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 [61]:
```
#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


from matplotlib import pyplot
import itertools

%matplotlib inline

import random

random.seed(42)
```

In [62]:
```
# 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 [63]:
```
# 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 [64]:
```
hd = pd.read_csv("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 [65]:
```
hd.head()
```

Out[65]:

	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

In [66]:
```
hd.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
```

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?

The boolean data labeled as sick with values(True/False) will have to be binary numerically encoded as (0/1), as the alphabetical format is incompattible with our models.

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

In [67]:
```
sample_incomplete_rows = hd[hd.isnull().any(axis=1)]
sample_incomplete_rows.size
```

Out[67]: 0

Luckily, it seems that the hd dataframe has no null values, so imputation will not be neccesary.

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()

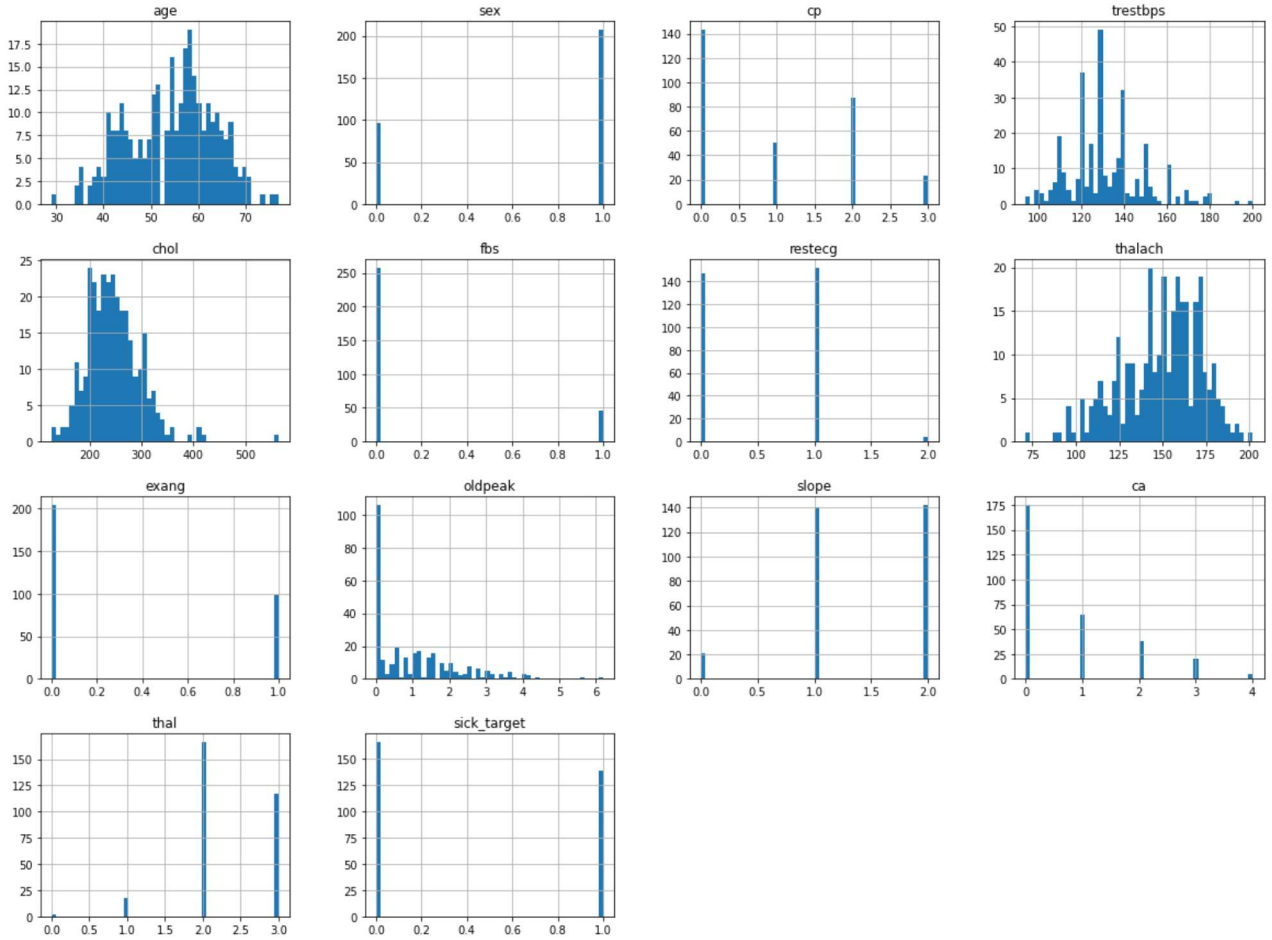
In [68]:
```
hd['sick_target'] = hd['sick'].astype(int)
hd = hd.drop(columns = 'sick')
hd.head()
```

Out[68]:

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

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 [69]:
```
hd.hist(bins=50, figsize=(20,15))
plt.show()
```



-binary features include sex, fbs, exang
the binary target is sick_target

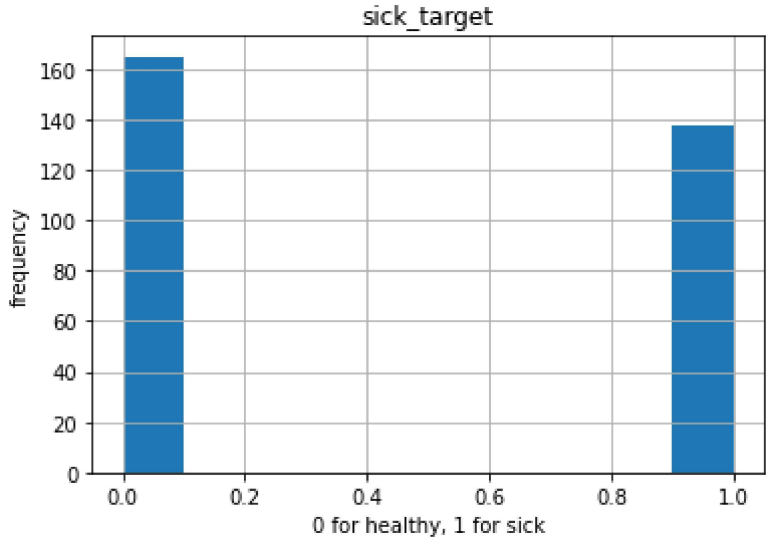
-limited selection features include cp, restecg, slope, ca, and thal

-gradient variables include age, trestbps, chol, thalach, and oldpeak

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 [71]:
```
hd['sick_target'].hist()
plt.title('sick_target')
plt.ylabel('frequency')
plt.xlabel('0 for healthy, 1 for sick')
```

Out[71]:
```
Text(0.5, 0, '0 for healthy, 1 for sick')
```



In [72]:
```
hd['sick_target'].value_counts()
```

Out[72]:
```
0    165
1    138
Name: sick_target, dtype: int64
```

In [73]:
```
hd['sick_target'].size == hd['sick_target'].value_counts()[0] + hd['sick_target'].value
```

Out[73]:
```
True
```

Using the informal unbalanced measure of 4:1, our target is balanced(165 healthy vs 138 sick)

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.

One main issue with arbitrary balancing a dataset is skewing the classification model to not accuratley reflect the population ratio of one target state to another. For example, if building a classifier to predict fradulent credit card transactions, it is important that the model results reflect that the population ratio of fradulent transactions to real ones is around 1:10000. If one arbitrarily balanced the dataset to say a 50:50 ratio, the model will heavily overclassify real-world transactions as fradulent as it was trained on a dataset with a much higher ratio of fradulent to real transactions than the real-world population has.

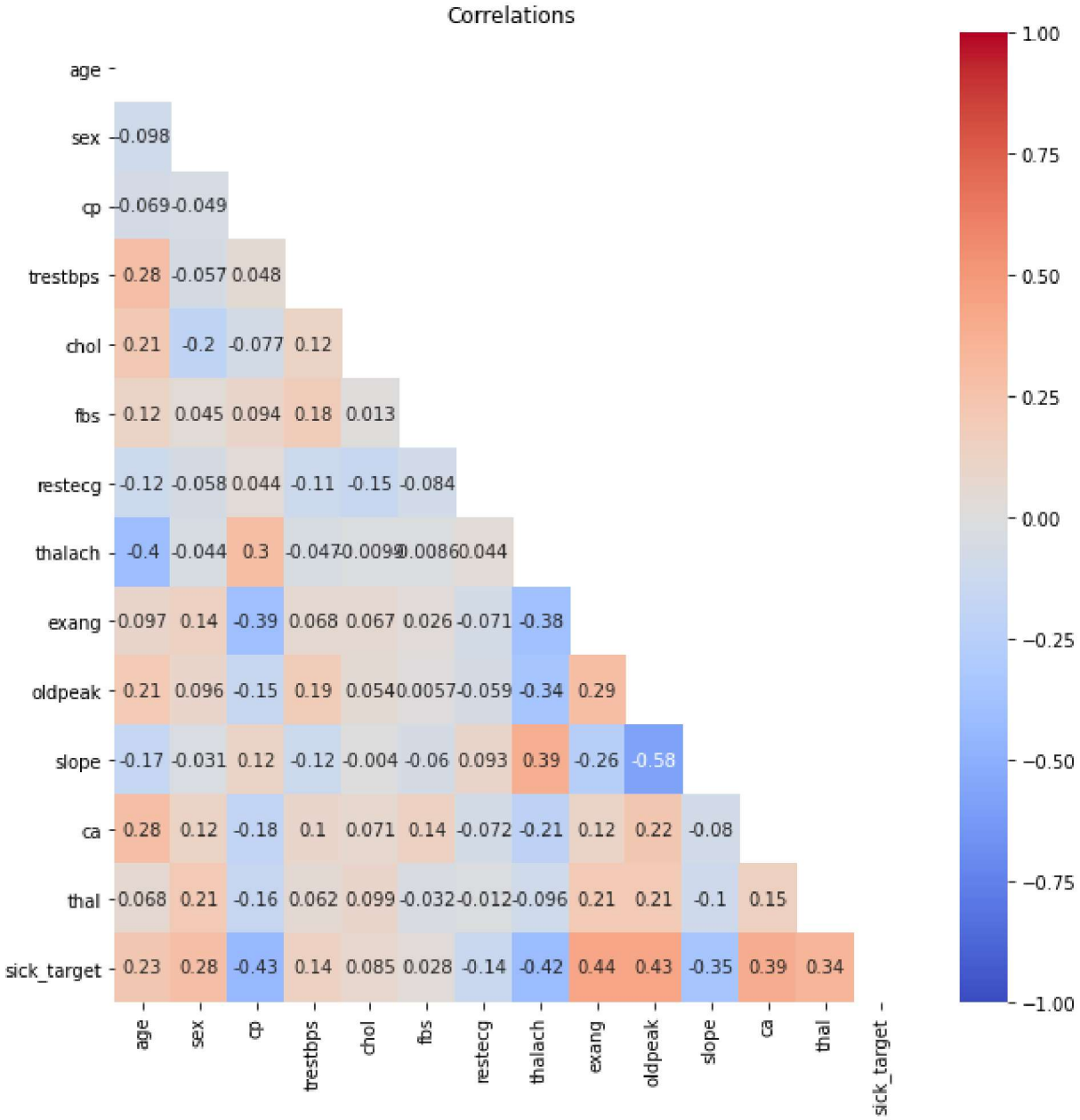
The method one uses to balance a dataset can also introduce new bias into the dataset...and hence the model. If one duplicates existing data to balance target data, the model will likely overfit in a correlated way to the ratio of duplicates added, defeating the purpose of balancing the dataset.

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 [74]:
```
#NOTE: code tutorial for diaganol heatmap taken from https://heartbeat.fritz.ai/seaborn
matrix = np.triu(hd.corr())
fig, ax = plt.subplots(figsize=(10,10))
plt.title("Correlations")
sns.heatmap(hd.corr(), annot = True, mask=matrix, vmin=-1, vmax=1, center= 0, cmap= 'co
```

Out[74]: <AxesSubplot:title={'center':'Correlations'}>



set cutoff to discuss correlation at +/-0.3, but there are a few features in the high 0.2's that will also be helpful

exang is highly correlated to sick_target(0.44): if exercise induces chest pain caused by reduced blood flow to the heart(angina), then that patient most likely has a weak heart or heart issues, of which the main one is heart disease

oldpeak is highly correlated to sick_target(0.43): if there is a large depression induced by exercise relative to rest, then the patient is likely not physically fit or has a weaker heart, increasing the risk of heart disease

ca is highly correlated to sick_target(0.39): I do not fully understand why a different number of major blood vessels gets colored by flouroscopy, but my best guess is that the more major blood vessels a patient has colored, the greater number of hospital visits related to blood flow the patient has had, which naturally correlates to a higher chance of heart disease.

thal is highly correlated to sick_target(0.34): a greater thalium stress test result indicates a greater chance of blood flow issues which is indicative of heart disease

---

cp is highly negatively correlated to sick_target(-0.43): increasing cp values represent chest pain more likely due to typical angina, which is a narrowing of the bloodvessels rather than a blockage or heart problem. If the cp value is low, it represents chest pain not caused by angina, leaving heart disease as a prevailing cause for the chest pain.

thalach is highly negatively correlated to sick_target(-0.42): a greater maximum heart rate is related to greater physical health which keeps the heart healthy and leads to a lower chance of heart disease

slope is highly negatively correlated to sick_target(-0.35): if the peak exercise ST segment is upsloping(max value of 2), this indicates a relatively physically healthy patient, putting them at less risk for heart disease.

The features that directly involve measurements of the heart tend to correlate more with sick_target, as these measurements are less effected by other organs/biological processes not relating to the heart. Many of such features are also correlated with each other, as heart issues heavily effect heart-related measurements. The features that increase in value corresponding to increased physical fitness negatively correlate to sick_target, as physically healtheir people exercise there heart more leading to a lower risk of heart disease and vice-versus.

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 [75]:
```
label = hd['sick_target']
hd = hd.drop(columns = 'sick_target')
hd.head()
```

Out[75]:

	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

In [76]:
```
label.head()
```

Out[76]:
```
0    0
1    0
2    0
3    0
4    0
Name: sick_target, dtype: int32
```

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 [77]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(hd, label, test_size=0.2)
```

In [82]:
```
f"X_train shape: {X_train.shape}"
```

Out[82]:
```
'X_train shape: (242, 13)'
```

In [83]:
```
f"y_train shape: {y_train.shape}"
```

Out[83]:
```
'y_train shape: (242,)'
```

In [84]:
```
f"X_test shape: {X_test.shape}"
```

Out[84]:
```
'X_test shape: (61, 13)'
```

In [85]:
```
f"y_test shape: {y_test.shape}"
```

Out[85]:
```
'y_test shape: (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 for details on implementation. Report on the accuracy of the resulting model.

In [86]:
```
# k-Nearest Neighbors algorithm
from sklearn.neighbors import KNeighborsClassifier
neigh = KNeighborsClassifier()
neigh.fit(X_train, y_train)

neigh_preds = neigh.predict(X_test)
```

In [89]:
```
# Report on model Accuracy
#Accuracy = (Tp+Tn)/(Tp+Fp+Tn+Fn)
f"Accuracy for knn with default n_neighbors=5: {metrics.accuracy_score(y_test, neigh_pr
```

Out[89]:
```
'Accuracy for knn with default n_neighbors=5: 0.5737704918032787'
```

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 [154…
```
#numerical: age, trestbps, chol, thalach, oldPeakST
#categorical that I will leave-as-is: sex, fbs, exang, slope, ca, thal
#categorical that I will one-hot-encode: cp, restecg

#UPDATE: decided to just one hot encode every cat variable

from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import OneHotEncoder
```

In [167…
```
hd_num = ['age', 'trestbps', 'chol', 'thalach', 'oldpeak']
```

In [168…
```
hd_cat_no = ['sex', 'fbs', 'exang', 'slope', 'ca', 'thal', 'cp', 'restecg']
```

In [169…
```
hd_cat_ohe = ['cp', 'restecg']
```

In [170…
```
num_pipeline = Pipeline([
    ('std_scaler', StandardScaler())
])

num_col = list(hd_num)

full_pipeline = ColumnTransformer([
    ("num",num_pipeline, num_col),
    ("cat_ohe", OneHotEncoder(), hd_cat_ohe),
    #("cat_no", hd_cat_no)
])
```

```
hd_prepared = full_pipeline.fit_transform(hd)
type(hd_prepared)
```

Out[170… numpy.ndarray

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 [173…
```
# k-Nearest Neighbors algorithm
X_train2, X_test2, y_train2, y_test2 = train_test_split(hd_prepared, label, test_size=0
neigh = KNeighborsClassifier()
neigh.fit(X_train2, y_train2)

neigh_preds2 = neigh.predict(X_test2)
```

In [174…
```
# Accuracy
f"Accuracy for knn with default n_neighbors=5: {metrics.accuracy_score(y_test2, neigh_p
```

Out[174… 'Accuracy for knn with default n_neighbors=5: 0.8032786885245902'

After pipelining by data by ohe the categorical variables and standard scaling the numerical vars, my accuracy went way up at 0.803 compared to 0.574 from the non-pipelined example. This indicates that by pipelining my data, my model was able to increase the rate at which it predicts Tp and Tn labels. This is in part due to removing some nonsensical implicit orderings in some of the categorical features.

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 [177…
```
for i in [1, 2, 3, 5, 7, 9, 10, 20, 50]:
    neigh = KNeighborsClassifier(n_neighbors=i)
    neigh.fit(X_train2, y_train2)

    neigh_preds2 = neigh.predict(X_test2)
    print(f"Accuracy for knn with n_neighbors={i}: {metrics.accuracy_score(y_test2, nei
```

```
Accuracy for knn with n_neighbors=1: 0.7049180327868853
Accuracy for knn with n_neighbors=2: 0.7377049180327869
Accuracy for knn with n_neighbors=3: 0.7540983606557377
Accuracy for knn with n_neighbors=5: 0.8032786885245902
Accuracy for knn with n_neighbors=7: 0.7868852459016393
Accuracy for knn with n_neighbors=9: 0.7377049180327869
Accuracy for knn with n_neighbors=10: 0.7704918032786885
Accuracy for knn with n_neighbors=20: 0.819672131147541
Accuracy for knn with n_neighbors=50: 0.819672131147541
```

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 for how to implement the model.

In [178…

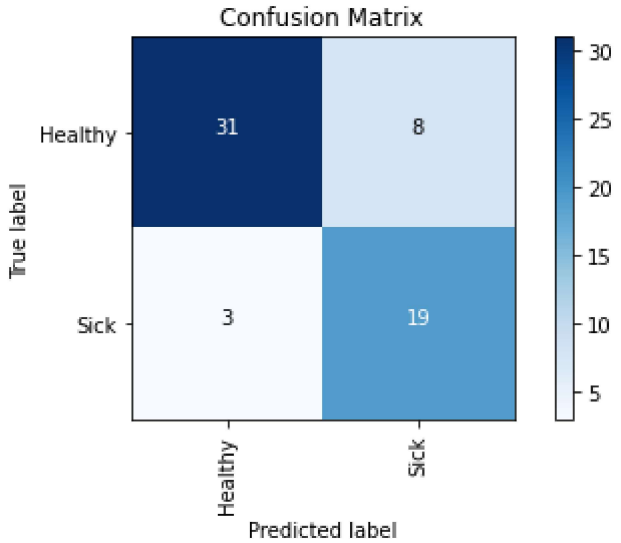
```
# Logistic Regression
from sklearn.linear_model import LogisticRegression

log_reg = LogisticRegression()
log_reg.fit(X_train2, y_train2)

log_reg_preds = log_reg.predict(X_test2)
```

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

In [179…

```
print(f"Accuracy: {metrics.accuracy_score(y_test2, log_reg_preds)}")
print(f"Precision: {metrics.precision_score(y_test2, log_reg_preds)}")
print(f"Recall: {metrics.recall_score(y_test2, log_reg_preds)}")
print(f"F1 Score: {metrics.f1_score(y_test2, log_reg_preds)}")

draw_confusion_matrix(y_test2, log_reg_preds, ['Healthy', 'Sick'])
```

```
Accuracy: 0.819672131147541
Precision: 0.7037037037037037
Recall: 0.8636363636363636
F1 Score: 0.7755102040816326
```



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 reports the fraction of predictions our model predicted correctly: (TP + TN)/(TP+FP+FN+TN) -It is fairly intuitive and useful when focusing on a model that makes correct predictions. It is a good measure to use when evaluating the model as a whole and focusing on both TN and TP. It is best used for balanced datasets, as very unbalanced datasets can obtain high accuracy simply by favoring the data value with the highest frequency. For example, if 0.01% of the population wins the lottery in their lifetime, a model can obtain very high accuracy by predicing that everyone will not win the lottery, leading to a useless model that has a high accuracy.

Precision accounts for some of the issues with accuracy by reporting the proportion of postive predictions that are actually correct: (TP) / (TP + FP) -It will help us narrow our model to only predict true values for the data that is actually labeled true. In the lottery example above, precision will give us a metric on of how many people we predicted to win the lottery, how many actually do. It is useful to prevent overpredicting entries as true. One scenario I would use it for is when trying to predict donors to give money to my foundation. I would like to eliminate as many FP's as I can to avoid wasting money and time of possible donors who will end up not giving me money.

Recall is the counterpart of precision, reporting on the fraction of correctly predicted positive observations vs all truly labeled positive entries: (TP)/(TP+FN) -It is a useful metric to hone in on correctly predicting all positive observations as positive. For example, in cancer predictions, one would need to focus on having as many TP and least FN predictions as possible, as FN's indicate someone has cancer but we predict they don't. So they are unable to get the help/treatment they need, leaving the cancer to spread and the patient to be lost.

An F1 score is an aggregator for recall and precision, allowing us an overall metric to evaluate performance across the board: 2*(Recall*Precision)/(Recall+Precision) -F1 scores give us a weighted average of both precision and recall. As such, it will better reflect imbalanced datasets by focusing on both recall and precision, preventing our model from overfocusing on either TP's or TN's while ignoring the other. So out of these 4 metrics, an F1 score would be the best for my lottery example.

The confusion matrix simple lays out our TP, FN, TN, FP in a clockwise order(in the provided implementation), allowing us an easy visual and accurate test numbers for these categories.

Graph the resulting ROC curve of the model

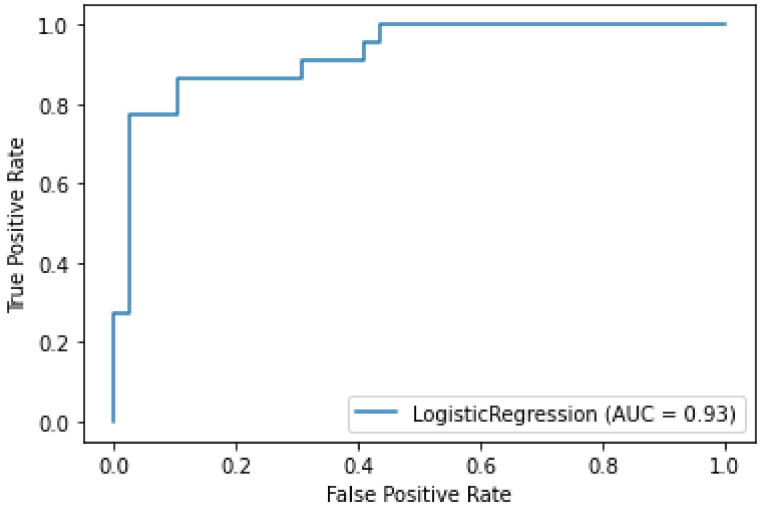
In [185…

```
print("SVM Model Performance Results:\n")

# score = log_reg.predict_proba(y_test2)
# fpr_svm, tpr_svm, thresholds = metrics.roc_curve(y_test2,log_reg_preds[:,1], pos_labe

# plt.figure(1)
# plt.plot(fpr_svm, tpr_svm, color='blue', lw=1)
# plt.xlabel('FPR')
# plt.ylabel('TPR')
# plt.show()
```

```
metrics.plot_roc_curve(log_reg, X_test2, y_test2)
```

```
SVM Model Performance Results:
```

Out[185… <sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x2122172f910>



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

An ROC curve models our TPR as the FPR increase at various threshold settings. The ideal scenario is for a TPR=1.0 to be achievable at a FPR=0.0, with a horizontal line stretching across the x-axis. My ROC curve starts off with TPR~0.3 with FPR=0, indicating our model can classify around 30% of the positive entries as positive without having a single FP. The ROC curve then goes up quickly to (0.1, 0.8) and then begins to step more in the +x direction, indicating at this stage we predict more FP's as we predict more TP's. It seems to plateua at (0.45, 1), indicating that for the model to predict all true positives as postive, it must incorrectly label about 45% of negative labels as positive. Overall, our roc curve indicates the model performs relatively well for 80% of positive labels, but begins to let in much more FP's to correctly classify the remaining 20% of positive labels as postive.

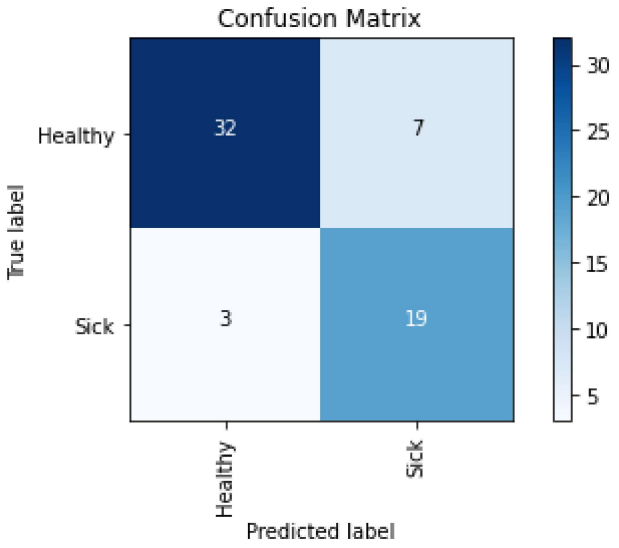
Let's tweak a few settings. First let's set your solver to 'sag', your max_iter= 10, and set penalty = 'none' and rerun your model. Report out the same metrics. Let's see how your results change!

In [189…

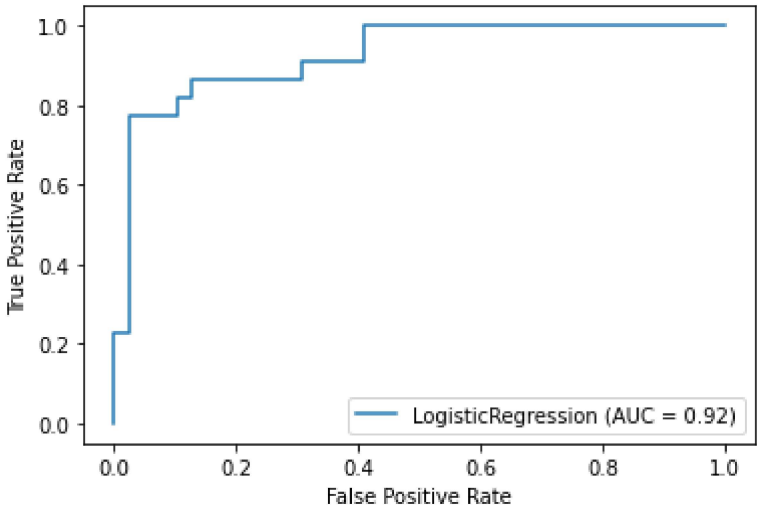
```
# Logistic Regression
log_reg = LogisticRegression(solver='sag', max_iter=10, penalty='none')
log_reg.fit(X_train2, y_train2)

log_reg_preds = log_reg.predict(X_test2)

print(f"Accuracy: {metrics.accuracy_score(y_test2, log_reg_preds)}")
print(f"Precision: {metrics.precision_score(y_test2, log_reg_preds)}")
print(f"Recall: {metrics.recall_score(y_test2, log_reg_preds)}")
print(f"F1 Score: {metrics.f1_score(y_test2, log_reg_preds)}")

draw_confusion_matrix(y_test2, log_reg_preds, ['Healthy', 'Sick'])
metrics.plot_roc_curve(log_reg, X_test2, y_test2)
```

```
Accuracy: 0.8360655737704918
Precision: 0.7307692307692307
```

```
Recall: 0.8636363636363636
F1 Score: 0.7916666666666666
C:\Users\dyera\anaconda3\lib\site-packages\sklearn\linear_model\_sag.py:329: ConvergenceWarning: The max_iter was reached which means the coef_ did not converge
  warnings.warn("The max_iter was reached which means "
```



Out[189… `<sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x21220a0de80>`



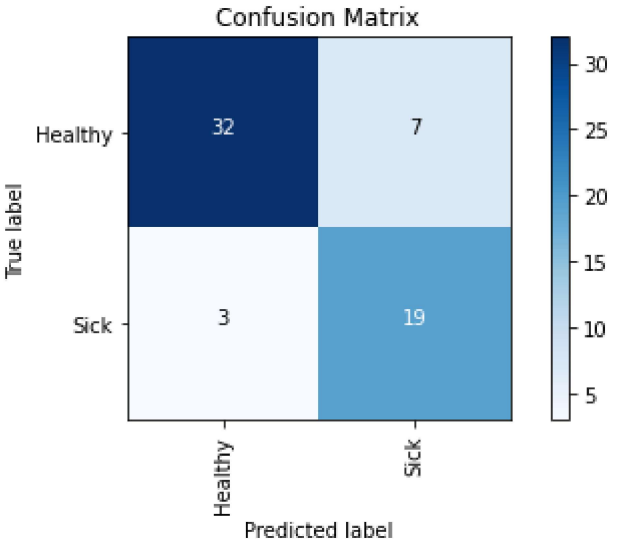
Did you notice that when you ran the previous model you got the following warning: "ConvergenceWarning: The max*iter was reached which means the coef* did not converge". Check the documentation and see if you can implement a fix for this problem, and again report your results.

In [190…

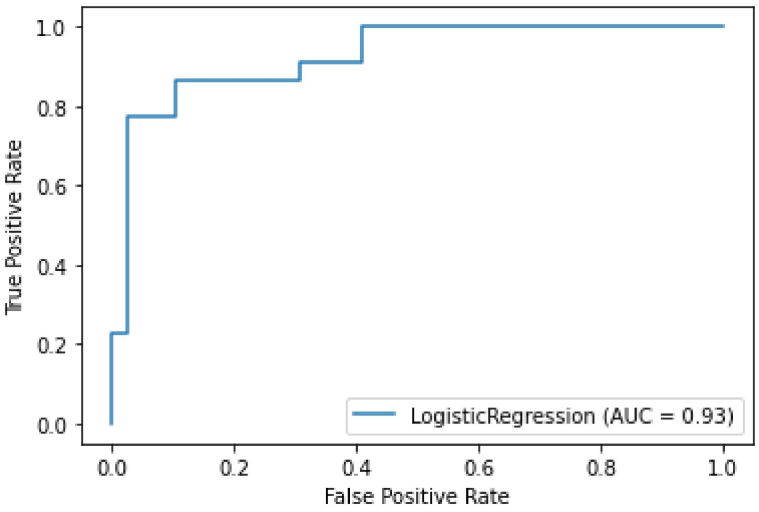
```
# Logistic Regression
log_reg = LogisticRegression(solver='sag', max_iter=100, penalty='none')
log_reg.fit(X_train2, y_train2)

log_reg_preds = log_reg.predict(X_test2)

print(f"Accuracy: {metrics.accuracy_score(y_test2, log_reg_preds)}")
print(f"Precision: {metrics.precision_score(y_test2, log_reg_preds)}")
print(f"Recall: {metrics.recall_score(y_test2, log_reg_preds)}")
print(f"F1 Score: {metrics.f1_score(y_test2, log_reg_preds)}")
```

```
draw_confusion_matrix(y_test2, log_reg_preds, ['Healthy', 'Sick'])
metrics.plot_roc_curve(log_reg, X_test2, y_test2)
```

```
Accuracy: 0.8360655737704918
Precision: 0.7307692307692307
Recall: 0.8636363636363636
F1 Score: 0.7916666666666666
```



Out[190… `<sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x21225587a30>`



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 ended up increasing the max itterations parameter by a factor of 10. Due to this, the logistic regression algorithm was able to run more gradient descent itterations to better select weights and bias to minimize the cost function. Doing so allowed for one more TP and one less FN, narrowly increasing the majority of my metrics. The no penalty indicates no regularization occured. My AUC increased from the previous example, but remained the same as my default example.

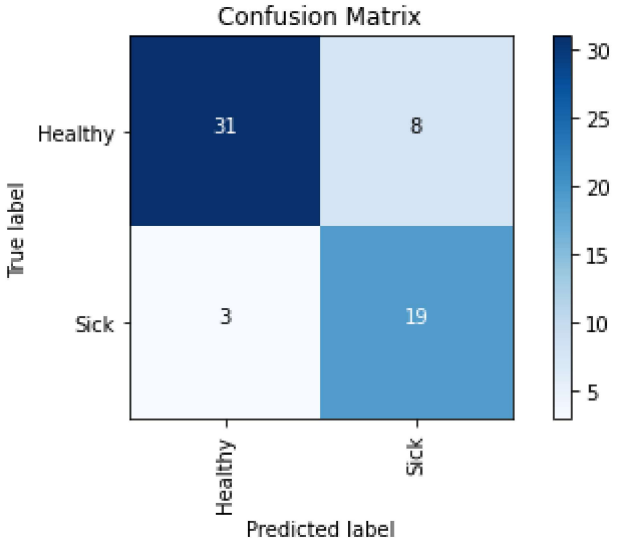
Rerun your logistic classifier, but modify the penalty = 'l1', solver='liblinear' and again report the results.

In [191…

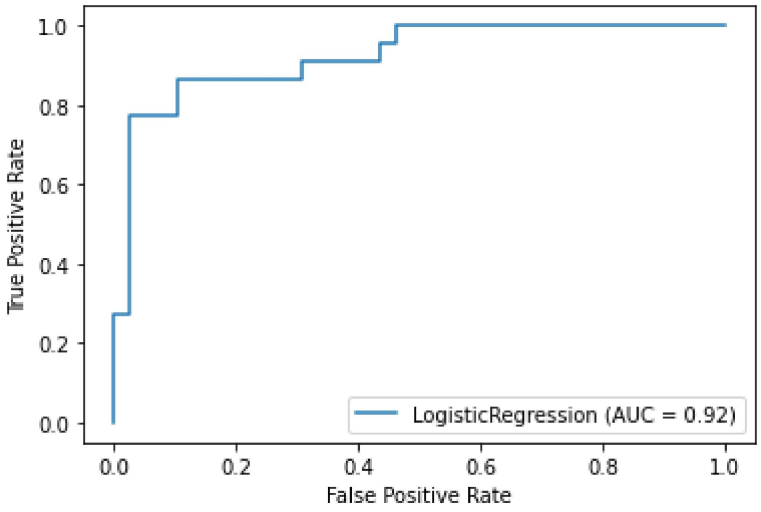
```
# Logistic Regression
```

```
log_reg = LogisticRegression(solver='liblinear', max_iter=100, penalty='l1')
log_reg.fit(X_train2, y_train2)

log_reg_preds = log_reg.predict(X_test2)

print(f"Accuracy: {metrics.accuracy_score(y_test2, log_reg_preds)}")
print(f"Precision: {metrics.precision_score(y_test2, log_reg_preds)}")
print(f"Recall: {metrics.recall_score(y_test2, log_reg_preds)}")
print(f"F1 Score: {metrics.f1_score(y_test2, log_reg_preds)}")

draw_confusion_matrix(y_test2, log_reg_preds, ['Healthy', 'Sick'])
metrics.plot_roc_curve(log_reg, X_test2, y_test2)
```

```
Accuracy: 0.819672131147541
Precision: 0.7037037037037037
Recall: 0.8636363636363636
F1 Score: 0.7755102040816326
```



Out[191… <sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x21224ee3d00>



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

According to the documentation, the sag solver is better for larger datasets and the liblinear works great for smaller ones. The sag solver stands for Stochastic Average Gradient Descent and is a variation of gradient descent and incremental aggregated gradient approaches that uses a random

sample of previous gradient values. As such, it does not fully implement Stochastic Gradient Descent, allowing for a speedup on large datasets at the cost of increasingly less accuracy for smaller datasets. On the other hand, liblinear stands for Library for Large Linear Classification. It uses a coordinate descent algorithm. Coordinate descent is based on minimizing a multivariate function by solving univariate optimization problems in a loop. In other words, it moves toward the minimum in one direction at a time, meaning a slower run time for better results. As out test dataset was small, liblinear was the obvious choice.

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 term is responsible for setting the way the logistic regression model computes regularization. Regularization reduces/shrinks the coefficients in the resulting regression, lowering variance and preventig overfitting. L1 indicates lasso regression and L2 indicates ridge regression. Lasso regression works by adding a penalty equal to the absolute value of the magnitude of coefficients. This type of regularization can result in sparse models with few coefficients. Some coefficients can become zero and are eliminated from the model. This is an example of shrinkage.

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 for how to implement a model. For this implementation you can simply use the default settings, but set probability = True.

In [193…

```
# SVM
from sklearn.svm import SVC
svm = SVC(probability=True)
svm.fit(X_train2, y_train2)
svm_preds = svm.predict(X_test2)
```

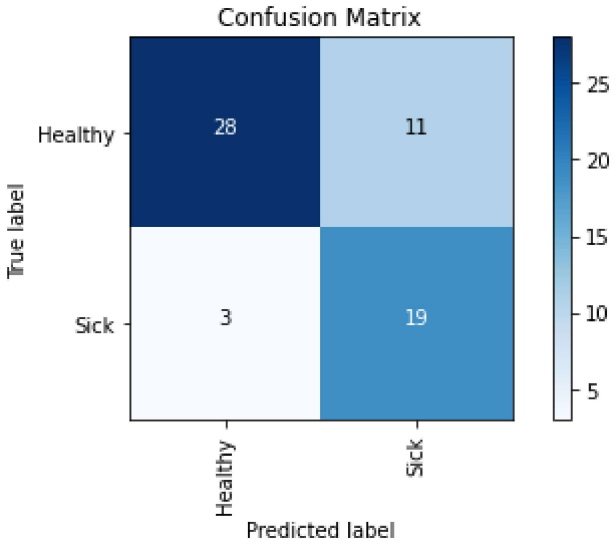
Report the accuracy, precision, recall, F1 Score, and confusion matrix and ROC Curve of the resulting model.

In [194…

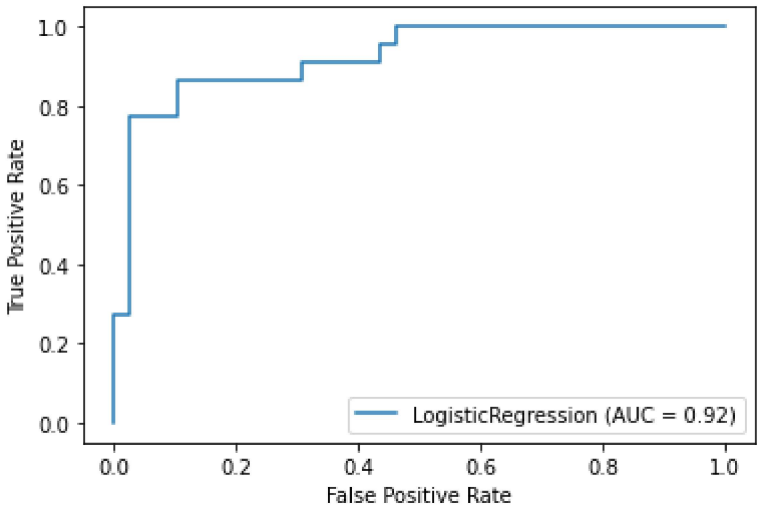
```
log_reg_preds = svm_preds #I'm to lazy
print(f"Accuracy: {metrics.accuracy_score(y_test2, log_reg_preds)}")
print(f"Precision: {metrics.precision_score(y_test2, log_reg_preds)}")
print(f"Recall: {metrics.recall_score(y_test2, log_reg_preds)}")
print(f"F1 Score: {metrics.f1_score(y_test2, log_reg_preds)}")

draw_confusion_matrix(y_test2, log_reg_preds, ['Healthy', 'Sick'])
metrics.plot_roc_curve(log_reg, X_test2, y_test2)
```

```
Accuracy: 0.7704918032786885
Precision: 0.6333333333333333
```

```
Recall: 0.8636363636363636
F1 Score: 0.7307692307692307
```



Out[194… `<sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x212254bce50>`



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 [196…

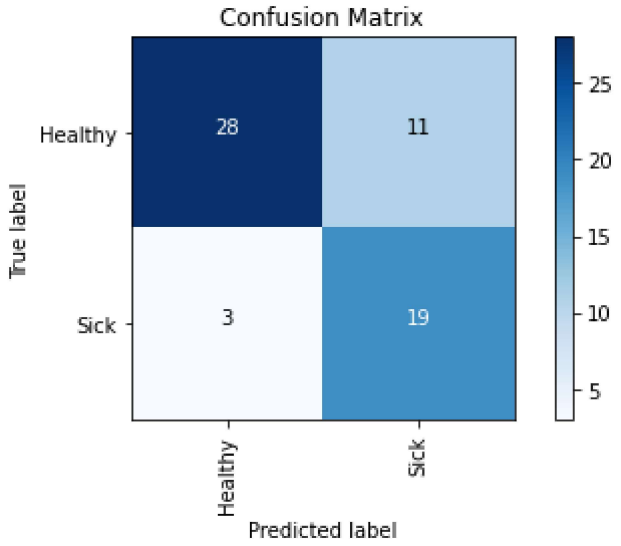
```
# SVM
svm = SVC(probability=True, kernel='linear')
svm.fit(X_train2, y_train2)
svm_preds = svm.predict(X_test2)
```

In [197…

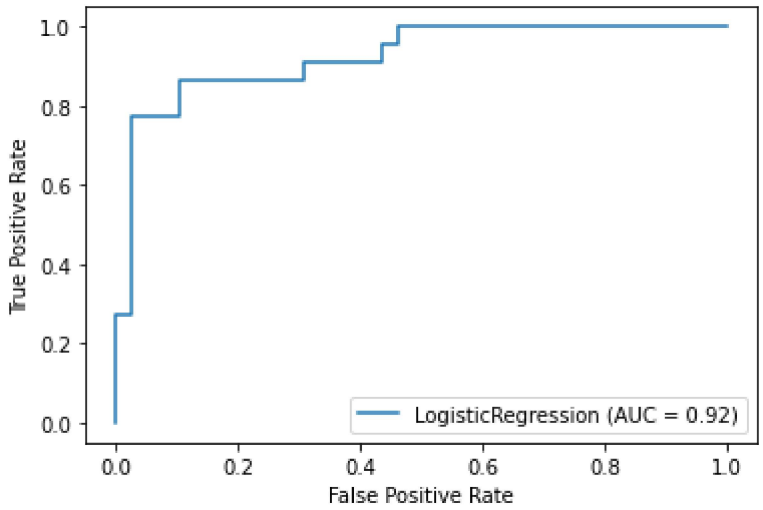
```
log_reg_preds = svm_preds #I'm to Lazy
print(f"Accuracy: {metrics.accuracy_score(y_test2, log_reg_preds)}")
print(f"Precision: {metrics.precision_score(y_test2, log_reg_preds)}")
print(f"Recall: {metrics.recall_score(y_test2, log_reg_preds)}")
print(f"F1 Score: {metrics.f1_score(y_test2, log_reg_preds)}")

draw_confusion_matrix(y_test2, log_reg_preds, ['Healthy', 'Sick'])
metrics.plot_roc_curve(log_reg, X_test2, y_test2)
```

```
Accuracy: 0.7704918032786885
Precision: 0.6333333333333333
```

```
Recall: 0.8636363636363636
F1 Score: 0.7307692307692307
```



Out[197… `<sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x212254d8d00>`



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.

My metrics have not changed, most likely due to the limited sample size I have. However, by setting the kernel to linear, I have adjusted the training method of the model to create more linear decision boundaries, rather than radial boundaries, as the default kernel='rbf' computes in polar coordinates.

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?

While both logistic regression and linear SVM classify data points using a linear decision boundar, logistic regression finds this boundary through a probabalistic model and SVM determines this through a geometric approach. To do this, logistic regression finds its boundary through itterations on the sigmoid function and minimizing a loss function, while SVM finds a decsion boundary that has the same distance from all boundary points on both sides.

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

In [198…

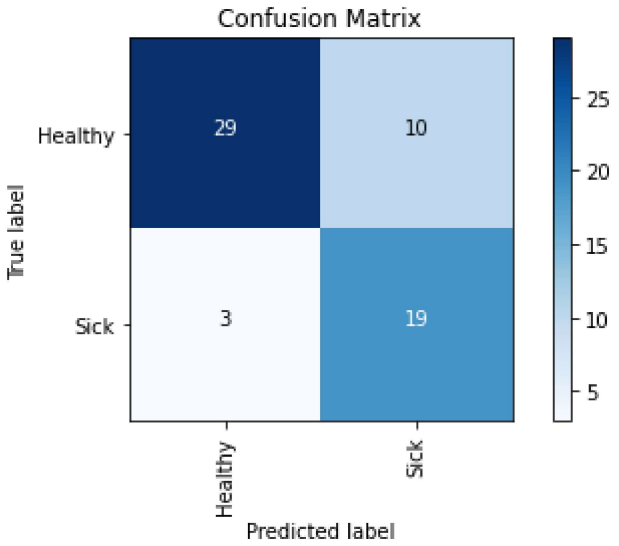
```
#copied from documentation

from sklearn.naive_bayes import GaussianNB
gnb = GaussianNB()
y_pred = gnb.fit(X_train2, y_train2).predict(X_test2)
print("Number of mislabeled points out of a total %d points : %d" % (X_test2.shape[0],_
```

```
Number of mislabeled points out of a total 61 points : 13
```

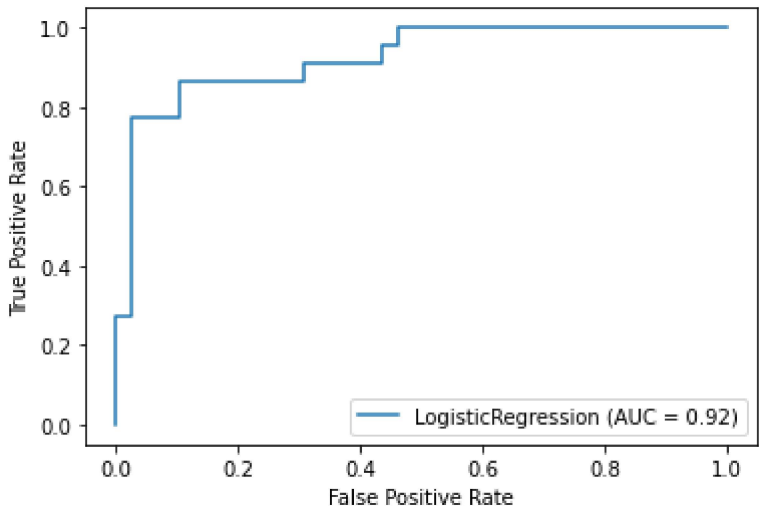
In [199…

```
log_reg_preds = y_pred #I'm to lazy pt9999999999
print(f"Accuracy: {metrics.accuracy_score(y_test2, log_reg_preds)}")
print(f"Precision: {metrics.precision_score(y_test2, log_reg_preds)}")
print(f"Recall: {metrics.recall_score(y_test2, log_reg_preds)}")
print(f"F1 Score: {metrics.f1_score(y_test2, log_reg_preds)}")

draw_confusion_matrix(y_test2, log_reg_preds, ['Healthy', 'Sick'])
metrics.plot_roc_curve(log_reg, X_test2, y_test2)
```

```
Accuracy: 0.7868852459016393
Precision: 0.6551724137931034
Recall: 0.8636363636363636
F1 Score: 0.7450980392156864
```



Out[199… <sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x212263a9ca0>



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

Once again, there is little variation of my metrics from previous models due to a limited sample size and homogenous random seed throughout. The major assumption when using a naive bayes classifier is that our predictors are independent. In this case, it is an innacurate assumption, as many of our predictors are highly correlated with each other. You can see an exact breakdown of this in the correlation triangle graph from earlier. Some exact examples are slope and old peak being negatively correlated with a value of -0.58. In general, it makes sense that multiple blood vessel and heart measuremnts taken from the same patient would be dependent on each other.

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 [203…
```
from sklearn.model_selection import KFold
from sklearn import model_selection

#kf = model_selection.KFold(n_splits=10, shuffle=True, random_state=42)
```

In [211…
```
# Logistic Regression
kfold = model_selection.KFold(n_splits=10, random_state=42, shuffle=True)
```

```
log_reg_k = LogisticRegression(solver='sag', max_iter=2000, penalty='none')

svm_k = SVC(probability=True, kernel='linear')

log_results_k = model_selection.cross_val_score(log_reg_k, hd_prepared, label, cv=kfold

svm_results_k = model_selection.cross_val_score(svm_k, hd_prepared, label, cv=kfold)

print(f"log_reg_accuracy: {log_results_k.mean()*100}")
print(f"svm_reg_accuracy: {svm_results_k.mean()*100}")
```

```
log_reg_accuracy: 76.19354838709678
svm_reg_accuracy: 75.21505376344085
```

In [213…

```
log_results_k
```

Out[213…

```
array([0.77419355, 0.90322581, 0.74193548, 0.63333333, 0.76666667,
       0.73333333, 0.6       , 0.83333333, 0.8       , 0.83333333])
```

My logistic regression using a sag solver ended up performing the best. I tried with a liblinear solver but got an accuracy around 75. Both returned a smaller accuracy using kfold, indicating i was previously lucky with my split.