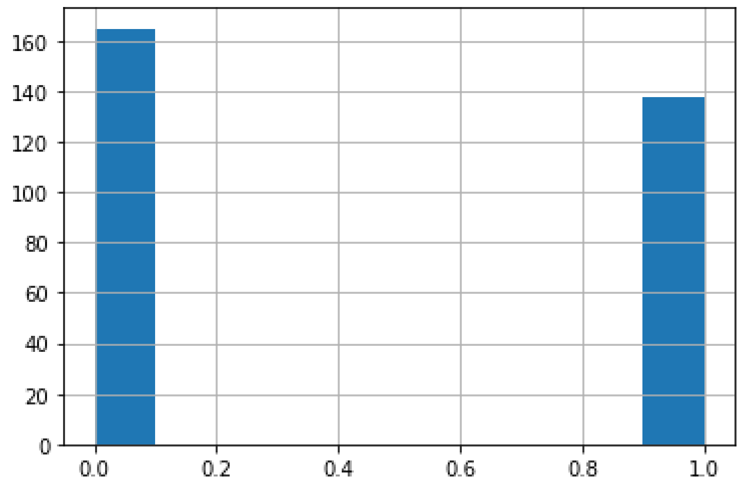
We also want to make sure we are dealing with a balanced dataset. In this case, we want to confirm whether or not we have an equitable number of sick and healthy individuals to ensure that our classifier will have a sufficiently balanced dataset to adequately classify the two. Plot a histogram specifically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

In [13]:

```
raw_data['sick'].hist()
raw_data['sick'].value_counts()
```

Out[13]:

```
0    165
1    138
Name: sick, dtype: int64
```



The data is pretty decently balanced between sick and not sick (54%/46% split).

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.

Certain datasets are heavily imbalanced such as predicting a natural disaster. To make this dataset balanced we would need to discard a large amount of the heavier weighted class or we would need to generate data for the lighter weighted class. The first case reduces the information in our model, while the second case can introduce randomness or bias to the model.

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)?

In [14]:

```
correlations = raw_data.corr()
correlations['sick']
```

Out[14]:

```
age         0.225439
sex         0.280937
cp         -0.433798
trestbps    0.144931
chol        0.085239
fbs         0.028046
restecg    -0.137230
thalach    -0.421741
exang       0.436757
oldpeak     0.430696
slope      -0.345877
ca          0.391724
thal        0.344029
sick        1.000000
Name: sick, dtype: float64
```

Thalach, cp, and slope have strong negative correlations.

Exang, oldpeak, ca, and thal have positive correlations.

Part 2. Prepare the Data and run a KNN Model

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!

Save the label column as a separate array and then drop it from the dataframe.

In [15]:

```
labels = raw_data['sick']
data = raw_data.drop(columns = ['sick'])
```

First Create your 'Raw' unprocessed training data by dividing your dataframe into training and testing cohorts, with your training cohort consisting of 80% 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.

In [16]:

```
output = train_test_split(data, labels, test_size = 0.2, random_state = 42)
raw_x_train, raw_x_test, raw_y_train, raw_y_test = output

print(raw_x_train.shape)
print(raw_y_train.shape)
print(raw_x_test.shape)
print(raw_y_test.shape)
```

```
(242, 13)
(242,)
(61, 13)
(61,)
```

In lecture we learned about K-Nearest Neighbor. One thing we noted was because KNN's rely on Euclidean distance, they are highly sensitive to the relative magnitude of different features. Let's see that in action! 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 (https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html) for details on implementation. Report on the accuracy of the resulting model.

In [17]:

```
# k-Nearest Neighbors algorithm
k_nearest = KNeighborsClassifier()
k_nearest.fit(raw_x_train, raw_y_train)
print("Train score: {}".format(k_nearest.score(raw_x_train, raw_y_train)))
```

```
Train score: 0.7603305785123967
```

Report on model Accuracy

Our model does not perform very well without using any scaling.

Now implement a pipeline of your choice. You can opt to handle categoricals however you wish, however please scale your numeric features using standard scaler

Pipeline:

In [18]:

```
from sklearn.preprocessing import StandardScaler

def run_pipeline(data):
    discrete_cat = ['cp', 'slope', 'ca', 'thal', 'restecg']
    one_hot_cp = pd.get_dummies(data['cp'],prefix='cp',drop_first=True)
    one_hot_slop = pd.get_dummies(data['slope'],prefix='slope',drop_first=True)
    one_hot_ca = pd.get_dummies(data['ca'],prefix='ca',drop_first=True)
    one_hot_thal = pd.get_dummies(data['thal'],prefix='thal',drop_first=True)
    one_hot_rest = pd.get_dummies(data['restecg'],prefix='restecg',drop_first=True)


    data = pd.concat([data, one_hot_cp, one_hot_slop, one_hot_ca, one_hot_thal, one
_hot_rest], axis=1)
    data.drop(columns=discrete_cat, inplace=True)
    data.head()

    scaler = StandardScaler()
    processed_data = data
    scaler.fit(processed_data)
    processed_data = scaler.transform(processed_data)
    
    return processed_data

processed_data = run_pipeline(data)
print(processed_data)
```

```
[[ 0.9521966   0.68100522  0.76395577 ... -0.79311554 -1.00330579
  -0.11566299]
 [-1.91531289  0.68100522 -0.09273778 ... -0.79311554  0.9967051
  -0.11566299]
 [-1.47415758 -1.46841752 -0.09273778 ... -0.79311554 -1.00330579
  -0.11566299]
 ...
 [ 1.50364073  0.68100522  0.70684287 ...  1.26085034  0.9967051
  -0.11566299]
 [ 0.29046364  0.68100522 -0.09273778 ...  1.26085034  0.9967051
  -0.11566299]
 [ 0.29046364 -1.46841752 -0.09273778 ... -0.79311554 -1.00330579
  -0.11566299]]
```

Now split your pipelined data into an 80/20 split and again run the same KNN, and report out on it's accuracy. Discuss the implications of the different results you are obtaining.

In [19]:

```
# k-Nearest Neighbors algorithm
output = train_test_split(processed_data, labels, test_size = 0.2, random_state = 4
2)
x_train, x_test, y_train, y_test = output

print(x_train.shape)
print(y_train.shape)
print(x_test.shape)
print(y_test.shape)

k_nearest = KNeighborsClassifier()
k_nearest.fit(x_train, y_train)
```

```
(242, 22)
(242,)
(61, 22)
(61,)
```

Out[19]:

```
KNeighborsClassifier()
```

In [20]:

```
# Accuracy
print("Train score: {}".format(k_nearest.score(x_train, y_train)))
```

```
Train score: 0.8677685950413223
```

After scaling we lift the training score from ~65 to ~86%

Parameter Optimization. As we saw in lecture, 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, 7, 9, 10, 20, and 50. Run your model for each value and report the accuracy for each. (HINT leverage python's ability to loop to run through the array and generate results without needing to manually code each iteration).

In [21]:

```
n_vals = [1,2,3,5,6,9,10,20,50]
for n in n_vals:
    k_nearest = KNeighborsClassifier(n_neighbors=n)
    k_nearest.fit(x_train, y_train)
    print("N: {}, Train score: {}".format(n, k_nearest.score(x_train, y_train)))
```

```
N: 1, Train score: 1.0
N: 2, Train score: 0.8842975206611571
N: 3, Train score: 0.8925619834710744
N: 5, Train score: 0.8677685950413223
N: 6, Train score: 0.8677685950413223
N: 9, Train score: 0.859504132231405
N: 10, Train score: 0.871900826446281
N: 20, Train score: 0.8347107438016529
N: 50, Train score: 0.8223140495867769
```

Part 3. Additional Learning Methods

So we have a model that seems to work well. But let's see if we can do better! To do so we'll employ multiple learning methods and compare result.

Linear Decision Boundary Methods

Logistic Regression

Let's now try another classifier, we introduced in lecture, one 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.

Implement a Logistical Regression Classifier. Review the Logistical Regression Documentation (https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html for how to implement the model.

In [22]:

```
# Logistic Regression
logreg = LogisticRegression()
logreg.fit(x_train, y_train)
y_predictions = logreg.predict(x_test)
y_true = list(y_test)
```

This time report four metrics: Accuracy, Precision, Recall, and F1 Score, and plot a Confusion Matrix.

In [23]:

```
print(f"Accuracy: {str(accuracy_score(y_true, y_predictions))}")
print(f"Precision: {str(precision_score(y_true, y_predictions))}")
print(f"Recall: {str(recall_score(y_true, y_predictions))}")
print(f"F1 Score: {str (f1_score(y_true, y_predictions))}")
print(f"Confusion matrix: {confusion_matrix(y_true, y_predictions)}")
```

```
Accuracy: 0.8688524590163934
Precision: 0.8620689655172413
Recall: 0.8620689655172413
F1 Score: 0.8620689655172413
Confusion matrix: [[28  4]
 [ 4 25]]
```

Discuss what each measure is reporting, why they are different, and why are 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 is the percentage of guesses that are correct with a given model. Accuracy can be used when the class distribution is similar while F1-score is a better metric when there are imbalanced classes.

Precision is a measure of how relevant the selected items are. Precision matters a lot when the cost of acting on an action is high.

Recall is a measure of how much coverage the model has. Recall may be very important in an early cancer diagnosis since we want to make sure that we capture as much as possible. False positives are okay in this case and can be addressed after the initial screening.

F1 score combines precision and recall. F1 is good to use because it encapsulates the tradeoff between precision and recall.

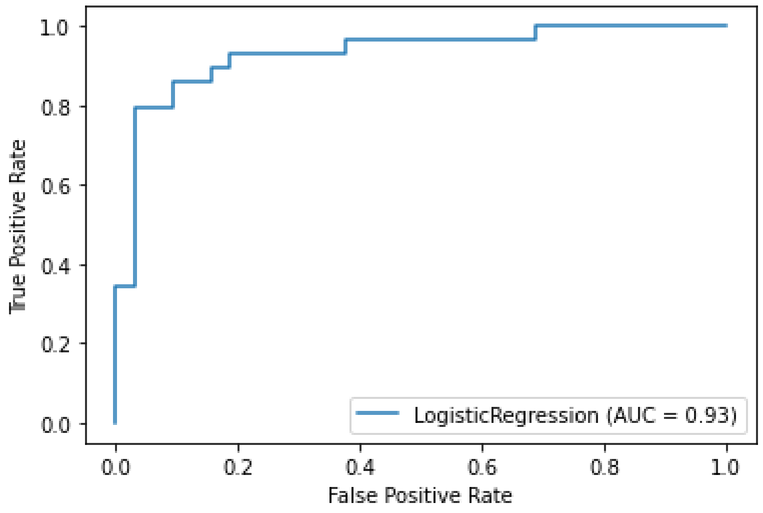
The confusion matrix shows the true positives, false positives, true negatives, and false negatives which is important to compute the metrics above.

Graph the resulting ROC curve of the model

In [24]:

```
from sklearn.metrics import plot_roc_curve

roc = plot_roc_curve(logreg, x_test, y_test)
plt.show()
```



Describe what an ROC curve is and what the results of this graph seem to be indicating

The ROC curve shows the tradeoff between FP and TP. An ideal model has TP=1 and FP=0 so we want to be in the top left corner. Models in the bottom left and top right are less desireable. The higher the area under the curve, the better our model is.

Let's tweak a few settings. First let's set your solver to 'sag', your max_iter= 10, and set penalty = 'none' and rerun your model. Report out the same metrics. Let's see how your results change!

In [25]:

```
# Logistic Regression
logreg = LogisticRegression(solver ='sag', max_iter = 10, penalty='none')
logreg.fit(x_train, y_train)
y_predictions = logreg.predict(x_test)
y_true = list(y_test)
print(f"Accuracy: {str(accuracy_score(y_true, y_predictions))}")
print(f"Precision: {str(precision_score(y_true, y_predictions))}")
print(f"Recall: {str(recall_score(y_true, y_predictions))}")
print(f"F1 Score: {str (f1_score(y_true, y_predictions))}")
print(f"Confusion matrix: {confusion_matrix(y_true, y_predictions)}")
```

```
Accuracy: 0.8688524590163934
Precision: 0.8620689655172413
Recall: 0.8620689655172413
F1 Score: 0.8620689655172413
Confusion matrix: [[28  4]
 [ 4 25]]
```

```
/Users/prithvikannan/anaconda3/lib/python3.8/site-packages/sklearn/line
ar_model/_sag.py:329: ConvergenceWarning: The max_iter was reached whic
h means the coef_ did not converge
  warnings.warn("The max_iter was reached which means "
```

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.

In [26]:

```
# Logistic Regression
logreg = LogisticRegression(solver ='sag', max_iter = 100)
logreg.fit(x_train, y_train)
y_predictions = logreg.predict(x_test)
y_true = list(y_test)
print(f"Accuracy: {str(accuracy_score(y_true, y_predictions))}")
print(f"Precision: {str(precision_score(y_true, y_predictions))}")
print(f"Recall: {str(recall_score(y_true, y_predictions))}")
print(f"F1 Score: {str (f1_score(y_true, y_predictions))}")
print(f"Confusion matrix: {confusion_matrix(y_true, y_predictions)}")
```

```
Accuracy: 0.8688524590163934
Precision: 0.8620689655172413
Recall: 0.8620689655172413
F1 Score: 0.8620689655172413
Confusion matrix: [[28  4]
 [ 4 25]]
```

Explain what you changed, and why do you think, even though you 'fixed' the problem, that you may have harmed the outcome. What other Parameters you set may have impacted this result?

I increased max_iter to 100 and removed the penalty. It's possible that with more iterations we are overfitting the model or just wasting compute time.

Rerun your logistic classifier, but modify the penalty = 'l1', solver='liblinear' and again report the results.

In [27]:

```
# Logistic Regression
logreg = LogisticRegression(solver ='liblinear', max_iter = 100, penalty='l1')
logreg.fit(x_train, y_train)
y_predictions = logreg.predict(x_test)
y_true = list(y_test)
print(f"Accuracy: {str(accuracy_score(y_true, y_predictions))}")
print(f"Precision: {str(precision_score(y_true, y_predictions))}")
print(f"Recall: {str(recall_score(y_true, y_predictions))}")
print(f"F1 Score: {str (f1_score(y_true, y_predictions))}")
print(f"Confusion matrix: {confusion_matrix(y_true, y_predictions)}")
```

```
Accuracy: 0.8688524590163934
Precision: 0.8620689655172413
Recall: 0.8620689655172413
F1 Score: 0.8620689655172413
Confusion matrix: [[28  4]
 [ 4 25]]
```

Explain what what the two solver approaches are, and why the liblinear likely produced the optimal outcome.

Again we got the same output scores.

SAG only supports L2 regularization while liblinear can do L1.

For small datasets, ‘liblinear’ is a good choice, whereas ‘sag’ and ‘saga’ are faster for large ones.

We also played around with different penalty terms (none, L1 etc.) Describe what the purpose of a penalty term is and how an L1 penalty works.

The penalty parameter is used for regularization which prevents the model from being too complex. If penalty='none' then we do not use regularization. Otherwise the coefficients are part of the loss function and help to prevent overfitting.

We used sag (stochastic avg gradient) which is a form of gradient descent based on a random sample. Another option is to use lbfgs

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.

Implement a Support Vector Machine classifier on your pipelined data. Review the SVM Documentation (https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html) for how to implement a model. For this implementation you can simply use the default settings, but set probability = True.

In [28]:

```
# SVM
svc = SVC(probability = True)
model = svc.fit(x_train, y_train)
```

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

In [29]:

```
y_predictions = model.predict(x_test)
y_true = list(y_test)
print(f"Accuracy: {str(accuracy_score(y_true, y_predictions))}")
print(f"Precision: {str(precision_score(y_true, y_predictions))}")
print(f"Recall: {str(recall_score(y_true, y_predictions))}")
print(f"F1 Score: {str (f1_score(y_true, y_predictions))}")
print(f"Confusion matrix: {confusion_matrix(y_true, y_predictions)}")
```

```
Accuracy: 0.819672131147541
Precision: 0.8
Recall: 0.8275862068965517
F1 Score: 0.8135593220338982
Confusion matrix: [[26  6]
 [ 5 24]]
```

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.

In [30]:

```
# SVM
svc = SVC(probability = True, kernel='linear')
model = svc.fit(x_train, y_train)
```

In [31]:

```
y_predictions = model.predict(x_test)
y_true = list(y_test)
print(f"Accuracy: {str(accuracy_score(y_true, y_predictions))}")
print(f"Precision: {str(precision_score(y_true, y_predictions))}")
print(f"Recall: {str(recall_score(y_true, y_predictions))}")
print(f"F1 Score: {str (f1_score(y_true, y_predictions))}")
print(f"Confusion matrix: {confusion_matrix(y_true, y_predictions)}")
```

```
Accuracy: 0.8524590163934426
Precision: 0.8333333333333334
Recall: 0.8620689655172413
F1 Score: 0.847457627118644
Confusion matrix: [[27  5]
 [ 4 25]]
```

Explain the 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.

Here we can see accuracy, precision, recall, and f1 all increase after applying the linear kernel SVM vs the default RBF kernel svm. Linear kernel tries to create a linear decision boundary whereas an RBF kernel can create a decision boundary of arbitrary shape. In this case, it may be that the data is linearly separable in which case it makes more sense to use a linear kernel, which is supported by our results.

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?

Linear SVM tries to identify the best margin to separate the classes. Logistic regression can have different decision boundaries with different weights near the optimal point. Additioanally, SVM is based on geometrical properties of the dataset and boundary whereas logistic regression is bsed on stastical appraoches.

In practice, both are good models to use.

Baysian (Statistical) Classification

In class we will be learning about Naive Bayes, and statistical classification.

Naive Bayes methods are a set of supervised learning algorithms based on applying Bayes’ theorem with the “naive” assumption of conditional independence between every pair of features given the value of the class variable. Bayes’ theorem states the following relationship, given class variable Y and dependent feature vector X1 through Xn.

Please implement a Naive Bayes Classifier on the pipelined data. For this model simply use the default parameters. Report out the number of mislabeled points that result (i.e., both the false positives and false negatives), along with the accuracy, precision, recall, F1 Score and Confusion Matrix. Refer to documentation on implementing a NB Classifier here (https://scikit-learn.org/stable/modules/naive_bayes.html)

In [32]:

```
from sklearn.naive_bayes import GaussianNB

gnb = GaussianNB()
model = gnb.fit(x_train, y_train)
```

In [33]:

```
y_predictions = model.predict(x_test)
y_true = list(y_test)
print(f"Accuracy: {str(accuracy_score(y_true, y_predictions))}")
print(f"Precision: {str(precision_score(y_true, y_predictions))}")
print(f"Recall: {str(recall_score(y_true, y_predictions))}")
print(f"F1 Score: {str (f1_score(y_true, y_predictions))}")
print(f"Confusion matrix: {confusion_matrix(y_true, y_predictions)}")
```

```
Accuracy: 0.5901639344262295
Precision: 0.5384615384615384
Recall: 0.9655172413793104
F1 Score: 0.691358024691358
Confusion matrix: [[ 8 24]
 [ 1 28]]
```

Discuss the observed results. What assumptions about our data are we making here and why might those be inacurate?

Here we have a model with great recall (95%) but poor precision which lowers the F1 score.

Gaussian Naive Bayes makes the assumption that there is independence between each predictor. Essentially the presence of one feature in a class should be unrelated to the presence of another one. For this dataset, this assumption may be inaccurate since certain variables may be related. In this case, age and resting blood pressure are related (increases as you get older).

Cross Validation and Model Selection

You've sampled a number of different classification techniques, leveraging clusters, linear classifiers, and Statistical Classifiers, as well as experimented with tweak different parameters to optimize perfiormance. Based on these experiments you should have settled on a particular model that performs most optimally on the chosen dataset.

Before our work is done though, we want to ensure that our results are not the result of the random sampling of our data we did with the Train-Test-Split. To ensure otherwise we will conduct a K-Fold Cross-Validation of our top two performing models, assess their cumulative performance across folds, and determine the best model for our particular data.

Select your top 2 performing models and run a K-Fold Cross Validation on both (use 10 folds). Report your best performing model.

In [57]:

```
import numpy as np
from sklearn.model_selection import KFold

# set up kfold
kf = KFold(n_splits=10, shuffle=True)
print(kf)

# prep data
X = processed_data
y = labels

# train logistic model
scores = []
for train_index, test_index in kf.split(X):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]
    logreg = LogisticRegression(solver ='liblinear', max_iter = 100, penalty='l1')
    logreg.fit(X_train, y_train)
    y_predictions = model.predict(X_test)
    y_true = list(y_test)
    f1 = f1_score(y_true, y_predictions)
    scores.append(f1)

print(f"Average F1 Logistic: {sum(scores)/len(scores)}")


# train SVC model
scores = []
for train_index, test_index in kf.split(X):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]
    svc = SVC(probability = True, kernel='linear')
    model = svc.fit(X_train, y_train)
    y_predictions = model.predict(X_test)
    y_true = list(y_test)
    f1 = f1_score(y_true, y_predictions)
    scores.append(f1)

print(f"Average F1 SVC: {sum(scores)/len(scores)}")
```

```
KFold(n_splits=10, random_state=None, shuffle=True)
Average F1 Logistic: 0.8382146885372693
Average F1 SVC: 0.8217486602714634
```

In K-fold cross validation, I partition my training dataset into 10 folds, and train 10 classifiers on a portion of the data and test on the remainder. I then compute the F1 score for each fold, and average them to produce the overall F1 score for a given model. This method allows us to train our models better for smaller datasets and eliminates irregularities.

Here my two best performing models were logistic regression and the SVC. This makes sense since it seems that the data was linearly seperable.

Overall the best model is the logistic regression model, which achieved 83.8% average F1 score.

In [ ]:

```
_
```