25W-COM SCI-M148 Project 2 - Binary Classification Comparative Methods

Name: Lana Lim

UID: 105817312

Submission Guidelines

1. Please fill in your name and UID above.
2. Please submit a **PDF printout** of your Jupyter Notebook to **Gradescope**. If you have any trouble accessing Gradescope, please let a TA know ASAP.
3. As the PDF can get long, please tag the respective sections to ensure the readers know where to look.

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

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

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

DEFINITIONS

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

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

Background: The Dataset

For this exercise, we will be using a subset of the UCI Heart Disease dataset. This dataset was created by collecting clinical data from patients undergoing diagnostic tests for heart disease. All identifying information about the patients has been removed to protect their privacy. The dataset represents data from patients who were suspected of having heart disease and underwent several diagnostic tests, including blood tests, electrocardiograms (ECG), exercise stress tests, and fluoroscopic imaging.

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

age: Patient age in years

sex: Patient sex (1 = male; 0 = female)

c_pain: Chest pain type (0 = asymptomatic; 1 = atypical angina (unusual discomfort due to reduced blood flow to the heart); 2 = non-anginal pain (chest pain unrelated to the heart); 3 = typical angina (classic chest discomfort due to reduced blood flow to the heart))

rbp: Resting blood pressure in mm Hg (measured at hospital admission)

chol: Serum cholesterol level in mg/dL

high_fbs: Fasting blood sugar > 120 mg/dL (1 = true; 0 = false)

r_ecg: Resting electrocardiographic results (0 = probable thickened left ventricular wall; 1 = normal; 2 = ST-T wave abnormality)

hr_max: Maximum heart rate achieved during the stress test

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

ecg_depress: Depression of the ST segment on ECG during exercise compared to rest (measured in mm)

stress_slope: Slope of the peak exercise ST segment (0 = downsloping (concerning); 1 = flat (abnormal); 2 = upsloping (normal))

num_vessels: Number of major vessels (0–3) showing good blood flow during fluoroscopy

thal_test_res: Thallium Stress Test result (assesses blood flow using trace amounts of radioactive thallium-201) (1 = normal; 2 = fixed defect; 7 = reversible defect)

heart_disease: Indicates whether heart disease is present (True = Disease; False = No disease)

Loading Essentials and Helper Functions

```
#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)
```

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.)

```
data = 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.

```
data.head()
```

	age	sex	chest_pain	rpb	chol	high_fbs	r_ecg	hr_max	ex_ang	ecg_depress	stress_slope	nu
0	63	1	3	145	233	1	0	150	0	2.3	0	0
1	37	1	2	130	250	0	1	187	0	3.5	0	0
2	41	0	1	130	204	0	0	172	0	1.4	2	0
3	56	1	1	120	236	0	1	178	0	0.8	2	0
4	57	0	0	120	354	0	1	163	1	0.6	2	0

```
data.describe()
```

	age	sex	chest_pain	rpb	chol	high_fbs	r_ecg	hr_max
count	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000
mean	54.366337	0.683168	0.966997	131.623762	246.264026	0.148515	0.528053	149.646865
std	9.082101	0.466011	1.032052	17.538143	51.830751	0.356198	0.525860	22.905161
min	29.000000	0.000000	0.000000	94.000000	126.000000	0.000000	0.000000	71.000000
25%	47.500000	0.000000	0.000000	120.000000	211.000000	0.000000	0.000000	133.500000
50%	55.000000	1.000000	1.000000	130.000000	240.000000	0.000000	1.000000	153.000000
75%	61.000000	1.000000	2.000000	140.000000	274.500000	0.000000	1.000000	166.000000
max	77.000000	1.000000	3.000000	200.000000	564.000000	1.000000	2.000000	202.000000

```
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   chest_pain     303 non-null    int64
 3   rpb            303 non-null    int64
 4   chol           303 non-null    int64
 5   high_fbs       303 non-null    int64
 6   r_ecg          303 non-null    int64
 7   hr_max         303 non-null    int64
 8   ex_ang         303 non-null    int64
 9   ecg_depress    303 non-null    float64
 10  stress_slope   303 non-null    int64
 11  num_vessels    303 non-null    int64
 12  thal_test_res  303 non-null    int64
 13  heart_disease  303 non-null    bool
dtypes: bool(1), float64(1), int64(12)
memory usage: 31.2 KB
```

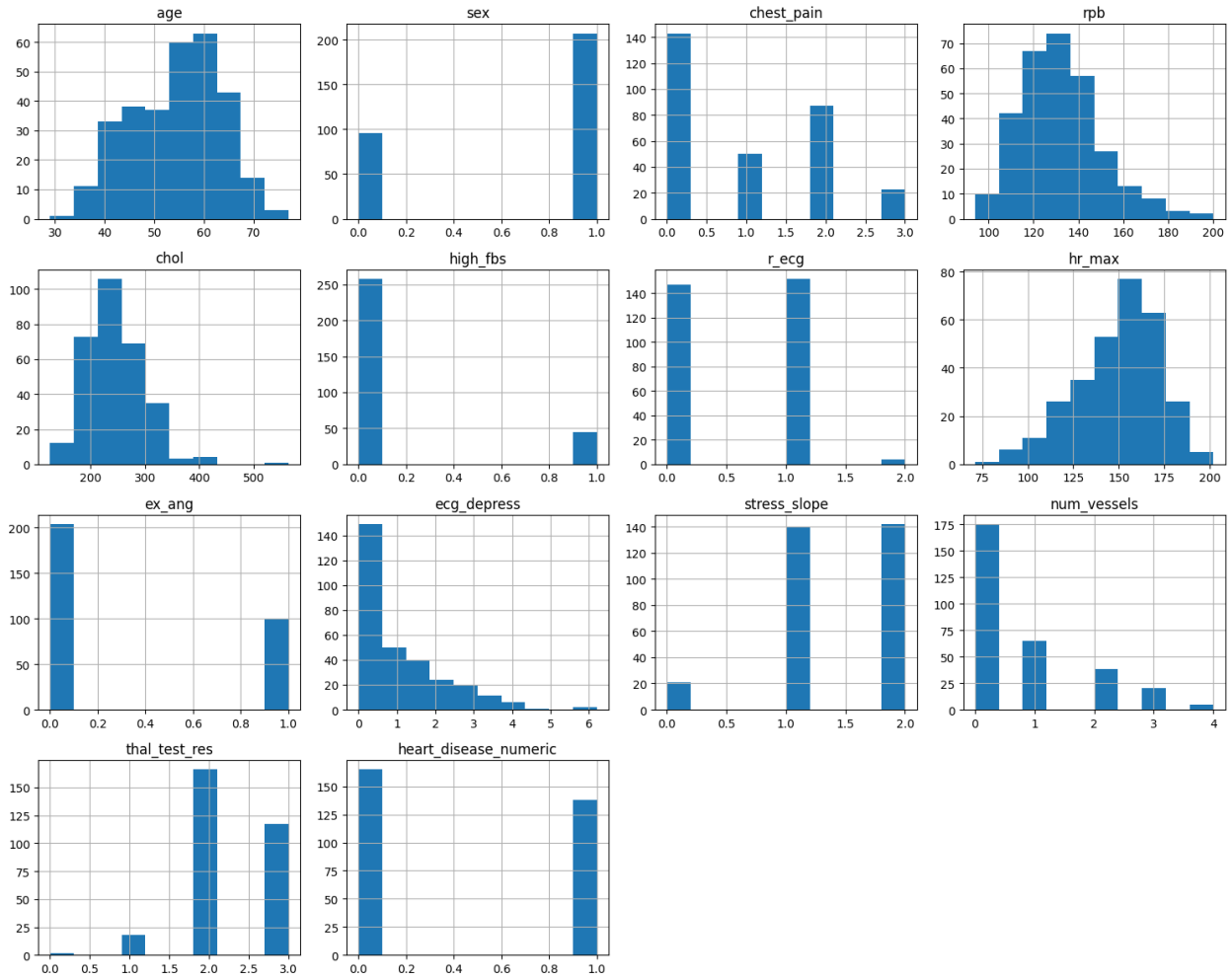
Before we begin our analysis we need to fix the field(s) that will be problematic. Specifically convert our boolean heart_disease variable into a binary numeric target variable (values of either '0' or '1'), and then drop the original heart_disease datafield from the dataframe. (hint: try label encoder or .astype())

```
data['heart_disease_numeric'] = data['heart_disease'].astype(int)
data = data.drop('heart_disease', axis=1)
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   chest_pain             303 non-null    int64
 3   rpb                    303 non-null    int64
 4   chol                   303 non-null    int64
 5   high_fbs               303 non-null    int64
 6   r_ecg                  303 non-null    int64
 7   hr_max                 303 non-null    int64
 8   ex_ang                 303 non-null    int64
 9   ecg_depress            303 non-null    float64
 10  stress_slope           303 non-null    int64
 11  num_vessels            303 non-null    int64
 12  thal_test_res          303 non-null    int64
 13  heart_disease_numeric  303 non-null    int64
dtypes: float64(1), int64(13)
memory usage: 33.3 KB
```

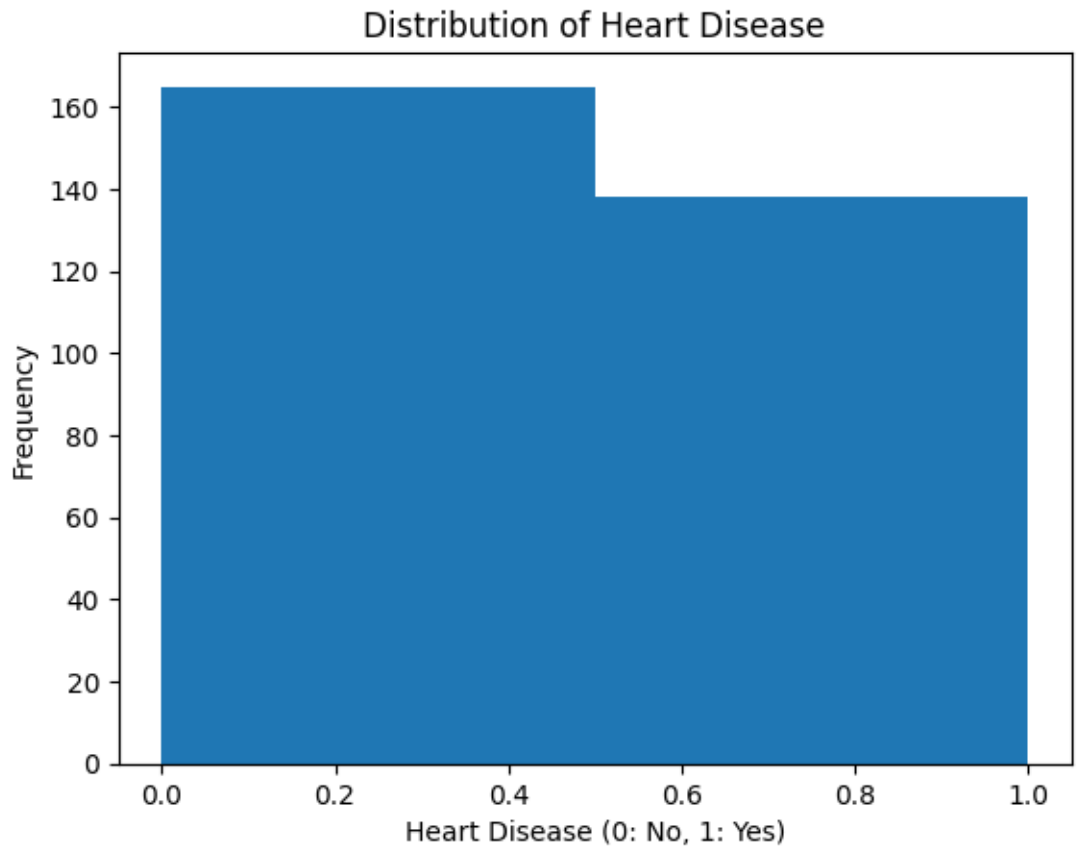
Now that we have a feel for the data-types for each of the variables, plot histograms of each field and attempt to ascertain how each variable performs (is it a binary, or limited selection, or does it follow a gradient?

```
data.hist(figsize=(15, 12))
plt.tight_layout()
plt.show()
```



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 heart_disease target, and conduct a count of the number of diseased and healthy individuals and report on the results:

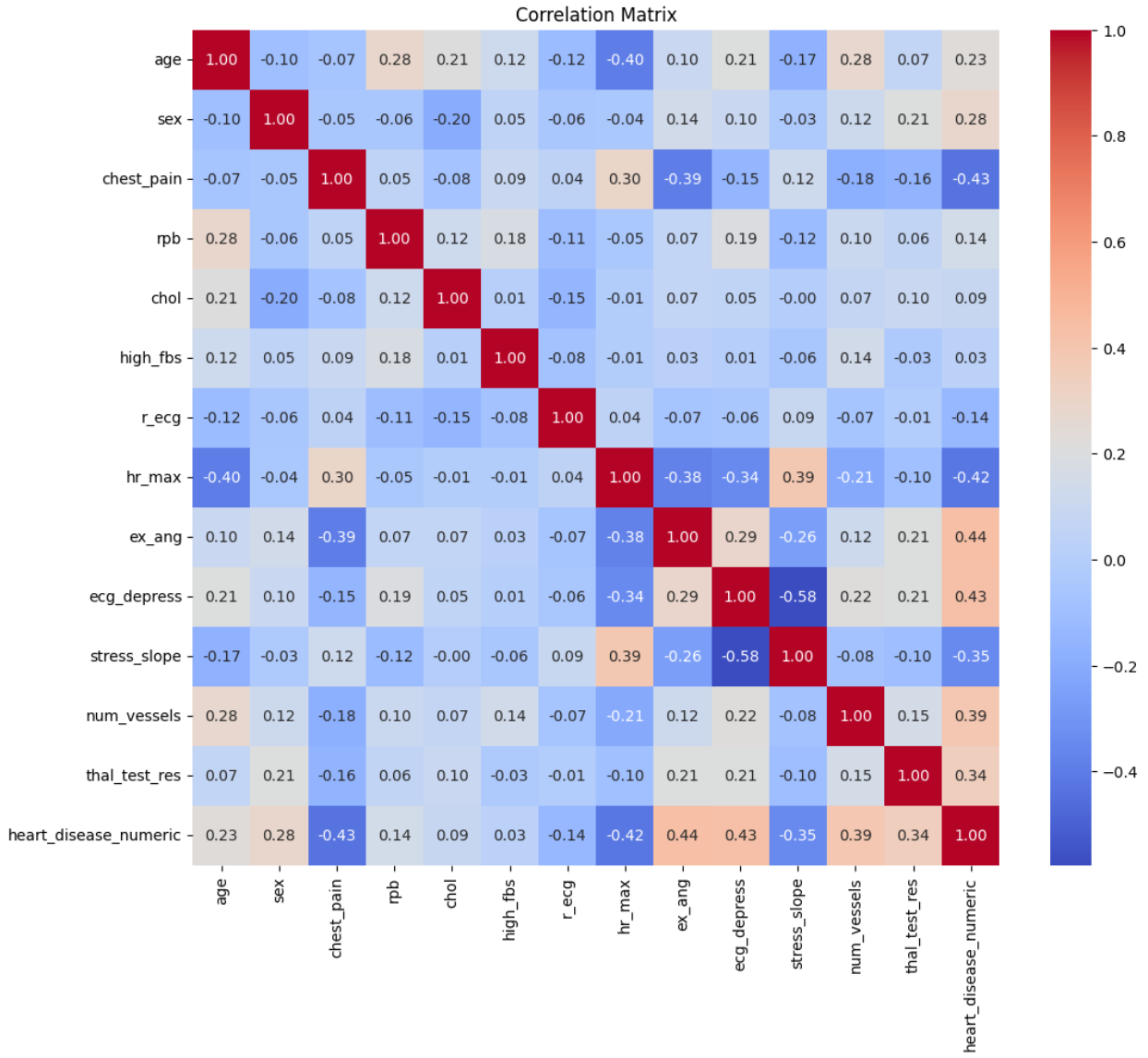
```
plt.hist(data['heart_disease_numeric'], bins=2)
plt.xlabel('Heart Disease (0: No, 1: Yes)')
plt.ylabel('Frequency')
plt.title('Distribution of Heart Disease')
plt.show()

disease_counts = data['heart_disease_numeric'].value_counts()
print(disease_counts)
```




```
heart_disease_numeric
0    165
1    138
Name: count, dtype: int64
```

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)?

```
# Calculate correlations
correlations = data.corr()

# Plot heatmap
plt.figure(figsize=(12, 10))  # Adjust figure size as needed
sns.heatmap(correlations, annot=True, cmap='coolwarm', fmt=".2f")
plt.title('Correlation Matrix')
plt.show()
```



Positive Correlation: A positive correlation (closer to 1) means that as one variable increases, the other tends to increase as well. For example, chest_pain and hr_max have a positive correlation.

Negative Correlation: A negative correlation (closer to -1) means that as one variable increases, the other tends to decrease. For example, age and hr_max have a negative correlation.

No Correlation: A correlation close to 0 indicates little or no linear relationship between the variables (e.g, ex_ang and age).

Some variables might correlate more highly with heart disease due to their direct physiological relevance. For example, chest pain is likely to have a strong correlation because certain types of chest pain are indicative of heart problems. Other variables might have weaker correlations due to indirect relationships or the presence of confounding factors.

Part 2. Prepare the 'Raw' 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.

```
labels = data['heart_disease_numeric'].values
features = data.drop('heart_disease_numeric', axis=1)
```

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

```
# Split the data into training and testing sets (70% train, 30% test)
X_train_raw, X_test_raw, y_train, y_test = train_test_split(
    features, labels, test_size=0.3, random_state=42
)
```

```
# Print the shapes of the resulting datasets
print("X_train_raw shape:", X_train_raw.shape)
print("X_test_raw shape:", X_test_raw.shape)
print("y_train shape:", y_train.shape)
print("y_test shape:", y_test.shape)
```

```
X_train_raw shape: (212, 13)
X_test_raw shape: (91, 13)
y_train shape: (212,)
y_test shape: (91,)
```

We'll explore how not processing your data can impact model performance by using the K-Nearest Neighbor classifier. One thing to note 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 raw 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.

```
# Dfault settings
knn_raw = KNeighborsClassifier()
knn_raw.fit(X_train_raw, y_train)
y_pred_raw = knn_raw.predict(X_test_raw)
```

```
from sklearn.metrics import accuracy_score
accuracy_raw = accuracy_score(y_test, y_pred_raw)
print("Accuracy (Raw Data):", accuracy_raw)
```

```
Accuracy (Raw Data): 0.6593406593406593
```

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. Use the fit_transform() to fit this pipeline to your training data. and then transform() to apply that pipeline to your test data

Hint: 1. Create separate pipelines for numeric and categorical features with Pipeline() and then combining them with ColumnTransformer() 2. First, fit the full pipeline with the training data. Then, apply it to the test data as well.

Pipeline:

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

```
numeric_features = ['age', 'rpb', 'chol', 'hr_max', 'ecg_depress']
categorical_features = ['sex', 'chest_pain', 'high_fbs', 'r_ecg', 'ex_ang',
'stress_slope', 'num_vessels', 'thal_test_res']

numeric_pipeline = Pipeline([
    ('scaler', StandardScaler())
])

categorical_pipeline = Pipeline([
    ('onehot', OneHotEncoder(sparse_output=False, handle_unknown='ignore'))  #
sparse=False for KNN
```

```
])

# Combine pipelines using ColumnTransformer
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_pipeline, numeric_features),
        ('cat', categorical_pipeline, categorical_features)
    ])
```

```
# Pipeline the training and test data
X_train_processed = preprocessor.fit_transform(X_train_raw)
X_test_processed = preprocessor.transform(X_test_raw)
```

Now retrain your model and compare the accuracy metrics (Accuracy, Precision, Recall, F1 Score) with the raw and pipelined data.

```
# KNN
knn_processed = KNeighborsClassifier()
knn_processed.fit(X_train_processed, y_train)
y_pred_processed = knn_processed.predict(X_test_processed)
```

```
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score
# Calculate metrics for raw data
accuracy_raw = accuracy_score(y_test, y_pred_raw)
precision_raw = precision_score(y_test, y_pred_raw)
recall_raw = recall_score(y_test, y_pred_raw)
f1_raw = f1_score(y_test, y_pred_raw)

# Calculate metrics for processed data
accuracy_processed = accuracy_score(y_test, y_pred_processed)
precision_processed = precision_score(y_test, y_pred_processed)
recall_processed = recall_score(y_test, y_pred_processed)
f1_processed = f1_score(y_test, y_pred_processed)

# Print the metrics
print("Metrics for Raw Data:")
print("Accuracy:", accuracy_raw)
print("Precision:", precision_raw)
print("Recall:", recall_raw)
print("F1 Score:", f1_raw)
print("\nMetrics for Processed Data:")
print("Accuracy:", accuracy_processed)
print("Precision:", precision_processed)
```

```
print("Recall:", recall_processed)
print("F1 Score:", f1_processed)
```

```
Metrics for Raw Data:
Accuracy: 0.6593406593406593
Precision: 0.631578947368421
Recall: 0.5853658536585366
F1 Score: 0.6075949367088608

Metrics for Processed Data:
Accuracy: 0.8571428571428571
Precision: 0.8333333333333334
Recall: 0.8536585365853658
F1 Score: 0.8433734939759037
```

The results clearly demonstrate a significant improvement in the model's performance after applying data preprocessing. This is evident from the substantial increase in all the evaluation metrics: accuracy, precision, recall, and F1 score.

Raw Data Metrics: - Accuracy (0.659): The model correctly classified around 66% of the samples in the testing set. - Precision (0.632): Out of all the samples predicted as positive (having heart disease), about 63% were actually positive. - Recall (0.585): Out of all the samples that actually had heart disease, the model correctly identified around 59%. - F1 Score (0.608): A balanced measure considering both precision and recall, indicating a moderate overall performance.

Processed Data Metrics: - Accuracy (0.857): The model's accuracy improved to about 86% after preprocessing. - Precision (0.833): The precision increased significantly, indicating a higher proportion of true positive predictions among all positive predictions. - Recall (0.854): The recall also improved, showing that the model is better at identifying actual positive cases. - F1 Score (0.843): The F1 score increased considerably, reflecting a much better overall performance.

Parameter Optimization. The KNN Algorithm includes an n_neighbors attribute that specifies how many neighbors to use when developing the cluster. (The default value is 5, which is what your previous model used.) Lets now try n values of: 1, 2, 3, 5, 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).

```
# Define the values of n_neighbors to try
n_neighbors_values = [1, 2, 3, 5, 7, 9, 10, 20, 50]

for n_neighbors in n_neighbors_values:
    knn = KNeighborsClassifier(n_neighbors=n_neighbors)

    # Train on training data
    knn.fit(X_train_processed, y_train)
```

```
    # Make predictions on testing data
    y_pred = knn.predict(X_test_processed)

    # Get accuracy
    accuracy = accuracy_score(y_test, y_pred)
    print(f"Accuracy (n_neighbors={n_neighbors}):", accuracy)
```

```
Accuracy (n_neighbors=1): 0.7692307692307693
Accuracy (n_neighbors=2): 0.8021978021978022
Accuracy (n_neighbors=3): 0.8351648351648352
Accuracy (n_neighbors=5): 0.8571428571428571
Accuracy (n_neighbors=7): 0.8901098901098901
Accuracy (n_neighbors=9): 0.8461538461538461
Accuracy (n_neighbors=10): 0.8461538461538461
Accuracy (n_neighbors=20): 0.8791208791208791
Accuracy (n_neighbors=50): 0.8461538461538461
```

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,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.

Report metrics for:

1. Accuracy
2. Precision
3. Recall
4. F1 Score

```
logreg = LogisticRegression()

logreg.fit(X_train_processed, y_train)

y_pred_logreg = logreg.predict(X_test_processed)
```

```
accuracy_logreg = accuracy_score(y_test, y_pred_logreg)
precision_logreg = precision_score(y_test, y_pred_logreg)
recall_logreg = recall_score(y_test, y_pred_logreg)
f1_logreg = f1_score(y_test, y_pred_logreg)

print("Metrics for Logistic Regression:")
print("Accuracy:", accuracy_logreg)
print("Precision:", precision_logreg)
print("Recall:", recall_logreg)
print("F1 Score:", f1_logreg)
```

```
Metrics for Logistic Regression:
Accuracy: 0.8571428571428571
Precision: 0.868421052631579
Recall: 0.8048780487804879
F1 Score: 0.8354430379746836
```

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: - What it reports: The proportion of correctly classified samples (both positive and negative) out of the total number of samples. - Why it's significant: It provides an overall measure of the model's correctness. - Use cases: Relying on accuracy alone can be misleading for imbalanced datasets where one class is much more prevalent than the other.

Precision: - What it reports: Out of all the samples predicted as positive, what proportion was actually positive. - Why it's significant: Focuses on the accuracy of positive predictions, minimizing false positives. - Use cases: Scenarios where the cost of false positives is high, such as spam detection (avoiding classifying legitimate emails as spam).

Recall: - What it reports: Out of all the actual positive samples, what proportion was correctly predicted as positive. - Why it's significant: Focuses on identifying all positive cases, minimizing false negatives. - Use cases: Scenarios where the cost of false negatives is high, such as disease diagnosis (avoiding missing actual cases of the disease).

F1 Score: - What it reports: The harmonic mean of precision and recall, providing a balanced measure of both. - Why it's significant: Considers both false positives and false negatives, giving a more comprehensive evaluation. - Use cases: When you need a balance between precision and recall, such as in information retrieval where you want to retrieve relevant documents while minimizing irrelevant ones.

Let's tweak a few settings. First let's set your solver to 'sag' (Stochastic Average Gradient), your max_iter= 10, and set penalty = None and rerun your model. Let's see how your results change!

```
logreg = LogisticRegression(solver='sag', max_iter=10, penalty=None)

logreg.fit(X_train_processed, y_train)

y_pred_logreg = logreg.predict(X_test_processed)

accuracy_logreg = accuracy_score(y_test, y_pred_logreg)
precision_logreg = precision_score(y_test, y_pred_logreg)
recall_logreg = recall_score(y_test, y_pred_logreg)
f1_logreg = f1_score(y_test, y_pred_logreg)

print("Metrics for Logistic Regression (with modified settings):")
print("Accuracy:", accuracy_logreg)
print("Precision:", precision_logreg)
print("Recall:", recall_logreg)
print("F1 Score:", f1_logreg)
```

```
Metrics for Logistic Regression (with modified settings):
Accuracy: 0.8351648351648352
Precision: 0.825
Recall: 0.8048780487804879
F1 Score: 0.8148148148148148
```

```
/usr/local/lib/python3.11/dist-packages/sklearn/linear_model/_sag.py:348:
ConvergenceWarning: The max_iter was reached which means the coef_ did not
converge
  warnings.warn(
```

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.

```
# Scale numeric features
scaler = StandardScaler()
X_train_raw[numeric_features] =
scaler.fit_transform(X_train_raw[numeric_features])
X_test_raw[numeric_features] = scaler.transform(X_test_raw[numeric_features])

# Process data (using the same preprocessor as before)
X_train_processed = preprocessor.fit_transform(X_train_raw)
```

```
X_test_processed = preprocessor.transform(X_test_raw)

# Train and evaluate with regularization
logreg = LogisticRegression(solver='sag', max_iter=100, penalty='l2', C=1.0)
logreg.fit(X_train_processed, y_train)

accuracy_logreg = accuracy_score(y_test, y_pred_logreg)
precision_logreg = precision_score(y_test, y_pred_logreg)
recall_logreg = recall_score(y_test, y_pred_logreg)
f1_logreg = f1_score(y_test, y_pred_logreg)

print("Metrics for Logistic Regression (with increased max_iter):")
print("Accuracy:", accuracy_logreg)
print("Precision:", precision_logreg)
print("Recall:", recall_logreg)
print("F1 Score:", f1_logreg)
```

```
Metrics for Logistic Regression (with increased max_iter):
Accuracy: 0.8351648351648352
Precision: 0.825
Recall: 0.8048780487804879
F1 Score: 0.8148148148148148
```

Changes Made: - Scaling numeric features before pipeline: applied StandardScaler directly to the numeric features (X_train_raw[numeric_features] and X_test_raw[numeric_features]) before feeding them into the ColumnTransformer pipeline. By scaling the numeric features before the pipeline, all numeric features have a similar range of values. Without proper scaling, features with larger values can dominate the optimization process, leading to slower convergence or divergence. - Adding L2 regularization: set penalty='l2' and C=1.0 in the LogisticRegression model. Regularization can help by making the loss function easier to converge to a minimum by discouraging the model from assigning extremely large weights to features.

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

```
# Train and evaluate with L1 penalty and liblinear solver
logreg = LogisticRegression(penalty='l1', solver='liblinear', max_iter=100)
logreg.fit(X_train_processed, y_train)
y_pred_logreg = logreg.predict(X_test_processed)

accuracy_logreg = accuracy_score(y_test, y_pred_logreg)
precision_logreg = precision_score(y_test, y_pred_logreg)
recall_logreg = recall_score(y_test, y_pred_logreg)
f1_logreg = f1_score(y_test, y_pred_logreg)

print("Metrics for Logistic Regression (with L1 penalty and liblinear solver):")
```

```
print("Accuracy:", accuracy_logreg)
print("Precision:", precision_logreg)
print("Recall:", recall_logreg)
print("F1 Score:", f1_logreg)
```

```
Metrics for Logistic Regression (with L1 penalty and liblinear solver):
Accuracy: 0.8571428571428571
Precision: 0.868421052631579
Recall: 0.8048780487804879
F1 Score: 0.8354430379746836
```

Explain what the two solver approaches are, and why liblinear may have produced an improved outcome (but not always, and it's ok if your results show otherwise!).

Solver Approaches: - 'sag' (Stochastic Average Gradient): a variant of Stochastic Gradient Descent (SGD). Suitable for both L1 and L2 regularization. - 'liblinear' (Library for Large Linear Classification): specialized library designed for linear classification problems. It uses a coordinate descent algorithm to optimize the model's coefficients. It's particularly efficient for smaller datasets and L1 regularization.

Why 'liblinear' Might Improve Outcome: - L1 Regularization and Feature Selection: L1 regularization encourages sparsity in the model's coefficients, meaning it tries to set some coefficients to exactly zero. This can act as a form of feature selection, potentially removing irrelevant or redundant features. This is particularly beneficial when dealing with datasets that have a large number of features but only a few of them are actually informative. - The 'liblinear' solver is known for its efficiency in handling smaller datasets. Therefore 'liblinear' might be better suited to find the optimal solution compared to 'sag', which is more geared towards larger datasets. - While the data might not be inherently sparse, L1 regularization can introduce sparsity in the model's coefficients, making the data effectively sparse. 'liblinear' is designed to handle sparse data efficiently, which could contribute to faster convergence and potentially better performance.

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.

More explanation here: https://en.wikipedia.org/wiki/Support_vector_machine.

For the sake of this project, you can regard it as a type of classifier.

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.

```
svm = SVC(probability=True)  # probability=True
svm.fit(X_train_processed, y_train)
y_pred_svm = svm.predict(X_test_processed)
```

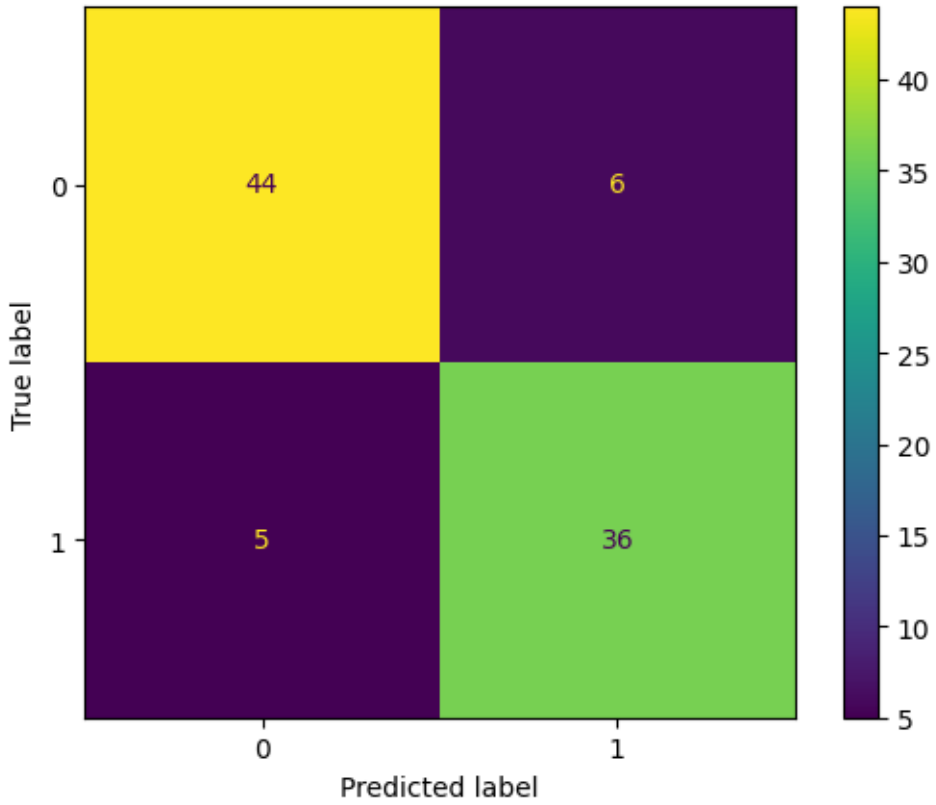
Report the accuracy, precision, recall, F1 Score, of your model, but in addition, plot a Confusion Matrix of your model's performance

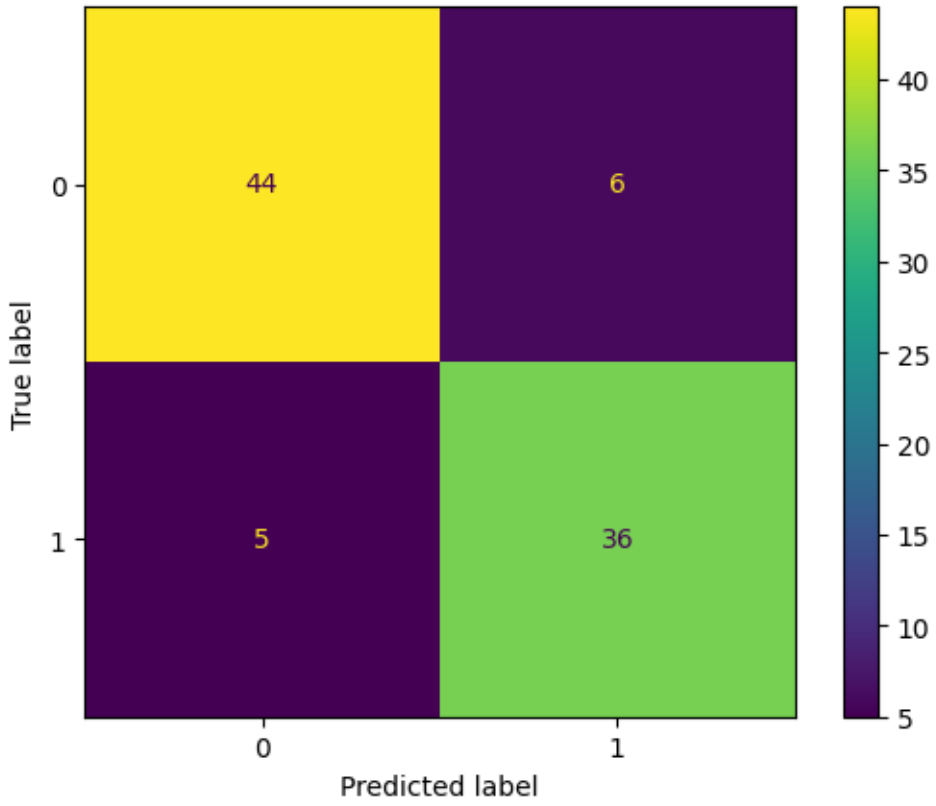
recommend using `from sklearn.metrics import ConfusionMatrixDisplay` for this one!

```
# Metrics
accuracy_svm = accuracy_score(y_test, y_pred_svm)
precision_svm = precision_score(y_test, y_pred_svm)
recall_svm = recall_score(y_test, y_pred_svm)
f1_svm = f1_score(y_test, y_pred_svm)

print("Metrics for SVM:")
print("Accuracy:", accuracy_svm)
print("Precision:", precision_svm)
print("Recall:", recall_svm)
print("F1 Score:", f1_svm)
```

```
Metrics for SVM:
Accuracy: 0.8791208791208791
Precision: 0.8571428571428571
Recall: 0.8780487804878049
F1 Score: 0.8674698795180723
```

```
# Confusion Matrix
from sklearn.metrics import ConfusionMatrixDisplay

cm = ConfusionMatrixDisplay.from_predictions(y_test, y_pred_svm)
cm.plot()
plt.show()
```

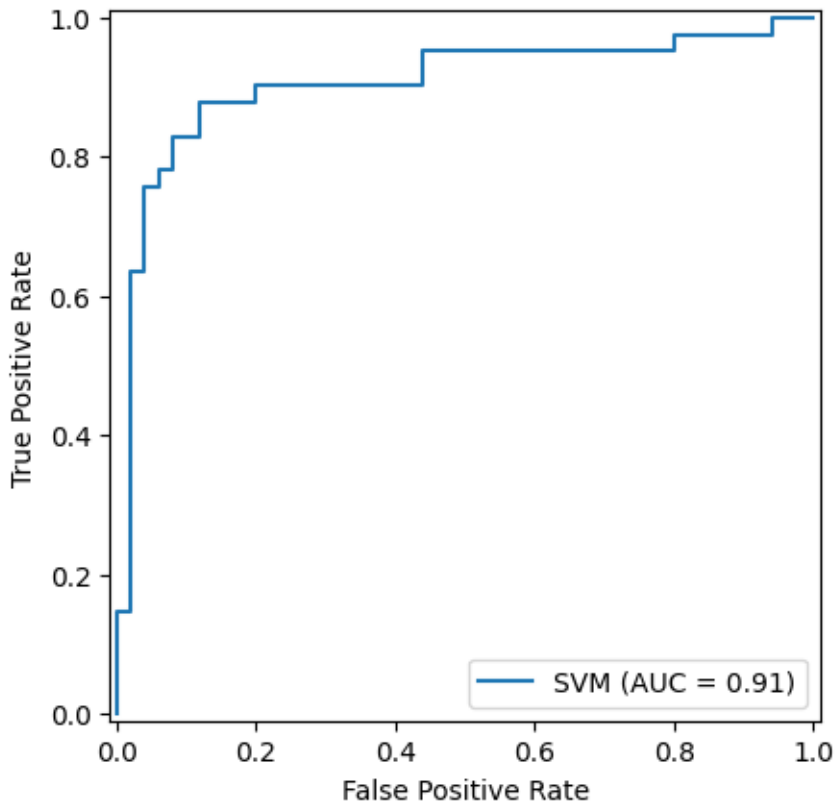

44
6
5
36
0
1
True label
0
1
Predicted label
40
35
30
25
20
15
10
5


44
6
5
36
0
1
True label
0
1
Predicted label
40
35
30
25
20
15
10
5

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

recommend using the `metrics.roc_curve` `metrics.auc` and `metrics.RocCurveDisplay` for this one!

```
# ROC
from sklearn.metrics import roc_curve, auc, RocCurveDisplay

y_scores_svm = svm.predict_proba(X_test_processed)[:, 1]  # Probability of class
1 (heart disease)

fpr, tpr, thresholds = roc_curve(y_test, y_scores_svm)
roc_auc = auc(fpr, tpr)

display = RocCurveDisplay(fpr=fpr, tpr=tpr, roc_auc=roc_auc,
estimator_name='SVM')
display.plot()
plt.show()
```



[Describe what an ROC Curve is and what the results mean here]

The ROC curve is created by plotting the true positive rate (TPR) against the false positive rate (FPR) at various threshold settings. The area under an ROC curve is a measure of the usefulness

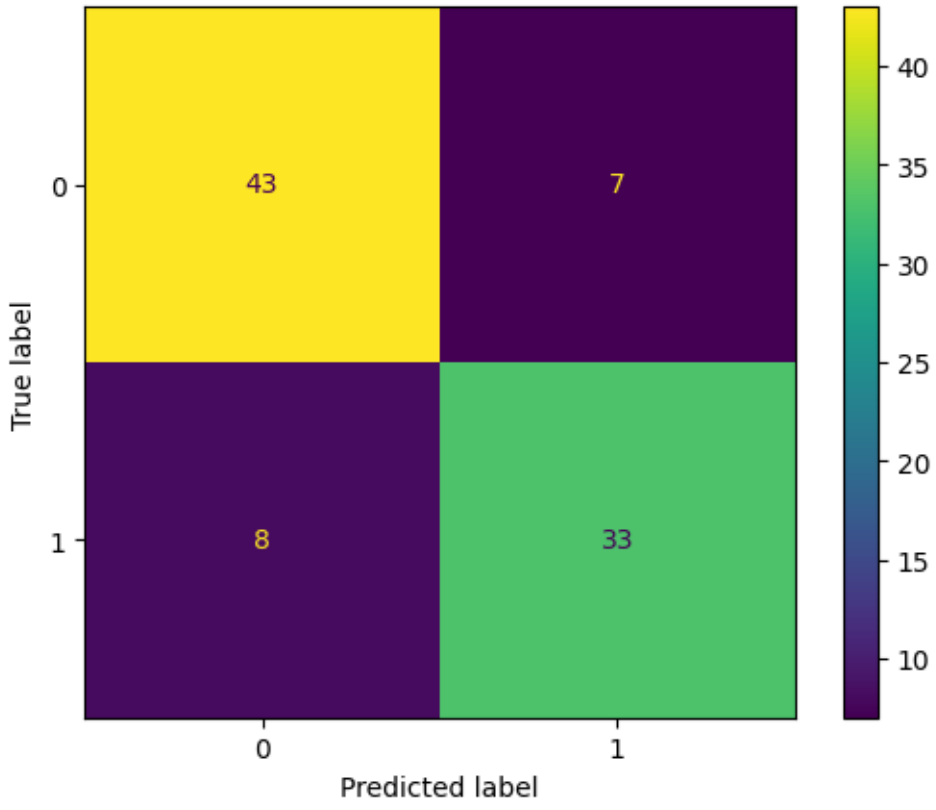
of a test in general, where a greater area means a more useful test, so the areas under ROC curves are used to compare the usefulness of tests. Here we see a relatively low area under the curve indicating a poorly performing model.

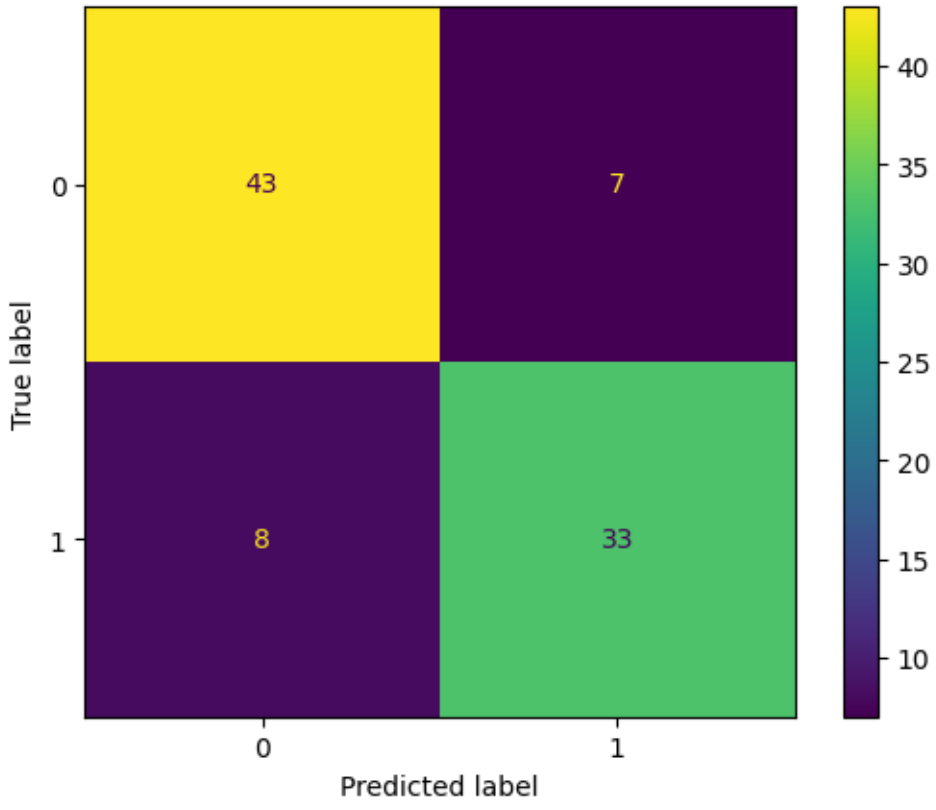
Rerun your SVM, but now modify your model parameter kernel to equal 'linear'. Again report your Accuracy, Precision, Recall, F1 scores, and Confusion matrix and plot the new ROC curve.

```
# Metrics
svm_linear = SVC(kernel='linear', probability=True)

svm_linear.fit(X_train_processed, y_train)

y_pred_svm_linear = svm_linear.predict(X_test_processed)

accuracy_svm_linear = accuracy_score(y_test, y_pred_svm_linear)
precision_svm_linear = precision_score(y_test, y_pred_svm_linear)
recall_svm_linear = recall_score(y_test, y_pred_svm_linear)
f1_svm_linear = f1_score(y_test, y_pred_svm_linear)

print("Metrics for SVM (Linear Kernel):")
print("Accuracy:", accuracy_svm_linear)
print("Precision:", precision_svm_linear)
print("Recall:", recall_svm_linear)
print("F1 Score:", f1_svm_linear)
```

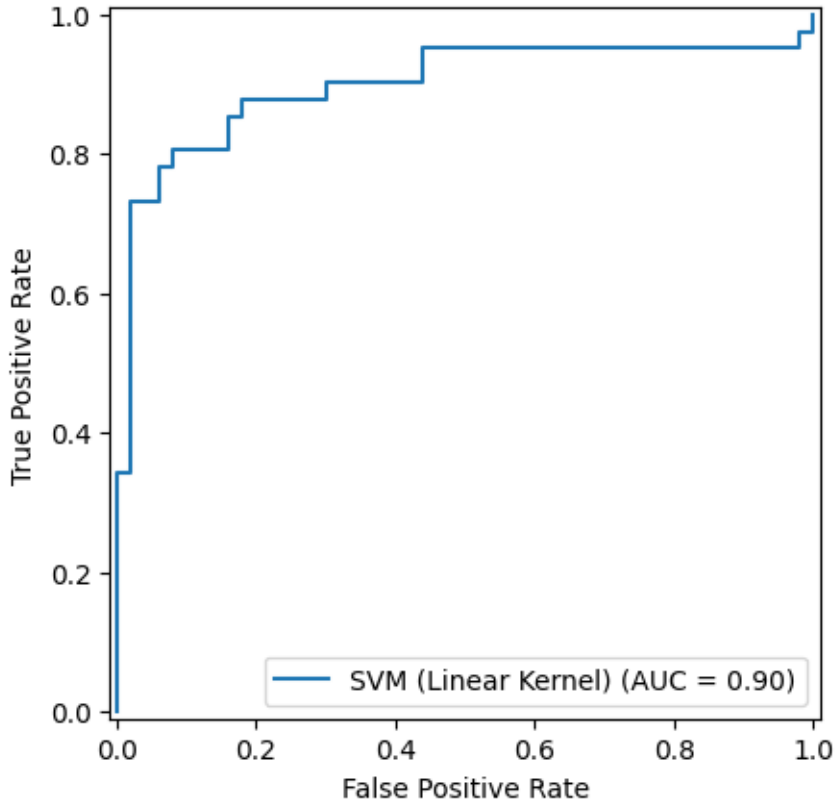
```
Metrics for SVM (Linear Kernel):
Accuracy: 0.8351648351648352
Precision: 0.825
Recall: 0.8048780487804879
F1 Score: 0.8148148148148148
```

```
# Confusion Matrix
cm_linear = ConfusionMatrixDisplay.from_predictions(y_test, y_pred_svm_linear)
cm_linear.plot()
plt.show()
```


43
7
8
33
0
1
True label
0
1
Predicted label
40
35
30
25
20
15
10


43
7
8
33
0
1
True label
0
1
Predicted label
40
35
30
25
20
15
10

```
# ROC
y_scores_svm_linear = svm_linear.predict_proba(X_test_processed)[:, 1]
fpr_linear, tpr_linear, thresholds_linear = roc_curve(y_test,
y_scores_svm_linear)
roc_auc_linear = auc(fpr_linear, tpr_linear)

display_linear = RocCurveDisplay(fpr=fpr_linear, tpr=tpr_linear,
roc_auc=roc_auc_linear, estimator_name='SVM (Linear Kernel)')
display_linear.plot()
plt.show()
```



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.

By setting kernel='linear' in the SVM model, the model uses a linear decision boundary to separate the data points. This means the model will try to find a straight line that best separates the classes. The default kernel is 'rbf' (Radial Basis Function), which uses a more complex, non-linear decision boundary.

Interpreting the Results: - Accuracy: The accuracy slightly decreased from 0.879 to 0.835 when using the linear kernel. This suggests that a linear decision boundary might not be as effective in

capturing the underlying patterns in the data compared to the more flexible 'rbf' kernel. - Precision: Precision decreased from 0.857 to 0.825. This indicates that the linear kernel model is slightly more prone to making false positive predictions (classifying a healthy individual as having heart disease). - Recall: Recall decreased from 0.878 to 0.805. This suggests that the linear kernel model is less effective at correctly identifying individuals with heart disease (more false negatives). - F1 Score: The F1 score, which balances precision and recall, also decreased from 0.867 to 0.815, reflecting the overall decrease in performance. - Confusion Matrix: The linear kernel model has more false positives and false negatives compared to the 'rbf' kernel model. - AUC: The AUC values are very similar for both models (0.91 for 'rbf' and 0.90 for 'linear'). This suggests that both models have similar overall discriminatory power, despite the differences in accuracy and other metrics.

In summary, the change in kernel impacted the results by potentially reducing the model's ability to capture complex relationships in the data, resulting in a decrease in accuracy, precision, recall, and F1 score. However, the AUC values suggest that the linear kernel is still a reasonable choice for this dataset. The choice between linear and 'rbf' kernels might depend on the specific trade-offs you're willing to make between accuracy, interpretability, and computational cost.

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?

Both Logistic Regression and Linear SVM find a linear decision boundary, but: - Logistic Regression: Maximizes the likelihood of the observed data belonging to their respective classes, focusing on probability estimates and minimizing classification errors across all data points. - Linear SVM: Maximizes the margin between classes, primarily influenced by support vectors (nearest data points) and ignoring points far from the boundary. It focuses on separating the classes with the largest margin.

Decision Trees

Create both a Decision Tree and a KNN and fit them onto your fully preprocessed data, then calculate an accuracy score for both (https://scikit-learn.org/stable/api/sklearn.tree.html).

What are Decision Trees?

Decision Trees are a non-parametric supervised learning methods used for classification and regression. The goal is to split data into branches based on feature conditions, forming a tree-like structure where each internal node represents a decision, and each branch represents an outcome.

Compared to KNN, decision trees is less influenced by the high dimensionality of the data, and can make the model output more predicable.

For more explanation, see here: https://en.wikipedia.org/wiki/Decision_tree. For the sake of this project, you can regard it as a type of classifier.

```
from sklearn.tree import DecisionTreeClassifier
# Decision Tree
tree = DecisionTreeClassifier()
tree.fit(X_train_processed, y_train)
y_pred_tree = tree.predict(X_test_processed)
accuracy_tree = accuracy_score(y_test, y_pred_tree)

# KNN
knn = KNeighborsClassifier()
knn.fit(X_train_processed, y_train)
y_pred_knn = knn.predict(X_test_processed)
accuracy_knn = accuracy_score(y_test, y_pred_knn)
```

```
# Decision Tree Accuracy
print("Decision Tree Accuracy:", accuracy_tree)

# KNN Accuracy
print("KNN Accuracy:", accuracy_knn)
```

```
Decision Tree Accuracy: 0.7912087912087912
KNN Accuracy: 0.8571428571428571
```

Categorical Preprocessing Only

Create a new preprocessing pipeline which ONLY preprocesses categorical values (leaving scalar variables in the data as they were originally, ie. no StandardScaler).
Process your data with this new pipeline, fit a decision tree and a KNN once more and report a new accuracy score for each.

Hint: Ensure that remainder = 'passthrough' in your ColumnTransformer to ensure scalar values are not dropped!

```
categorical_features = ['sex', 'chest_pain', 'high_fbs', 'r_ecg', 'ex_ang',
'stress_slope', 'num_vessels', 'thal_test_res']
numeric_features = ['age', 'rpb', 'chol', 'hr_max', 'ecg_depress']

# Create a preprocessor for only categorical features
categorical_preprocessor = ColumnTransformer(
    transformers=[
        ('cat', Pipeline([('onehot', OneHotEncoder(sparse_output=False,
handle_unknown='ignore'))]), categorical_features)
    ],
    remainder='passthrough'  # Passthrough numeric features without scaling
)

# Process the training data with both preprocessors
X_train_cat_processed = categorical_preprocessor.fit_transform(X_train_raw)
```

```
# Process the testing data with both preprocessors
X_test_cat_processed = categorical_preprocessor.transform(X_test_raw)
```

```
# Fit Decision Tree with categorical preprocessing only
tree_cat = DecisionTreeClassifier()
tree_cat.fit(X_train_cat_processed, y_train)

# Fit KNN with categorical preprocessing
knn_cat = KNeighborsClassifier()
knn_cat.fit(X_train_cat_processed, y_train)
```

```
KNeighborsClassifier()
```

```
# Make predictions and calculate accuracy for each model
y_pred_tree_cat = tree_cat.predict(X_test_cat_processed)
y_pred_knn_cat = knn_cat.predict(X_test_cat_processed)

accuracy_tree_cat = accuracy_score(y_test, y_pred_tree_cat)
accuracy_knn_cat = accuracy_score(y_test, y_pred_knn_cat)

print("Decision Tree Accuracy (Categorical Preprocessing Only):",
accuracy_tree_cat)
print("KNN Accuracy (Categorical Preprocessing Only):", accuracy_knn_cat)
```

```
Decision Tree Accuracy (Categorical Preprocessing Only): 0.7802197802197802
KNN Accuracy (Categorical Preprocessing Only): 0.6593406593406593
```

Explain the difference in accuracy loss in Decision Trees vs KNNs when Standardization was removed.

Decision Trees: Accuracy loss was minimal (1.4%). It is less sensitive to feature scaling. Splitting criteria based on relative order of feature values, not absolute magnitudes. Standardization has minimal impact on this order.

KNNs: Accuracy loss was significant (23.1%). It is highly sensitive to feature scaling. Relies on distance calculations, which are distorted by unscaled features. This effect is amplified with smaller n_neighbors, leading to misclassifications.

Printing Jupyter notebook to PDF (Google Colab Only, Optional)

It may take a few minutes to run