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

Name: Tae Hwan Kim

UID: 506043010

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

In [209…

```
#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.)

In [210…
```
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.

In [211…
```
data.head()
```

Out[211…

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

In [212…
```
data.describe()
```

Out[212…

	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	3
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	

In [213…
```
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())

In [214…

```python
data['heart_disease_numeric'] = data['heart_disease'].astype(int)
data.drop(columns=['heart_disease'], inplace=True)
```

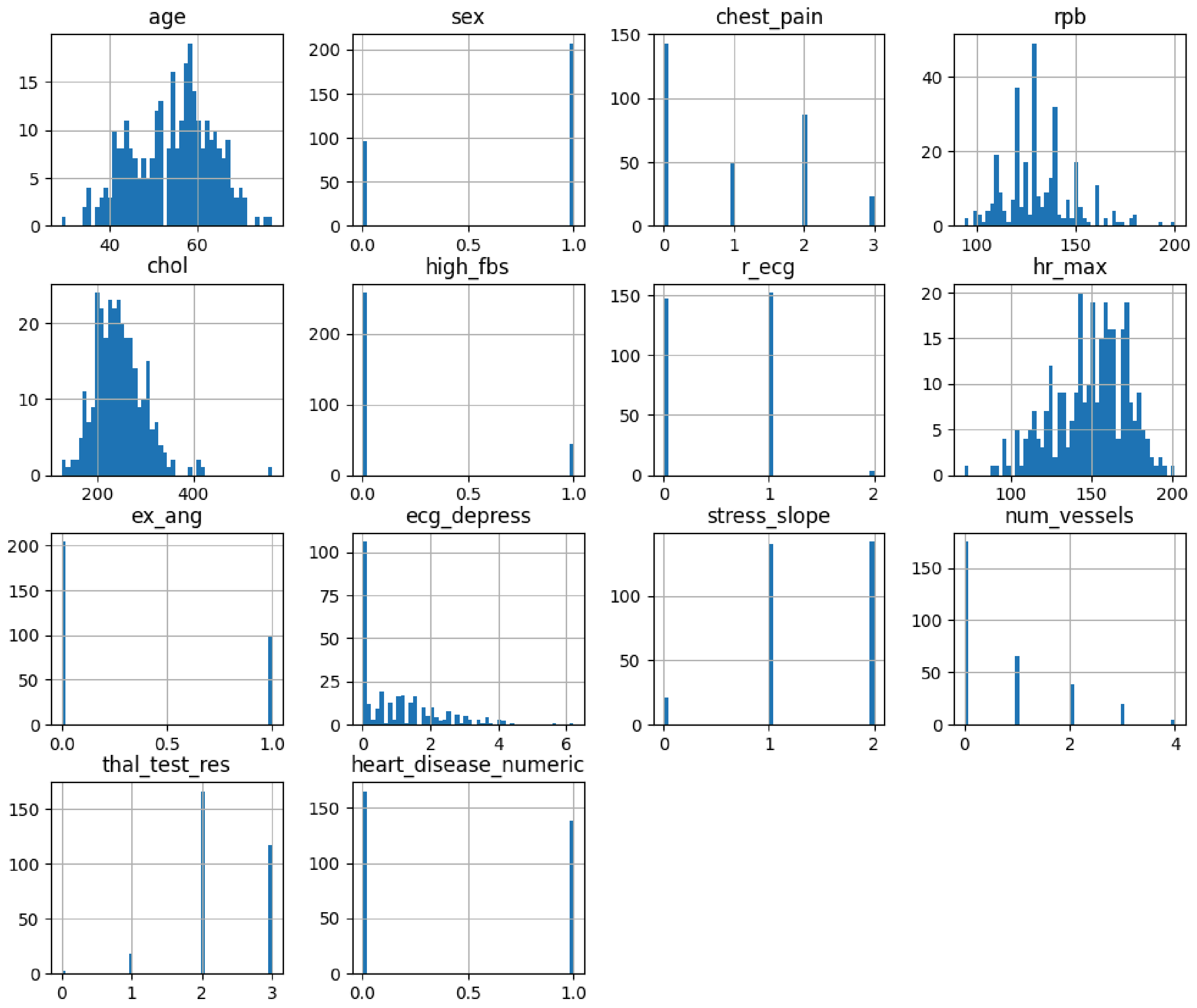
Now that we have a feel for the data-types for each of the variables, plot histograms of each field and attempt to ascertain how each variable performs (is it a binary, or limited selection, or does it follow a gradient?

In [215…

```python
data.hist(figsize=(12, 10), bins=50)
```

Out[215…

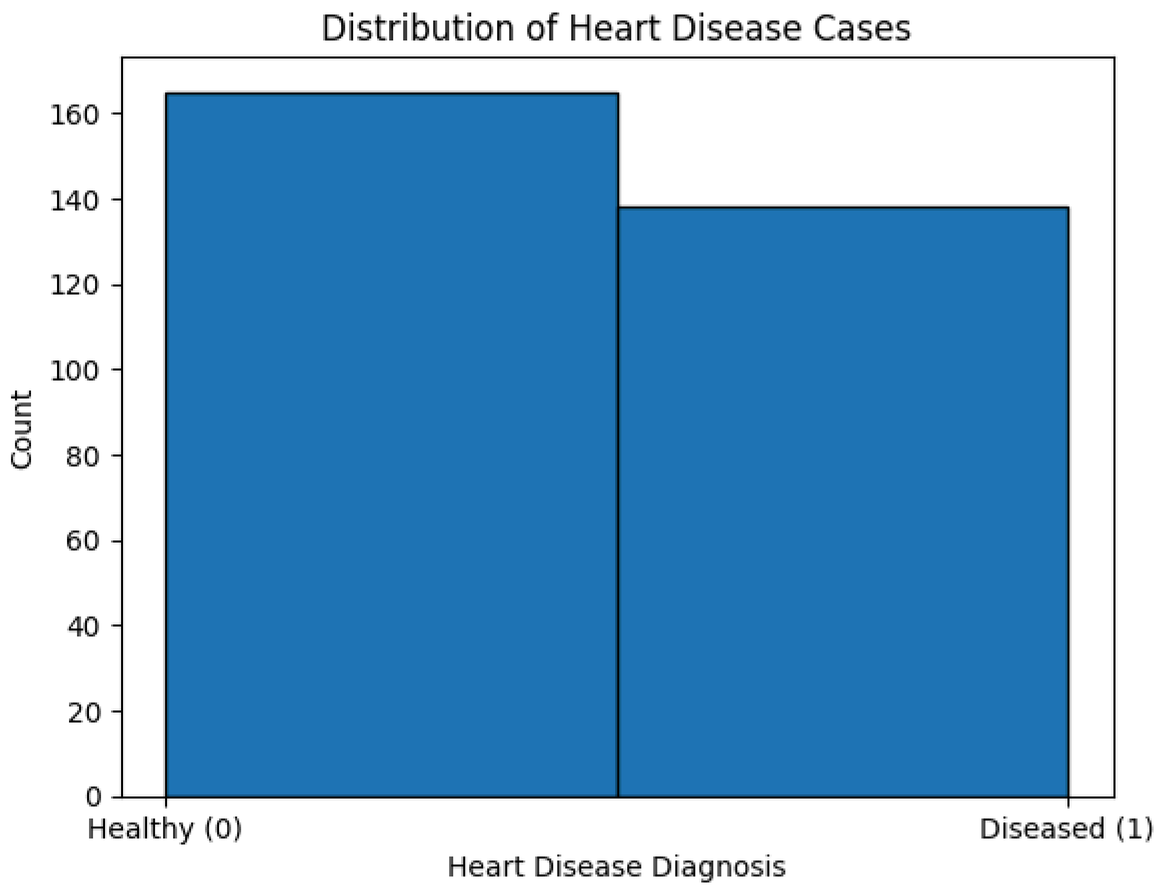
```
array([[<Axes: title={'center': 'age'}>, <Axes: title={'center': 'sex'}>,
        <Axes: title={'center': 'chest_pain'}>,
        <Axes: title={'center': 'rpb'}>],
       [<Axes: title={'center': 'chol'}>,
        <Axes: title={'center': 'high_fbs'}>,
        <Axes: title={'center': 'r_ecg'}>,
        <Axes: title={'center': 'hr_max'}>],
       [<Axes: title={'center': 'ex_ang'}>,
        <Axes: title={'center': 'ecg_depress'}>,
        <Axes: title={'center': 'stress_slope'}>,
        <Axes: title={'center': 'num_vessels'}>],
       [<Axes: title={'center': 'thal_test_res'}>,
        <Axes: title={'center': 'heart_disease_numeric'}>, <Axes: >,
        <Axes: >]], dtype=object)
```



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:

In [216…

```
plt.hist(data['heart_disease_numeric'], bins=2, edgecolor='black', align='mid')
plt.xticks([0, 1], ['Healthy (0)', 'Diseased (1)'])
plt.xlabel('Heart Disease Diagnosis')
plt.ylabel('Count')
plt.title('Distribution of Heart Disease Cases')
plt.show()

count_values = data['heart_disease_numeric'].value_counts()
healthy_count = count_values[0]
diseased_count = count_values[1]

print(f"Healthy individuals (0): {healthy_count}")
print(f"Diseased individuals (1): {diseased_count}")
```

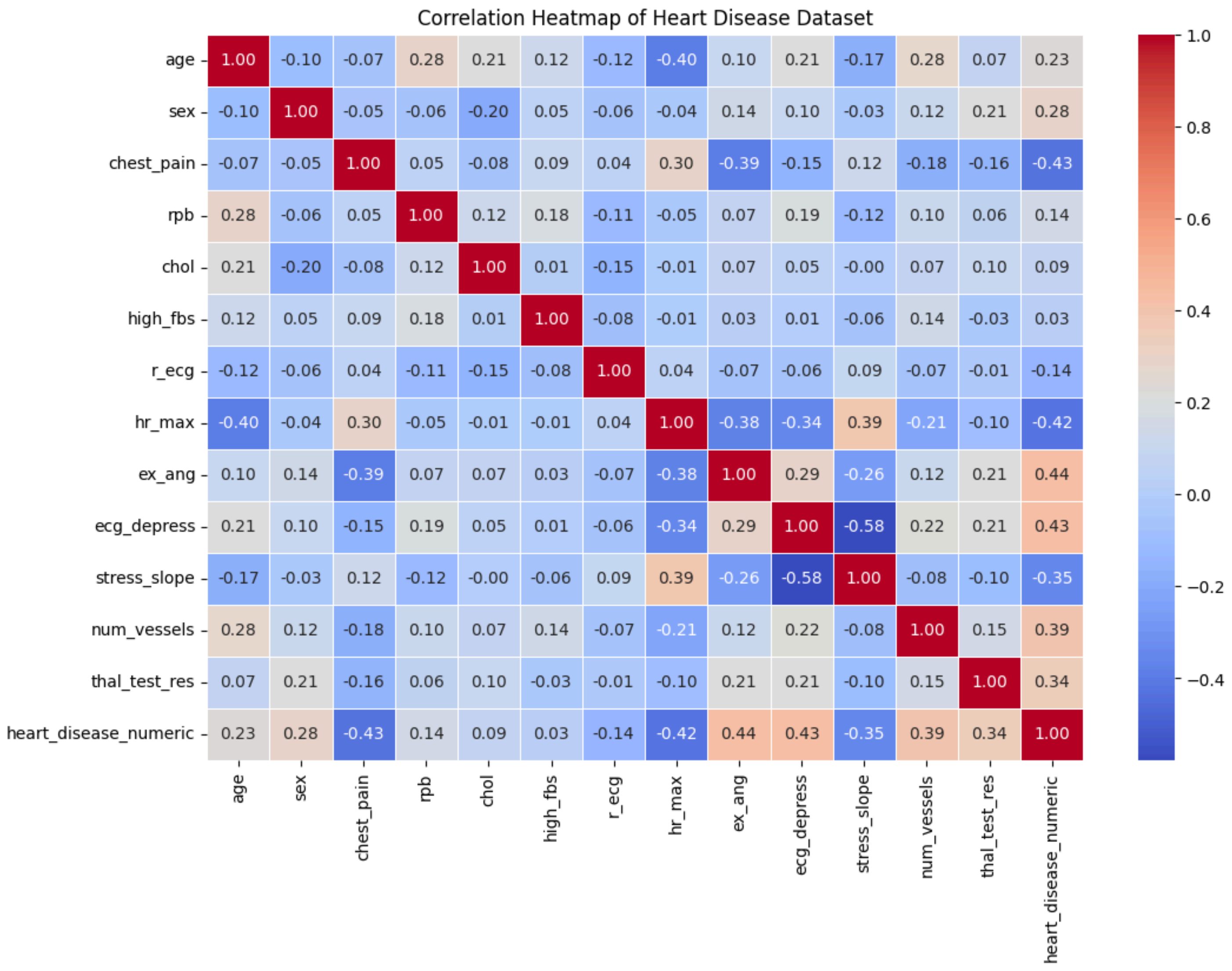


```
Healthy individuals (0): 165
Diseased individuals (1): 138
```

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 [217…

```
corr_matrix = data.corr()

plt.figure(figsize=(12, 8))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt=".2f", linewidths=0.5)
plt.title("Correlation Heatmap of Heart Disease Dataset")
plt.show()
```



The correlation heatmap shows that ex_ang, ecg_depress, num_vessels, and thal_test_res have strong positive correlations with heart disease. Conversely, hr_max, chest_pain, and stress_slope have strong negative correlations with heart disease. Other variables such as chol and rpb show relatively weak correlations. Some variables correlate more strongly with heart disease because they are direct indicators that directly measure heart function. Others are indirect risk factors that contribute to heart disease over time but don’t immediately indicate its presence. Additionally, synergy effects and hidden confounders can amplify correlations when multiple risk factors interact, making some variables appear strongerly correlate than others.

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.

In [218…

```
y = data['heart_disease_numeric'].values
X = data.drop(columns=['heart_disease_numeric'])
X.head()
```

Out[218…

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

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

In [219…

```
X_train_raw, X_test_raw, y_train_raw, y_test_raw = train_test_split(X, y, test_size=0.3, random_
```

In [220…

```
print("Training set shape (X_train_raw):", X_train_raw.shape)
print("Testing set shape (X_test_raw):", X_test_raw.shape)
print("Training labels shape (y_train_raw):", y_train_raw.shape)
print("Testing labels shape (y_test_raw):", y_test_raw.shape)
```

```
Training set shape (X_train_raw): (212, 13)
Testing set shape (X_test_raw): (91, 13)
Training labels shape (y_train_raw): (212,)
Testing labels shape (y_test_raw): (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.

In [221…

```
knn = KNeighborsClassifier()
knn.fit(X_train_raw, y_train_raw)
y_pred_raw = knn.predict(X_test_raw)
```

In [222…

```
correct_predictions = sum(y_pred_raw == y_test_raw)
accuracy_raw = correct_predictions / len(y_test_raw)
print(f"Accuracy of KNN on raw data: {accuracy_raw:.4f}")
```

```
Accuracy of KNN on raw data: 0.6813
```

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:

In [223…
```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer, make_column_transformer
```

In [224…
```
# Create pipelines
numeric_features = ['age', 'rpb', 'chol', 'hr_max', 'ecg_depress']
categorical_features = ['sex', 'chest_pain', 'high_fbs', 'r_ecg', 'ex_ang', 'stress_slope', 'num_

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

categorical_transformer = Pipeline([
    ('encoder', OneHotEncoder(handle_unknown='ignore'))
])

preprocessor = ColumnTransformer([
    ('num', numeric_transformer, numeric_features),
    ('cat', categorical_transformer, categorical_features)
])
```

In [225…
```
# Pipeline the training and test data
X_train_preprocessed = preprocessor.fit_transform(X_train_raw)
X_test_preprocessed = preprocessor.transform(X_test_raw)

print("Preprocessed Training set shape:", X_train_preprocessed.shape)
print("Preprocessed Testing set shape:", X_test_preprocessed.shape)
```

```
Preprocessed Training set shape: (212, 30)
Preprocessed Testing set shape: (91, 30)
```

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

In [226…
```
knn_processed = KNeighborsClassifier()
knn_processed.fit(X_train_preprocessed, y_train_raw)
y_pred_processed = knn_processed.predict(X_test_preprocessed)
```

In [227…
```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
# Report Metrics

# Metrics for KNN on Raw Data
```

```
accuracy_raw = accuracy_score(y_test_raw, y_pred_raw)
precision_raw = precision_score(y_test_raw, y_pred_raw, zero_division=1)
recall_raw = recall_score(y_test_raw, y_pred_raw, zero_division=1)
f1_raw = f1_score(y_test_raw, y_pred_raw, zero_division=1)

print("\nMetrics for KNN (Raw Data):")
print(f"Accuracy:  {accuracy_raw:.4f}")
print(f"Precision: {precision_raw:.4f}")
print(f"Recall:    {recall_raw:.4f}")
print(f"F1 Score:  {f1_raw:.4f}")

# Metrics for KNN on Preprocessed Data
accuracy_processed = accuracy_score(y_test_raw, y_pred_processed)
precision_processed = precision_score(y_test_raw, y_pred_processed, zero_division=1)
recall_processed = recall_score(y_test_raw, y_pred_processed, zero_division=1)
f1_processed = f1_score(y_test_raw, y_pred_processed, zero_division=1)

print("\nMetrics for KNN (Preprocessed Data):")
print(f"Accuracy:  {accuracy_processed:.4f}")
print(f"Precision: {precision_processed:.4f}")
print(f"Recall:    {recall_processed:.4f}")
print(f"F1 Score:  {f1_processed:.4f}")
```

```
Metrics for KNN (Raw Data):
Accuracy:  0.6813
Precision: 0.6875
Recall:    0.5366
F1 Score:  0.6027

Metrics for KNN (Preprocessed Data):
Accuracy:  0.8352
Precision: 0.8421
Recall:    0.7805
F1 Score:  0.8101
```

The preprocessed KNN model performed better than the raw data model with accuracy increasing from 68.13% to 83.52% and F1 score improving from 0.60 to 0.81. This suggests that scaling numeric features and encoding categorical variables helps KNN make more accurate classifications and 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).

In [228…

```
neighbors_list = [1, 2, 3, 5, 7, 9, 10, 20, 50]

accuracy_results = {}

for n in neighbors_list:
    knn = KNeighborsClassifier(n_neighbors=n)
    knn.fit(X_train_preprocessed, y_train_raw)
    y_pred = knn.predict(X_test_preprocessed)

    accuracy = accuracy_score(y_test_raw, y_pred)
    accuracy_results[n] = accuracy
    print(f"n_neighbors = {n}, Accuracy: {accuracy:.4f}")
```

```
n_neighbors = 1, Accuracy: 0.7582
n_neighbors = 2, Accuracy: 0.7692
n_neighbors = 3, Accuracy: 0.8242
n_neighbors = 5, Accuracy: 0.8352
n_neighbors = 7, Accuracy: 0.8022
n_neighbors = 9, Accuracy: 0.8352
n_neighbors = 10, Accuracy: 0.8352
n_neighbors = 20, Accuracy: 0.8022
n_neighbors = 50, Accuracy: 0.7692
```

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

In [229…

```
# Logistic Regression
log_reg = LogisticRegression(max_iter=1000)
log_reg.fit(X_train_preprocessed, y_train_raw)
y_pred_log_reg = log_reg.predict(X_test_preprocessed)

accuracy_log_reg = accuracy_score(y_test_raw, y_pred_log_reg)
precision_log_reg = precision_score(y_test_raw, y_pred_log_reg, zero_division=1)
recall_log_reg = recall_score(y_test_raw, y_pred_log_reg, zero_division=1)
f1_log_reg = f1_score(y_test_raw, y_pred_log_reg, zero_division=1)

print("\nMetrics for Logistic Regression:")
print(f"Accuracy:  {accuracy_log_reg:.4f}")
print(f"Precision: {precision_log_reg:.4f}")
print(f"Recall:    {recall_log_reg:.4f}")
print(f"F1 Score:  {f1_log_reg:.4f}")
```

```
Metrics for Logistic Regression:
Accuracy:  0.8352
Precision: 0.8250
Recall:    0.8049
F1 Score:  0.8148
```

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 measures the proportion of correct predictions out of all predictions. It is useful when both classes are well-balanced. However, in imbalanced datasets, accuracy can be misleading because a model might predict the majority class most of the time and still appear highly accurate. Precision focuses on how many of the predicted positive cases are actually correct, which is important when false positives have serious consequences. For example, in fraud detection, precision is important because incorrectly flagging legitimate transactions as fraud can cause inconvenience for customers. Recall measures how many actual positive cases were correctly identified. It is important in medical diagnosis, where missing a true case can be life-threatening. F1 Score balances precision and recall. It is ideal for scenarios like spam detection, where both false positives and false negatives are problematic. The reason we might choose to evaluate the performance of differing models using different metrics is that the cost of errors varies depending on the application.

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!

In [230…

```python
log_reg_modified = LogisticRegression(solver='sag', max_iter=10, penalty=None)
log_reg_modified.fit(X_train_preprocessed, y_train_raw)
y_pred_log_reg_modified = log_reg_modified.predict(X_test_preprocessed)

accuracy_mod = accuracy_score(y_test_raw, y_pred_log_reg_modified)
precision_mod = precision_score(y_test_raw, y_pred_log_reg_modified, zero_division=1)
recall_mod = recall_score(y_test_raw, y_pred_log_reg_modified, zero_division=1)
f1_mod = f1_score(y_test_raw, y_pred_log_reg_modified, zero_division=1)

# Print results
print("\nMetrics for Logistic Regression (Modified Settings):")
print(f"Accuracy:  {accuracy_mod:.4f}")
print(f"Precision: {precision_mod:.4f}")
print(f"Recall:    {recall_mod:.4f}")
print(f"F1 Score:  {f1_mod:.4f}")
```

```
Metrics for Logistic Regression (Modified Settings):
Accuracy:  0.8352
Precision: 0.8611
Recall:    0.7561
F1 Score:  0.8052
```

```
c:\Users\TAE HWAN KIM\AppData\Local\Programs\Python\Python313\Lib\site-packages\sklearn\linear_mo
del\_sag.py:348: ConvergenceWarning: The max_iter was reached which means the coef_ did not conve
rge
  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.

In [231…

```
log_reg_modified = LogisticRegression(solver='sag', max_iter=3000, penalty=None)
log_reg_modified.fit(X_train_preprocessed, y_train_raw)
y_pred_log_reg_modified = log_reg_modified.predict(X_test_preprocessed)

accuracy_mod = accuracy_score(y_test_raw, y_pred_log_reg_modified)
precision_mod = precision_score(y_test_raw, y_pred_log_reg_modified, zero_division=1)
recall_mod = recall_score(y_test_raw, y_pred_log_reg_modified, zero_division=1)
f1_mod = f1_score(y_test_raw, y_pred_log_reg_modified, zero_division=1)

# Print results
print("\nMetrics for Logistic Regression (Modified Settings):")
print(f"Accuracy:  {accuracy_mod:.4f}")
print(f"Precision: {precision_mod:.4f}")
print(f"Recall:    {recall_mod:.4f}")
print(f"F1 Score:  {f1_mod:.4f}")
```

```
Metrics for Logistic Regression (Modified Settings):
Accuracy:  0.8242
Precision: 0.8378
Recall:    0.7561
F1 Score:  0.7949
```

Explain what you changed, and why do you think that may have altered the outcome.

I increased the max_iter parameter from 10 to 3000 in the logistic regression model. The setting with max_iter=10 was too low and caused the optimization process to stop before the model learn the correct weights. By increasing max_iter, the solver would perform more iterations and had enough steps to properly adjust the coefficients and reach convergence.

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

In [232…

```
log_reg_modified = LogisticRegression(solver='liblinear', max_iter=3000, penalty='l1')
log_reg_modified.fit(X_train_preprocessed, y_train_raw)
y_pred_log_reg_modified = log_reg_modified.predict(X_test_preprocessed)

accuracy_mod = accuracy_score(y_test_raw, y_pred_log_reg_modified)
precision_mod = precision_score(y_test_raw, y_pred_log_reg_modified, zero_division=1)
recall_mod = recall_score(y_test_raw, y_pred_log_reg_modified, zero_division=1)
f1_mod = f1_score(y_test_raw, y_pred_log_reg_modified, zero_division=1)

# Print results
print("\nMetrics for Logistic Regression (Modified Settings):")
print(f"Accuracy:  {accuracy_mod:.4f}")
print(f"Precision: {precision_mod:.4f}")
print(f"Recall:    {recall_mod:.4f}")
print(f"F1 Score:  {f1_mod:.4f}")
```

```
Metrics for Logistic Regression (Modified Settings):
Accuracy:  0.8462
Precision: 0.8649
Recall:    0.7805
F1 Score:  0.8205
```

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!).

The sag solver is an optimization algorithm based on Stochastic Gradient Descent, which updates model weights iteratively using mini-batches of training data rather than the entire dataset at once. The liblinear solver uses Coordinate Descent, optimizing one weight at a time while keeping the others fixed. The liblinear solver may have produced an improved outcome because it uses coordinate descent, which can converge faster. In addition, since L1 regularization (penalty='l1') was applied, liblinear performed feature selection by setting some coefficients to zero, which may have reduced noise and improved generalization.

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.

In [233…

```python
svm_model = SVC(probability=True)
svm_model.fit(X_train_preprocessed, y_train_raw)
y_pred_svm = svm_model.predict(X_test_preprocessed)
```

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!

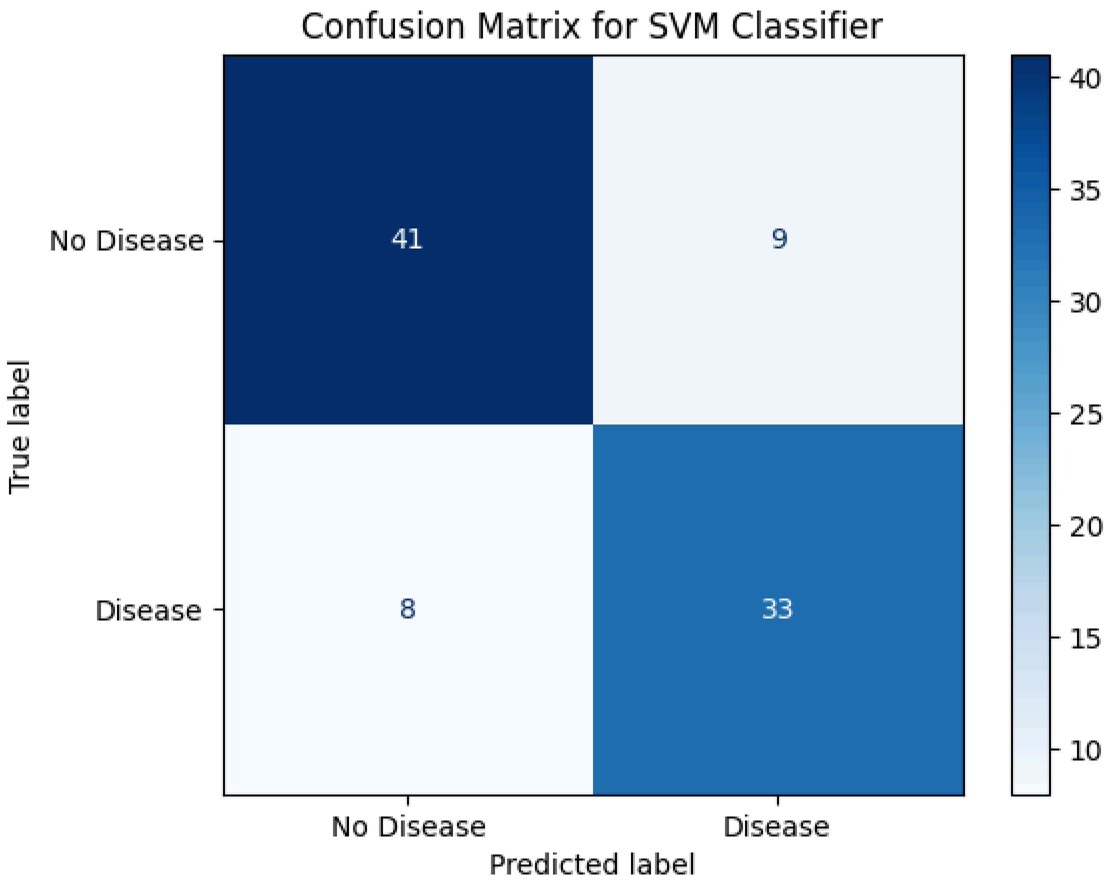
In [234…

```python
# Report Metrics
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, ConfusionMa
import matplotlib.pyplot as plt

accuracy_svm = accuracy_score(y_test_raw, y_pred_svm)
precision_svm = precision_score(y_test_raw, y_pred_svm, zero_division=1)
recall_svm = recall_score(y_test_raw, y_pred_svm, zero_division=1)
f1_svm = f1_score(y_test_raw, y_pred_svm, zero_division=1)

print("\nMetrics for Support Vector Machine (SVM):")
print(f"Accuracy:  {accuracy_svm:.4f}")
print(f"Precision: {precision_svm:.4f}")
print(f"Recall:    {recall_svm:.4f}")
print(f"F1 Score:  {f1_svm:.4f}")
```

```
Metrics for Support Vector Machine (SVM):
Accuracy:  0.8132
Precision: 0.7857
Recall:    0.8049
F1 Score:  0.7952
```

In [235…

```
# Confusion Matrix
conf_matrix = confusion_matrix(y_test_raw, y_pred_svm)

disp = ConfusionMatrixDisplay(confusion_matrix=conf_matrix, display_labels=["No Disease", "Disea
disp.plot(cmap="Blues", values_format="d")
plt.title("Confusion Matrix for SVM Classifier")
plt.show()
```



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

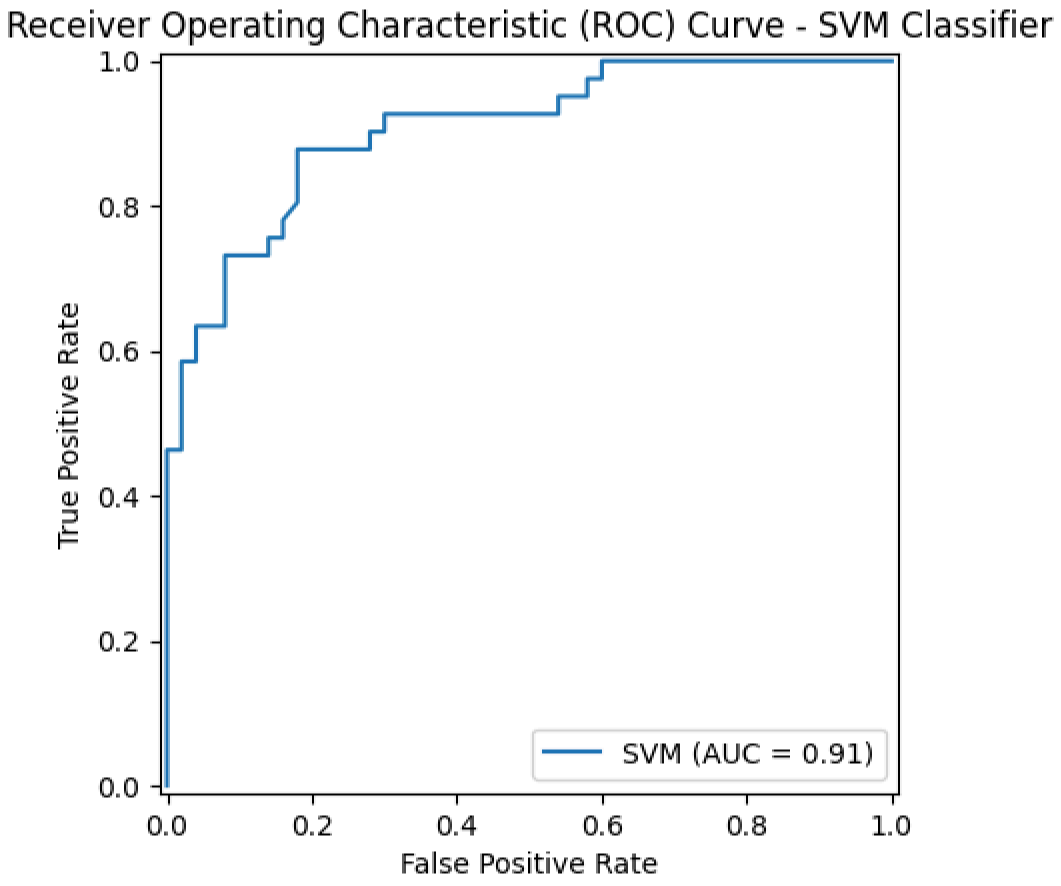
recommend using the `metrics.roc_curve` `metrics.auc` and `metrics.RocCurveDisplay` for this one!

In [236…

```
# ROC

from sklearn.metrics import roc_curve, auc, RocCurveDisplay

y_probs_svm = svm_model.predict_proba(X_test_preprocessed)[:, 1]

fpr, tpr, thresholds = roc_curve(y_test_raw, y_probs_svm)
roc_auc = auc(fpr, tpr)
```

```
# Plot
plt.figure(figsize=(8, 6))
roc_display = RocCurveDisplay(fpr=fpr, tpr=tpr, roc_auc=roc_auc, estimator_name="SVM")
roc_display.plot()
plt.title("Receiver Operating Characteristic (ROC) Curve - SVM Classifier")
plt.show()
```

```
<Figure size 800x600 with 0 Axes>
```



An ROC curve is a representation that illustates the trade-off between the true positive rate and the false positive rate across different classification thresholds. It helps to evaluate a model's ability to distinguish between classes. From the plot above, the ROC curve with an AUC of 0.91 indicates that the SVM classifier is performing very well in distinguishing between individuals with and without heart disease.

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.

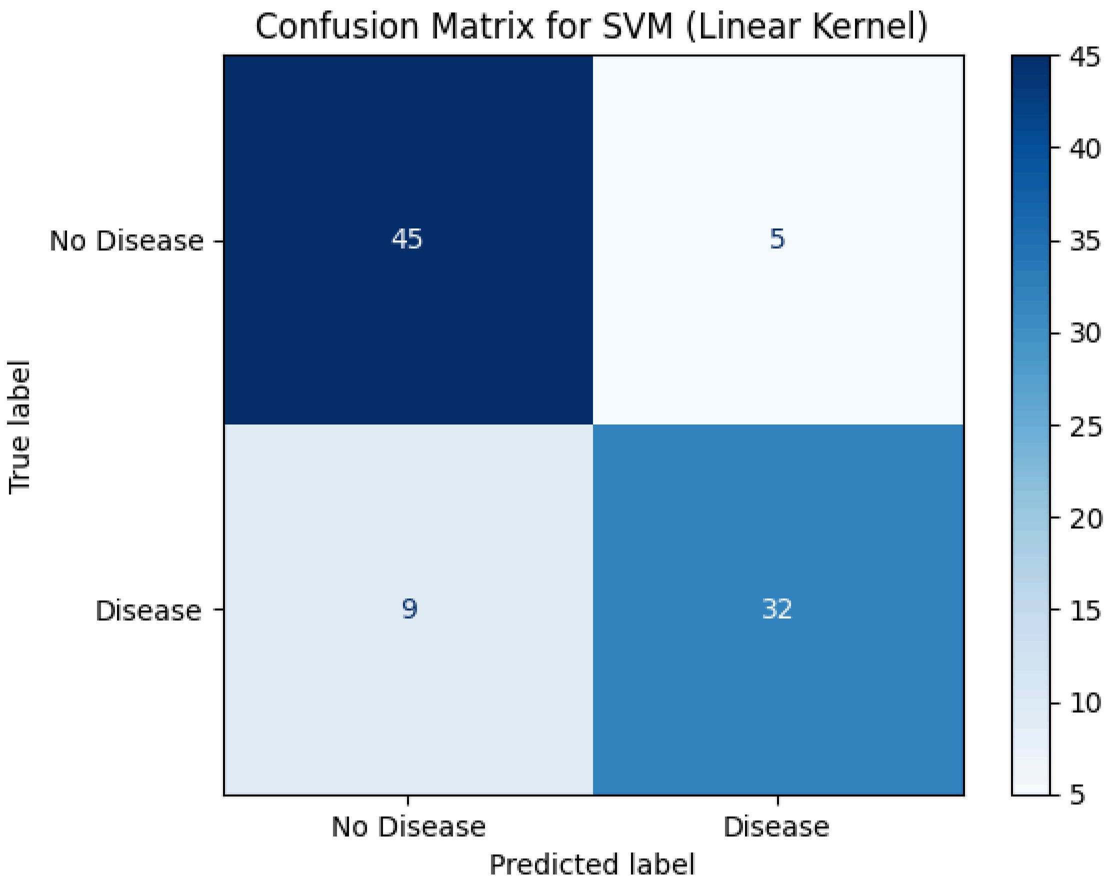
In [237…
```
svm_linear = SVC(kernel='linear', probability=True)
svm_linear.fit(X_train_preprocessed, y_train_raw)
y_pred_svm_linear = svm_linear.predict(X_test_preprocessed)
```

In [238…
```
# Metrics
accuracy_linear = accuracy_score(y_test_raw, y_pred_svm_linear)
precision_linear = precision_score(y_test_raw, y_pred_svm_linear, zero_division=1)
recall_linear = recall_score(y_test_raw, y_pred_svm_linear, zero_division=1)
f1_linear = f1_score(y_test_raw, y_pred_svm_linear, zero_division=1)

# Print the results
print("\nMetrics for SVM (Linear Kernel):")
print(f"Accuracy:  {accuracy_linear:.4f}")
print(f"Precision: {precision_linear:.4f}")
print(f"Recall:    {recall_linear:.4f}")
print(f"F1 Score:  {f1_linear:.4f}")
```

```
Metrics for SVM (Linear Kernel):
Accuracy:  0.8462
Precision: 0.8649
Recall:    0.7805
F1 Score:  0.8205
```

In [239…

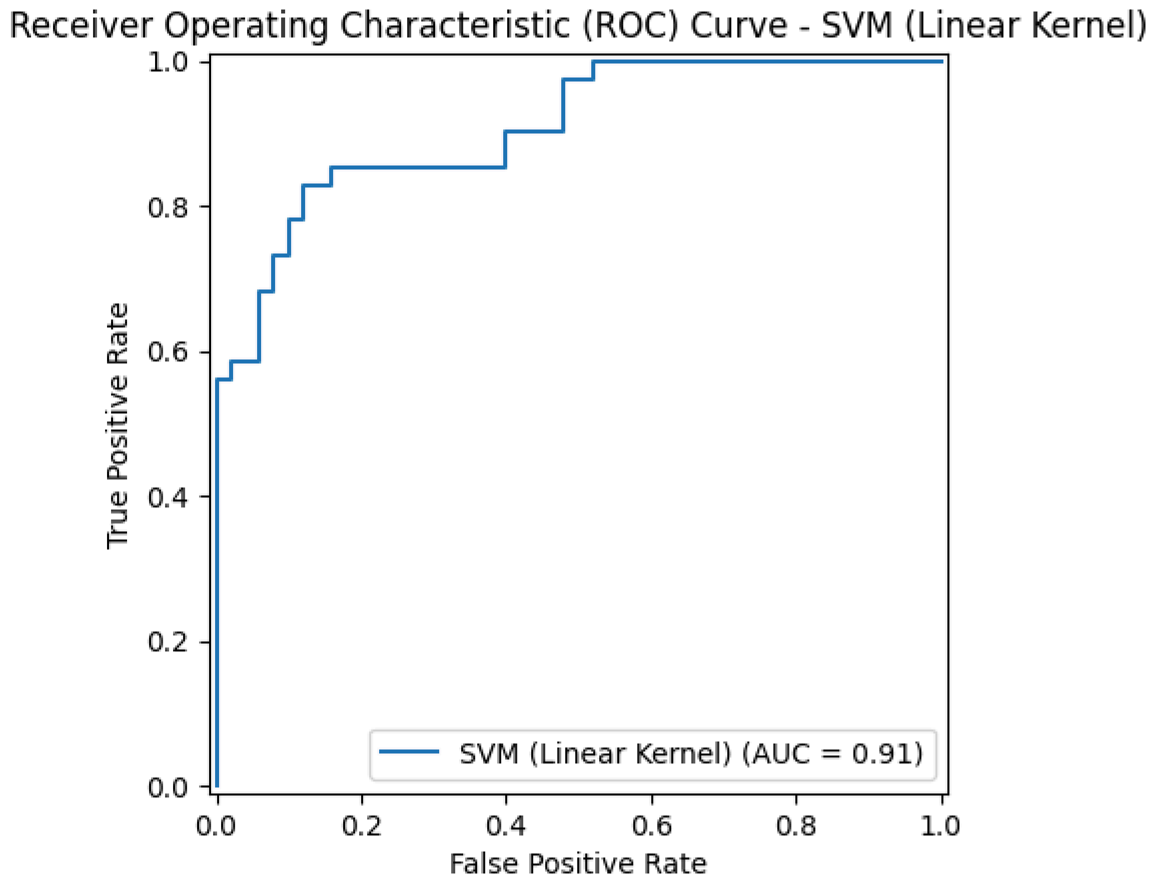
```
# Confusion Matrix
conf_matrix_linear = confusion_matrix(y_test_raw, y_pred_svm_linear)

disp = ConfusionMatrixDisplay(confusion_matrix=conf_matrix_linear, display_labels=["No Disease",
disp.plot(cmap="Blues", values_format="d")
plt.title("Confusion Matrix for SVM (Linear Kernel)")
plt.show()
```



In [240…

```
# ROC
y_probs_svm_linear = svm_linear.predict_proba(X_test_preprocessed)[:, 1]
fpr_linear, tpr_linear, thresholds_linear = roc_curve(y_test_raw, y_probs_svm_linear)
roc_auc_linear = auc(fpr_linear, tpr_linear)

plt.figure(figsize=(8, 6))
roc_display_linear = RocCurveDisplay(fpr=fpr_linear, tpr=tpr_linear, roc_auc=roc_auc_linear, est
roc_display_linear.plot()
plt.title("Receiver Operating Characteristic (ROC) Curve - SVM (Linear Kernel)")
plt.show()
```

```
<Figure size 800x600 with 0 Axes>
```



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 changing the SVM kernel from its default to linear, the model assumes that the data is linearly separable and attempts to find a linear decision boundary. Since the linear kernel improved overall accuracy and precision, the model became more confident in its positive predictions while slightly sacrificing recall. This suggests that the dataset is at least partially linearly separable. The linear kernel may be preferable for better interpretability and computational efficiency.

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

Logistic Regression is a probabilistic model that minimizes log loss to estimate the probability of a data point belonging to a class. In contrast, SVM takes a geometric approach to find the line or hyperplane that maximizes the margin between the closest points of each class.

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.

In [241…
```
from sklearn.tree import DecisionTreeClassifier
# Decision Tree
decision_tree = DecisionTreeClassifier(random_state=42)
decision_tree.fit(X_train_preprocessed, y_train_raw)
y_pred_tree = decision_tree.predict(X_test_preprocessed)

# KNN
knn = KNeighborsClassifier()
knn.fit(X_train_preprocessed, y_train_raw)
y_pred_knn = knn.predict(X_test_preprocessed)
```

In [242…
```
# Decision Tree Accuracy
accuracy_tree = accuracy_score(y_test_raw, y_pred_tree)
print(f"Decision Tree Accuracy: {accuracy_tree:.4f}")

# KNN Accuracy
accuracy_knn = accuracy_score(y_test_raw, y_pred_knn)
print(f"KNN Accuracy: {accuracy_knn:.4f}")
```

```
Decision Tree Accuracy: 0.7253
KNN Accuracy: 0.8352
```

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!

In [243…
```
# Categorical Preprocessing Only
categorical_transformer = Pipeline([
    ('encoder', OneHotEncoder(handle_unknown='ignore'))
])

preprocessor_cat_only = ColumnTransformer([
    ('cat', categorical_transformer, categorical_features)
], remainder='passthrough')

X_train_cat_only = preprocessor_cat_only.fit_transform(X_train_raw)
X_test_cat_only = preprocessor_cat_only.transform(X_test_raw)
```

In [244…
```
# Fit Decision Tree
decision_tree_cat = DecisionTreeClassifier(random_state=42)
decision_tree_cat.fit(X_train_cat_only, y_train_raw)
```

```
y_pred_tree_cat = decision_tree_cat.predict(X_test_cat_only)

# Fit KNN
knn_cat = KNeighborsClassifier()
knn_cat.fit(X_train_cat_only, y_train_raw)
y_pred_knn_cat = knn_cat.predict(X_test_cat_only)
```

In [245…

```
# Decision Tree Accuracy
accuracy_tree_cat = accuracy_score(y_test_raw, y_pred_tree_cat)
print(f"Decision Tree Accuracy (Categorical Only): {accuracy_tree_cat:.4f}")

# KNN Accuracy
accuracy_knn_cat = accuracy_score(y_test_raw, y_pred_knn_cat)
print(f"KNN Accuracy (Categorical Only): {accuracy_knn_cat:.4f}")
```

```
Decision Tree Accuracy (Categorical Only): 0.7143
KNN Accuracy (Categorical Only): 0.6813
```

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

After applying only categorical preprocessing without standardizing numeric features, Decision Tree accuracy dropped slightly, while KNN accuracy dropped significantly. This difference occurs because Decision Trees make decisions by splitting data, so they don't need numbers to be on the same scale. On the other hand, KNN relies on distance calculations, and without scaling, larger numbers had too much influence, making the model less accurate. Without standardization, KNN may struggle to classify correctly because the distances gets distorted.