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 [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
```

```
from matplotlib import pyplot
import itertools

%matplotlib inline

import random
__
random.seed(42)
```

In [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)
```

In [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()
```

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 [4]:

```
df = 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 [5]:

```
df.head()
```

Out[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

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 [6]:

```
df.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
```

According to the colimns above, we can see that only sick is boolean which is different from other data type such as int and float. Hence, we need to modify the values of "sick" column with one-hot-encoding or any similar way to make it to be numeric.

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

In [7]:

```
df.isnull().values.any()
```

Out[7]:

```
False
```

We are not dealing with any null values since all columns are non-null in the dataframe and we checked by isnull().values.any() which gives us the false result.

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 [8]:

```
df["sick"] = df["sick"].astype(int)
df["sick"]
```

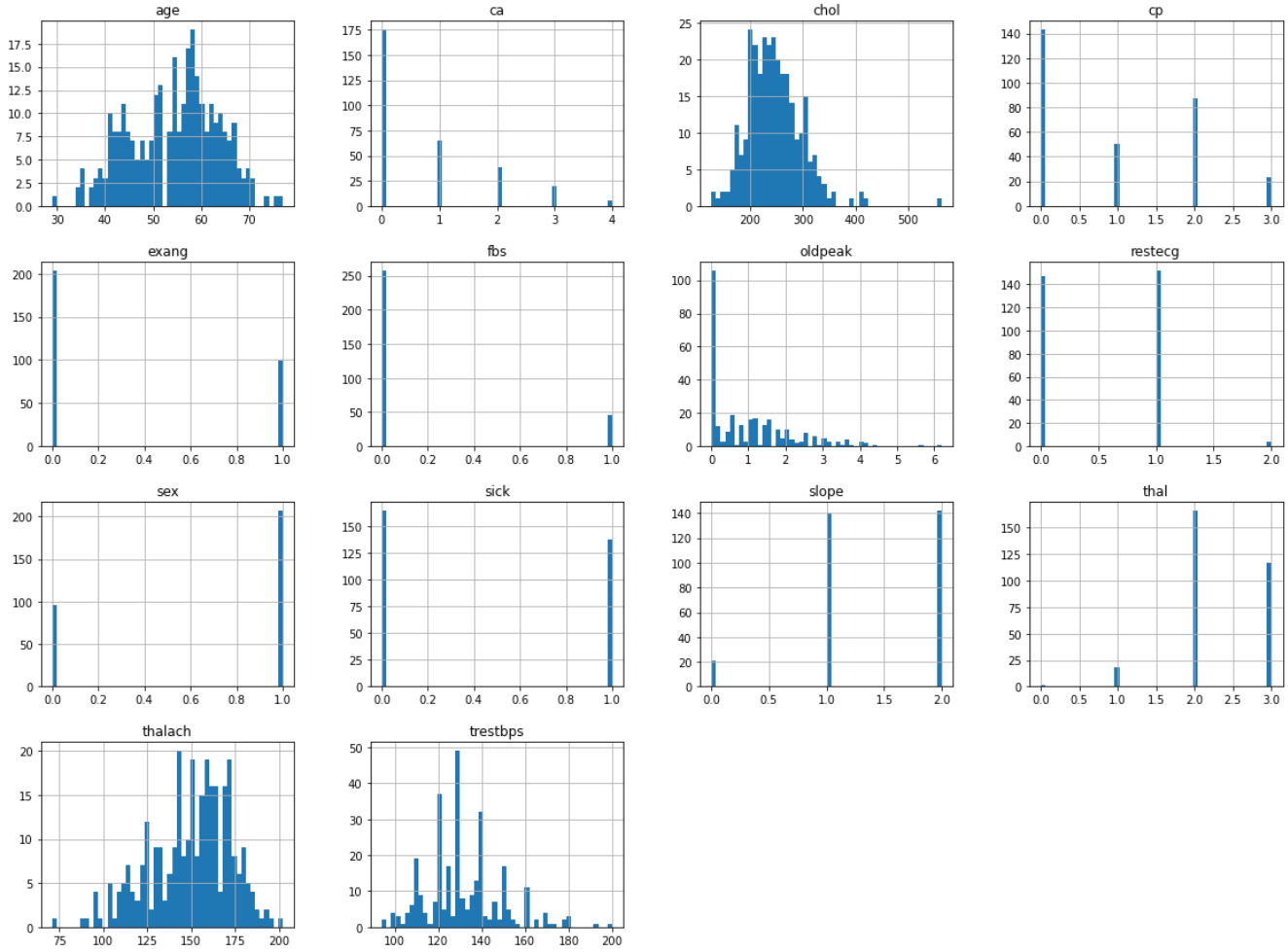
Out[8]:

```
0      0
1      0
2      0
3      0
4      0
      ..
298    1
299    1
300    1
301    1
302    1
Name: sick, Length: 303, dtype: int64
```

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 [9]:

```
df.hist(bins=50, figsize=(20,15))
plt.show()
```

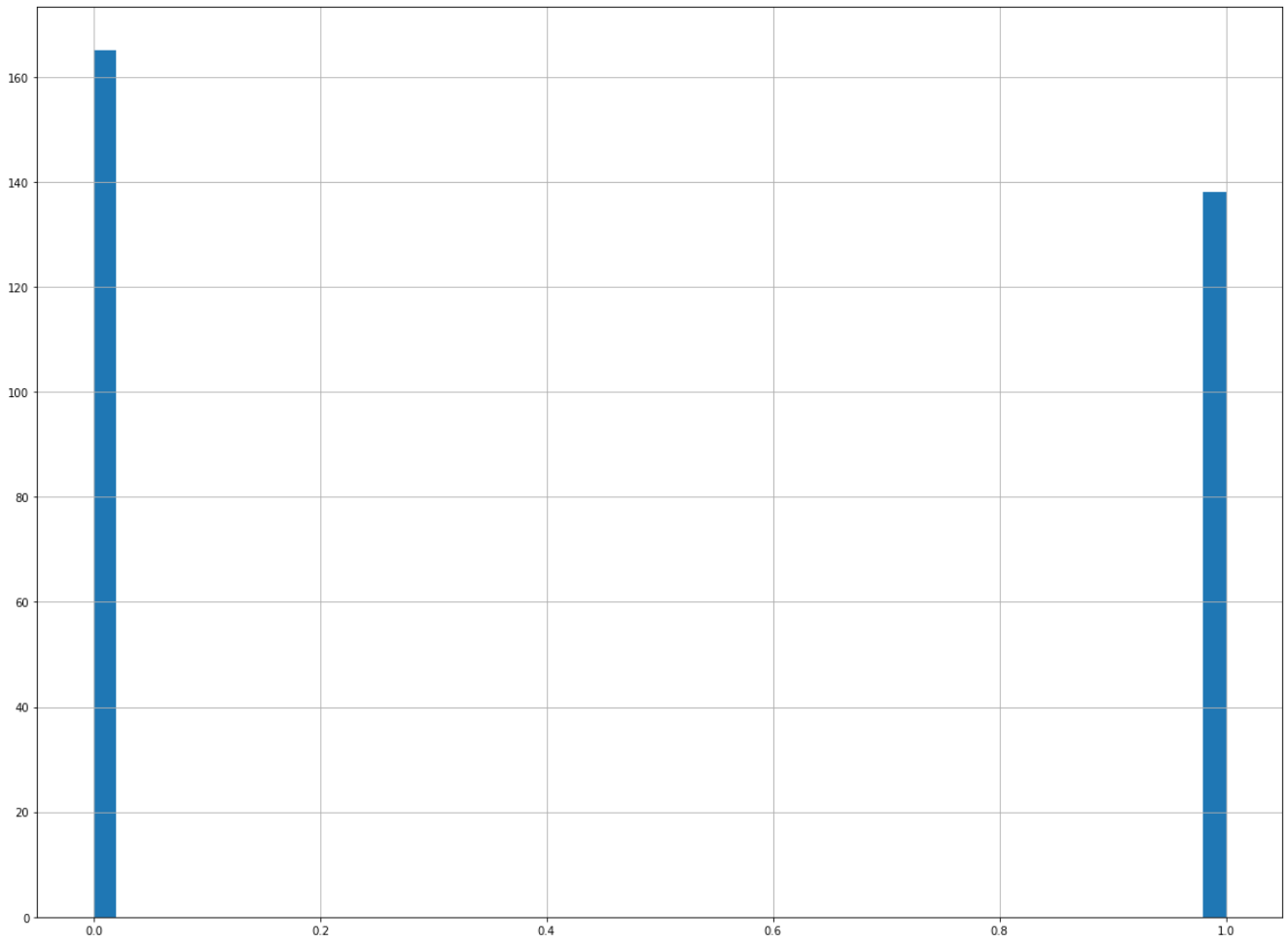


binary: exang, fbs, sex, sick limited selection: ca, cp restecg, slope, thal gradient: age, trestbps, chol, thalach, oldpeak

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 [10]:

```
df["sick"].hist(bins=50, figsize=(20,15))
plt.show()
df['sick'].value_counts()
```



Out[10]:

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

There are 165 unsick and 138 sick people in the data frame, it's a relatively balanced dataset.

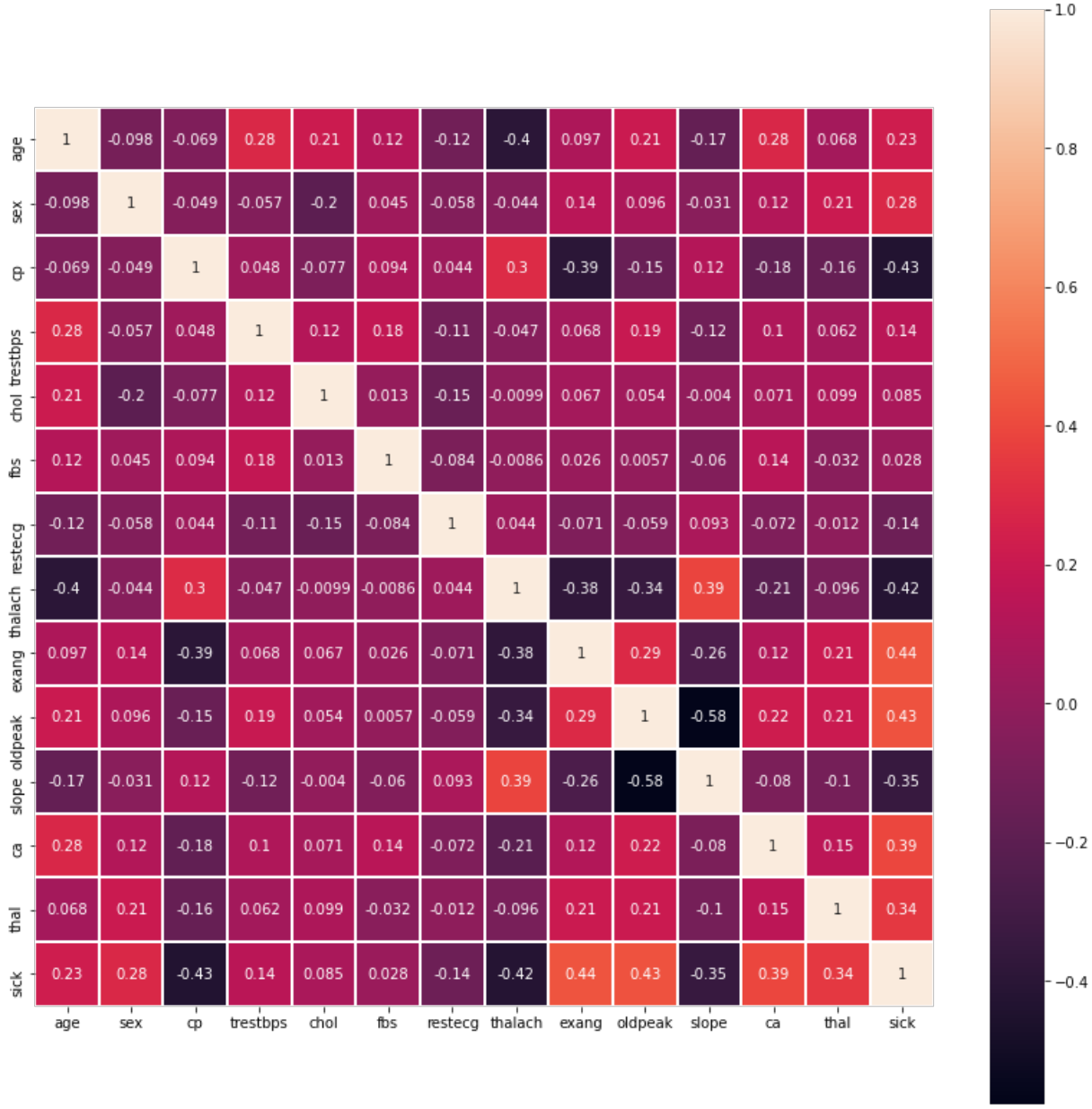
Balanced datasets are important to ensure that classifiers train adequately and don't overfit, however arbitrary balancing of a dataset might introduce its own issues. Discuss some of the problems that might arise by artificially balancing a dataset.

To balance the data set, we might down select the data, which will cause a lot of problems, for example, for the binary data type, if the quantity of 1s is much higher than 2s say like 90% vs 10%, because of the small training set, the classifier can mistakenly get as high as 90% accuracy.

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 [11]:

```
corr_matrix = df.corr()
corr_matrix
f, ax = plt.subplots(figsize=(14, 14))
ax = sns.heatmap(corr_matrix, annot=True, square=True, linewidths=1)
```



The correlation can be visualized by colors on the above heatmap that as the color getting brigher, it's more correlated. For example, sick is strongly correlated with exang, old peak, ca, and thal with the value 0.44, 0.43, 0.39 and 0.34. Exang is the exercise induced angina, intuitively, I think the strong correlation between heart disease and exang is because the typical angina pectoris is induced by exercise and lasts for a few minutes. It is located behind the sternum, and may have radiating pain in the upper limbs, teeth and other parts. If the chest pain lasts for more than 30 minutes, it is necessary to suspect myocardial infarction.

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 [12]:

```
df_y = df["sick"].copy()
df_X = df.drop("sick", axis=1)
```

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 [13]:

```
X_train_raw, X_test_raw, y_train_raw, y_test_raw = train_test_split(df_X, df_y, test_size = 0.2, ra
ndom_state = 42)
print("df_X shape: ", df_X.shape)
print("df_y shape: ", df_y.shape)

print("X_train_raw shape: ", X_train_raw.shape)
print("X_test_raw shape: ", X_test_raw.shape)
print("y_train_raw shape: ", y_train_raw.shape)
print("y_test_raw shape: ", y_test_raw.shape)
```

```
df_X shape:  (303, 13)
df_y shape:  (303,)
X_train_raw shape:  (242, 13)
X_test_raw shape:  (61, 13)
y_train_raw shape:  (242,)
y_test_raw 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 [14]:

```
# k-Nearest Neighbors algorithm
from sklearn.neighbors import KNeighborsClassifier

neigh = KNeighborsClassifier()
neigh.fit(X_train_raw, y_train_raw)
y_pred = neigh.predict(X_test_raw)
```

In [15]:

```
# Report on model Accuracy
from sklearn.metrics import accuracy_score
accuracy_score(y_test_raw, y_pred, normalize=True)
```

Out[15]:

```
0.6885245901639344
```

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 [16]:

```
df_X.head()
```

Out[16]:

	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

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
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 [17]:

```
# choose the categoricals and drop them
categoricals = ["sex", "cp", "fbs", "restecg", "exang", "slope", "ca", "thal"]
numericals = list(df_X.drop(categoricals, axis=1))
print(numericals)

# binary: exang, fbs, sex, sick
# limited selection: ca, cp restecg, slope, thal
```

```
['age', 'trestbps', 'chol', 'thalach', 'oldpeak']
```

In [18]:

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

full_pipeline = ColumnTransformer([
    ("num", StandardScaler(), numericals),
    ("cat", OneHotEncoder(sparse=False), categoricals)
])
df_X_prep = full_pipeline.fit_transform(df_X)
df_X_prep
```

Out[18]:

```
array([[ 0.9521966 ,  0.76395577, -0.25633371, ...,  1.        ,
         0.        ,  0.        ],
       [-1.91531289, -0.09273778,  0.07219949, ...,  0.        ,
         1.        ,  0.        ],
       [-1.47415758, -0.09273778, -0.81677269, ...,  0.        ,
         1.        ,  0.        ],
       ...,
       [ 1.50364073,  0.70684287, -1.029353  , ...,  0.        ,
         0.        ,  1.        ],
       [ 0.29046364, -0.09273778, -2.2275329 , ...,  0.        ,
         0.        ,  1.        ],
       [ 0.29046364, -0.09273778, -0.19835726, ...,  0.        ,
         1.        ,  0.        ]])
```

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
X_train, X_test, y_train, y_test = train_test_split(df_X_prep, df_y, test_size = 0.2, random_state
= 42)
neigh = KNeighborsClassifier()
neigh.fit(X_train, y_train)
y_pred = neigh.predict(X_test)
```

In [20]:

```
# Accuracy
accuracy_score(y_test, y_pred)
```

Out[20]:

```
0.9016393442622951
```

I got 90.16% versus 68.85% accuracy scores, the current one is higher than the raw data. Firstly, I think the reason is because of the one hot encoder. Secondly, the standard scalar is playing a important role, instead of keep using the original numerical data that varies in a relatively large range in our raw data. In this case, we used standarzation and reduce the bias and side effects on the model training. Therefore, we get a more accurate prediction.

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_values
neighbors = [1, 2, 3, 5, 7, 9, 10, 20, 50]
for n in neighbors:
    neigh = KNeighborsClassifier(n_neighbors=n)
    neigh.fit(X_train, y_train)
    y_pred = neigh.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    print(f"{n} neighbors has accuracy score: {accuracy}")
```

```
1 neighbors has accuracy score: 0.8032786885245902
2 neighbors has accuracy score: 0.8688524590163934
3 neighbors has accuracy score: 0.8688524590163934
5 neighbors has accuracy score: 0.9016393442622951
7 neighbors has accuracy score: 0.9016393442622951
9 neighbors has accuracy score: 0.8852459016393442
10 neighbors has accuracy score: 0.8852459016393442
20 neighbors has accuracy score: 0.9016393442622951
50 neighbors has accuracy score: 0.8852459016393442
```

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 [22]:

```
# Logistic Regression
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression
clf = LogisticRegression(random_state=0).fit(X_train, y_train)
y_pred = clf.predict(X_test)
y_pred
```

Out[22]:

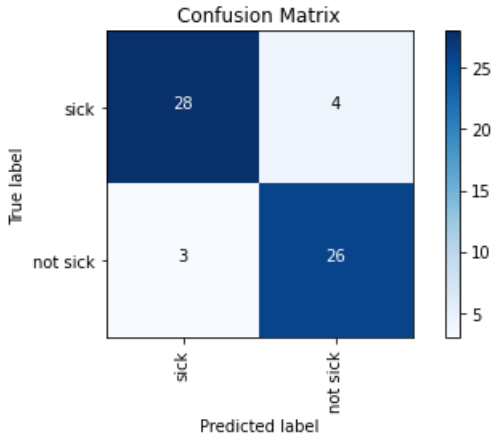
```
array([1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1,
       1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1])
```

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

In [23]:

```
from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score
def reportFourMetrics(y, y_pred):
    print("Accuracy: ", accuracy_score(y, y_pred))
    print("Precision: ", precision_score(y, y_pred))
    print("recall: ", recall_score(y, y_pred))
    print("F1 Score: ", f1_score(y, y_pred))

reportFourMetrics(y_test, y_pred)
draw_confusion_matrix(y_test, y_pred, ["sick", "not sick"])
```

```
Accuracy:  0.8852459016393442
Precision:  0.8666666666666667
recall:  0.896551724137931
F1 Score:  0.8813559322033899
```



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 most intuitive performance measure and it is simply a ratio of correctly predicted observation to the total observations. Precision is the ratio of correctly predicted positive observations to the total predicted positive observations. Recall is the ratio of correctly predicted positive observations to the all observations in actual class. F1 Score is the weighted average of Precision and Recall. Therefore, this score takes both false positives and false negatives into account. Generally, F1 is the more useful than accuracy if we have uneven class distribution. However, the situation will be reversed if false positives and false negatives have similar cost. And if the cost are very different, we need to consider both the precison and recall. By evaluating the performance of differing models differently based on these factor, we will have a more comprehensive view of the performance towards the different situations.

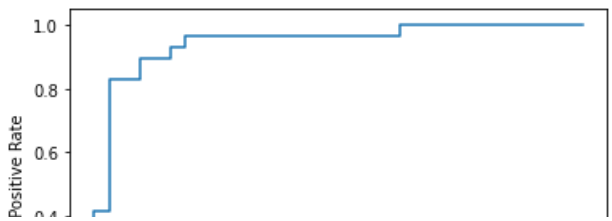
Graph the resulting ROC curve of the model

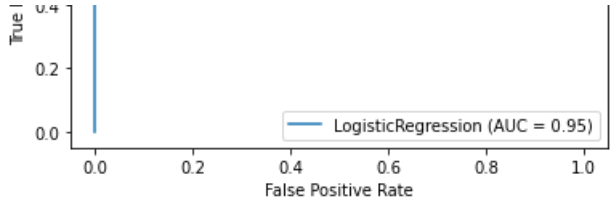
In [24]:

```
metrics.plot_roc_curve(clf, X_test, y_test)
```

Out[24]:

```
<sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x7fc993897280>
```





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

AUC - ROC curve is a performance measurement for the classification problems at various threshold settings. It is a probability curve with AUC which is used to represent the degree of separability. For the curve above, it tells us the capablility of the model distinguishing between classes is 0.95(95%). As the AUC getting higher, the model is better distinguish sick and non sick people. In this cse, is 95% chance to get the correct prediction.

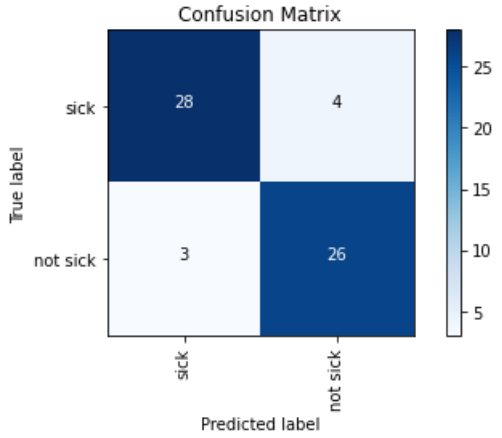
Let's tweak a few settings. First let's set your solver to 'sag', your max_iter= 10, and set penalty = 'none' and rerun your model. Report out the same metrics. Let's see how your results change!

In [25]:

```
# Logistic Regression
clf = LogisticRegression(solver='sag', penalty='none', max_iter=10).fit(X_train, y_train)
y_pred = clf.predict(X_test)
reportFourMetrics(y_test, y_pred)
draw_confusion_matrix(y_test, y_pred, ["sick", "not sick"])
```

```
/Users/carinaxiong/opt/anaconda3/lib/python3.8/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 "
```

```
Accuracy:  0.8852459016393442
Precision:  0.8666666666666667
recall:  0.896551724137931
F1 Score:  0.8813559322033899
```

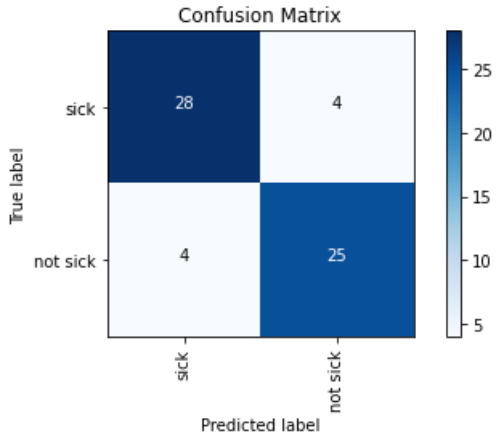


Did you notice that when you ran the previous model you got the following warning: "ConvergenceWarning: The max*iter was reached which means the coef* did not converge". Check the documentation and see if you can implement a fix for this problem, and again report your results.

In [26]:

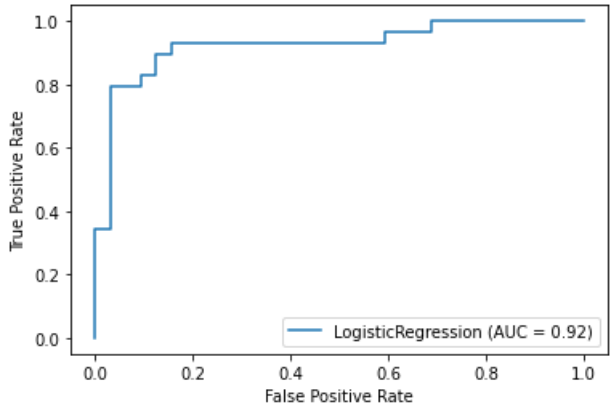
```
# Logistic Regression
clf = LogisticRegression(solver='sag', penalty='none', max_iter=50000).fit(X_train, y_train)
y_pred = clf.predict(X_test)
reportFourMetrics(y_test, y_pred)
draw_confusion_matrix(y_test, y_pred, ["sick", "not sick"])
metrics.plot_roc_curve(clf, X_test, y_test)
```

```
Accuracy:  0.8688524590163934
Precision:  0.8620689655172413
recall:  0.8620689655172413
F1 Score:  0.8620689655172413
```



Out[26]:

```
<sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x7fc9929643d0>
```



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?

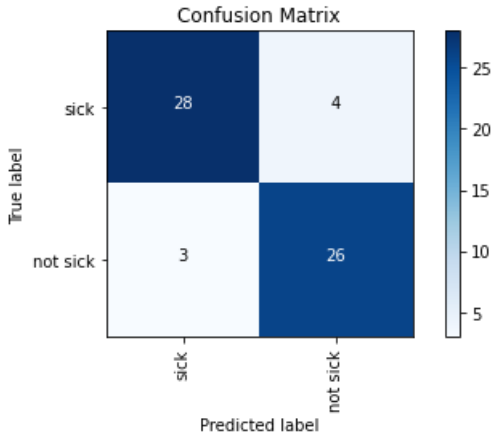
Since the former model is not converged, I make the max iteration from 10 to 50000. If the iteration value is too small, the model has no enough rounds to get converged. The coef_ converge means that the model's training error is not descending anymore as the iteration increase, hence, we get the best prediction value. However, in this case, our AUC dropped from 0.95 to 0.92 which is not what we expected which might be caused by the penalty setting.

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

In [27]:

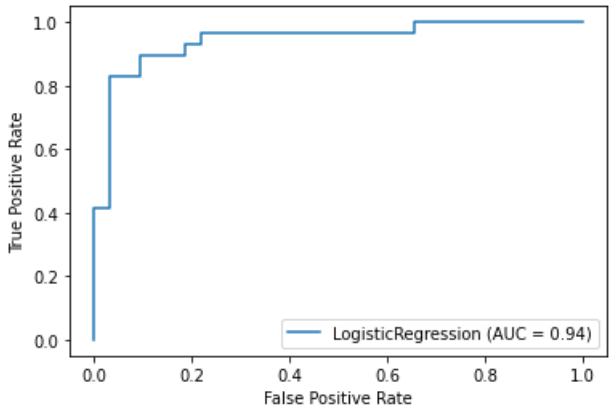
```
# Logistic Regression
clf = LogisticRegression(solver='liblinear', penalty='l1', max_iter=50000).fit(X_train, y_train)
y_pred = clf.predict(X_test)

reportFourMetrics(y_test, y_pred)
draw_confusion_matrix(y_test, y_pred, ["sick", "not sick"])
metrics.plot_roc_curve(clf, X_test, y_test)
```

```
Accuracy:  0.8852459016393442
Precision:  0.8666666666666667
recall:  0.896551724137931
F1 Score:  0.8813559322033899
```



Out[27]:

<sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x7fc992fc0e20>



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

We used two solvers: sag and liblinear. Liblinear is a linear classification that supports logistic regression and linear support vector machines. SAG method optimizes the sum of a finite number of smooth convex functions. Its iteration cost is independent of the number of terms in the sum.

The liblinear likely to produce the optimal outcome is because it uses a coordinate descent (CD) algorithm that solves optimization problems by successively performing approximate minimization along coordinate directions or coordinate hyperplanes.

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.

L1 regularization adds an L1 penalty equal to the absolute value of the magnitude of coefficients(limit the size of coef). No penalty.

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 [28]:

```
# SVM
```

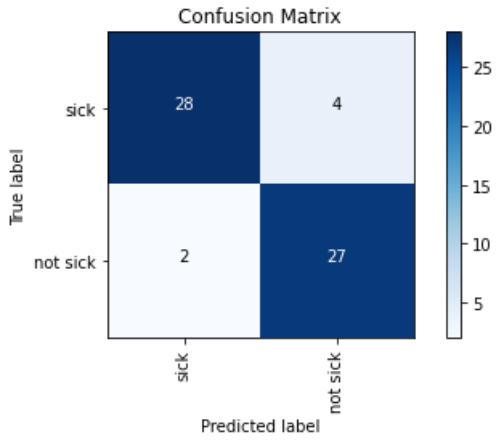
```
from sklearn.svm import SVC
clf = SVC(probability=True).fit(X_train, y_train)
y_pred = clf.predict(X_test)
```

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

In [29]:

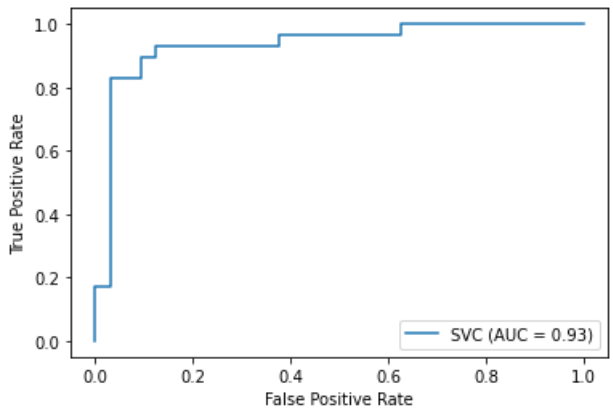
```
reportFourMetrics(y_test, y_pred)
draw_confusion_matrix(y_test, y_pred, ["sick", "not sick"])
metrics.plot_roc_curve(clf, X_test, y_test)
```

```
Accuracy:  0.9016393442622951
Precision:  0.8709677419354839
recall:  0.9310344827586207
F1 Score:  0.9
```



Out[29]:

```
<sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x7fc992198a90>
```

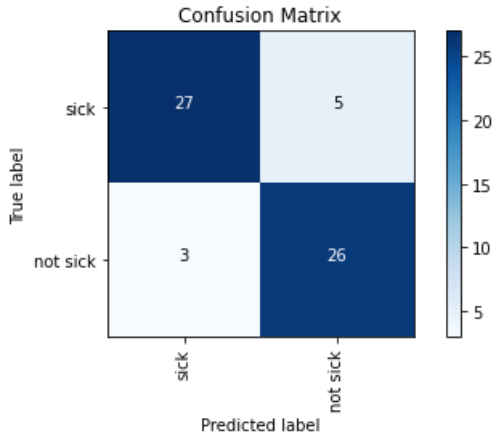


Rerun your SVM, but now modify your model parameter kernel to equal 'linear'. Again report your Accuracy, Precision, Recall, F1 scores, and Confusion matrix and plot the new ROC curve.

In [30]:

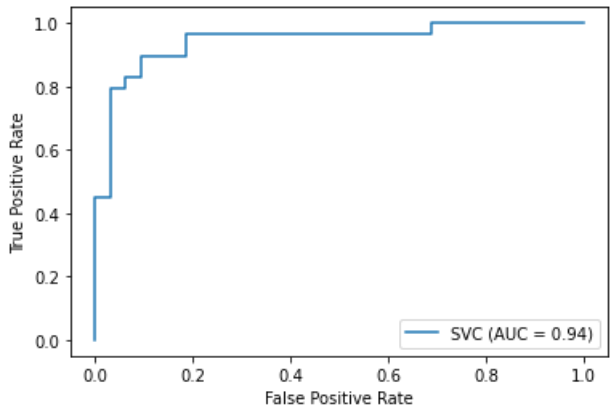
```
# SVM
clf_new = SVC(probability=True, kernel='linear').fit(X_train, y_train)
y_pred = clf_new.predict(X_test)

reportFourMetrics(y_test, y_pred)
draw_confusion_matrix(y_test, y_pred, ["sick", "not sick"])
metrics.plot_roc_curve(clf_new, X_test, y_test)
```

```
Accuracy:  0.8688524590163934
Precision:  0.8387096774193549
recall:  0.896551724137931
F1 Score:  0.8666666666666666
```



Out[30]:

```
<sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x7fc992fd6940>
```



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.

The new AUC I achieved is 0.94 which is higher than the first one which is 0.93. I changed the kernel to linear, which is going to not map to an even higher feature space. The first one might cannot be well linear separated, so that the second's performance is better.

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?

For SVM, it tries to find the "best" margin, which is the distance between the line and support vectors that seperate the classes. However, unlike SVM, Logistic regression is ablet to have diffenrent decision boundaries regarding to different weights that are close the optimal point.

Baysian (Statistical) Classification

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

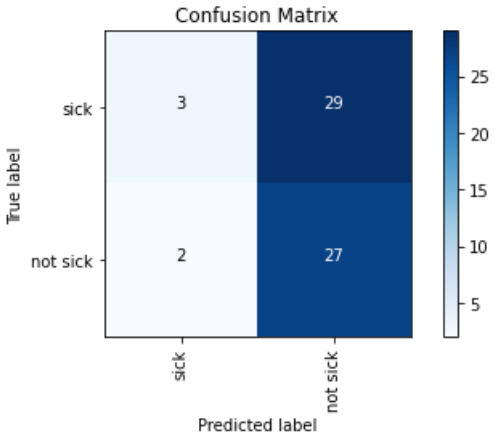
Naive Bayes methods are a set of supervised learning algorithms based on applying Bayes' theorem with the "naive" assumption of conditional independence between every pair of features given the value of the class variable. Bayes' theorem states the following relationship, given class variable Y and dependent 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 [31]:

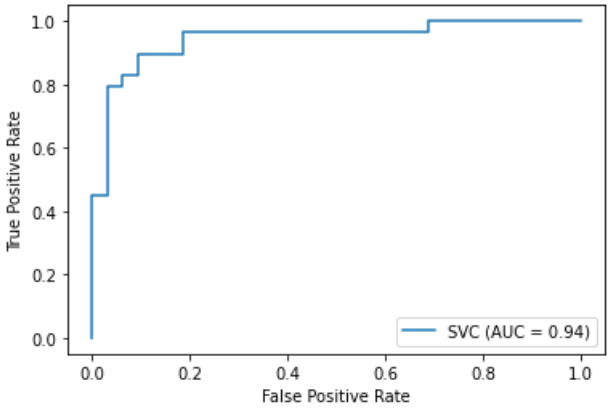
```
from sklearn.naive_bayes import GaussianNB
gnb = GaussianNB()
y_pred = gnb.fit(X_train, y_train).predict(X_test)

reportFourMetrics(y_test, y_pred)
draw_confusion_matrix(y_test, y_pred, ["sick", "not sick"])
metrics.plot_roc_curve(clf_new, X_test, y_test)
```

```
Accuracy:  0.4918032786885246
Precision:  0.48214285714285715
recall:  0.9310344827586207
F1 Score:  0.6352941176470589
```



Out[31]:

```
<sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x7fc99209bd90>
```



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

According to the result we get, the accuracy, precision, and F1 score are very low with 0.49, 0.48, and 0.64. By looking at the confusion matrix we can see that the number sick and non sick people are very different. As we expected, every feature in the data should independent and have qual contribution to the prediction result. In this case, we didn't get what we need with the inequal controbution, and that's why might those be inacurate.

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 performance. Based on these experiments you should have settled on a particular model that performs most optimally on the chosen dataset.

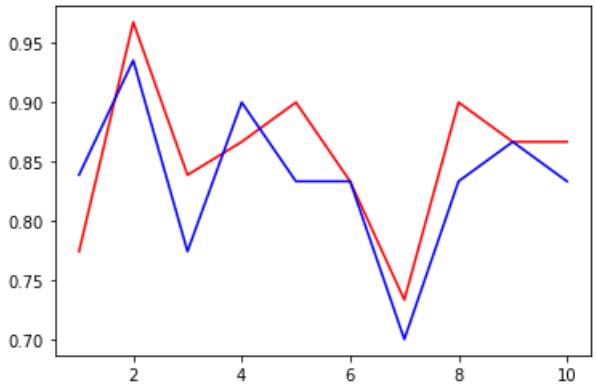
Before our work is done though, we want to ensure that our results are not the result of the random sampling of our data we did with the Train-Test-Split. To ensure otherwise we will conduct a K-Fold Cross-Validation of our top two performing models, assess their cumulative performance across folds, 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 [32]:

```
clf_1 = LogisticRegression(solver='liblinear', penalty='l1')
clf_2 = SVC(probability=True, kernel='linear')
cv = KFold(n_splits = 10, random_state = 42, shuffle = True)

cross_1 = cross_val_score(clf_1, df_X_prep, df_y, cv = cv)
cross_2 = cross_val_score(clf_2, df_X_prep, df_y, cv = cv)
x = [i for i in range(1, 11)]
line1 = plt.plot(x, cross_1,"r")
line2 =  plt.plot(x, cross_2,"b")
print(f"Logistic Regression max: {max(cross_1)}")
print(f"SVC max: {max(cross_2)}")
```

```
Logistic Regression max: 0.967741935483871
SVC max: 0.9354838709677419
```



My best performing mode is Logistic Regression with liblinear solver.