University of Moratuwa

Faculty of Engineering

Department of Electronic & Telecommunication Engineering

EN3150 -Pattern Recognition

Assignment 02

Learning from data and related challenges and classification

210031H   A.A.W.L.R.Amarasinghe

1 Logistic Regression

Question 1

2. What is the purpose of "y_encoded = le.fit_transform(df_filtered['species'])" ?

Convert the categorical labels of the species column into numeric format because logistic regressio needs numerical input for the target variable

3. What is the purpose of "X = df.drop(['species', 'island', 'sex'], axis=1)" ?

Excluding 'species', 'island', 'sex' columns that are not used in the modeling process.
'species' is the target variable (not a feature)
'island' and 'sex' are categorical variables, need to be encoded into numerical form.
they could interfere with the model's ability to process numerical data correctly

4. Why we cannot use "island" and "sex" features?

Logistic regression, expect numerical input. Categorical variables like 'island' and 'sex' need to be converted to a numerical format if we need to use.

6. What is the usage of "random_state=42" ?

The random_state=42 parameter ensures reproducibility by setting a fixed seed for the random number generator. This means that functions involving randomness (like data splitting or model training) produce the same results each time when we run the code, allowing for consistent and comparable results.

7. Why is accuracy low? Accuracy: 0.5813953488372093

- Model Complexity: Logistic regression is a relatively simple model. If the relationship between the features and the target variable is complex, a more sophisticated model is needed.
- Imbalanced Classes: If the dataset has a disproportionate number of examples for each class, the model might perform poorly. Accuracy might be misleading in such cases.
- Inadequate Feature Engineering: The features used might not capture enough information for the model to make accurate predictions

Why does the saga solver perform poorly?

- Solver Suitability: The saga solver is generally effective for large-scale problems and supports both L1 and L2 regularization. However, its performance vary depending on the nature of the data and the problem.

- Convergence and Hyperparameters: The saga solver has several hyperparameters. Since the parameters are not set optimally, it could lead to suboptimal model performance.

8. What is the classification accuracy with liblinear configuration?

Accuracy: 1.0

9. Why does the "liblinear" solver perform better than "saga" solver ?

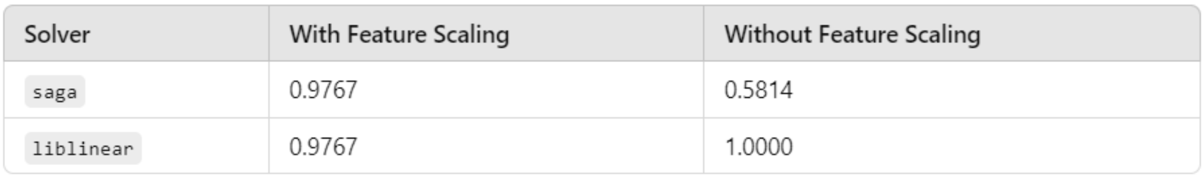
Solver Suitability:

- **liblinear:** This is well-suited for small to medium-sized datasets and performs well with binary and multiclass classification tasks, especially when the dataset is not too large or complex.
- **saga:** This is designed for large-scale datasets and supports both L1 and L2 regularization, it might not be as effective on smaller or simpler datasets.

Convergence and Optimization:

- **liblinear**: Uses a coordinate descent algorithm that is effective for problems with a smaller number of features or when the dataset is relatively small.
- **saga**: This might struggle with the convergence and require more iterations to find the optimal solution.

10. Compare the performance of the "liblinear" and "saga" solvers with feature scaling.



Solver	With Feature Scaling	Without Feature Scaling
`saga`	0.9767	0.5814
`liblinear`	0.9767	1.0000

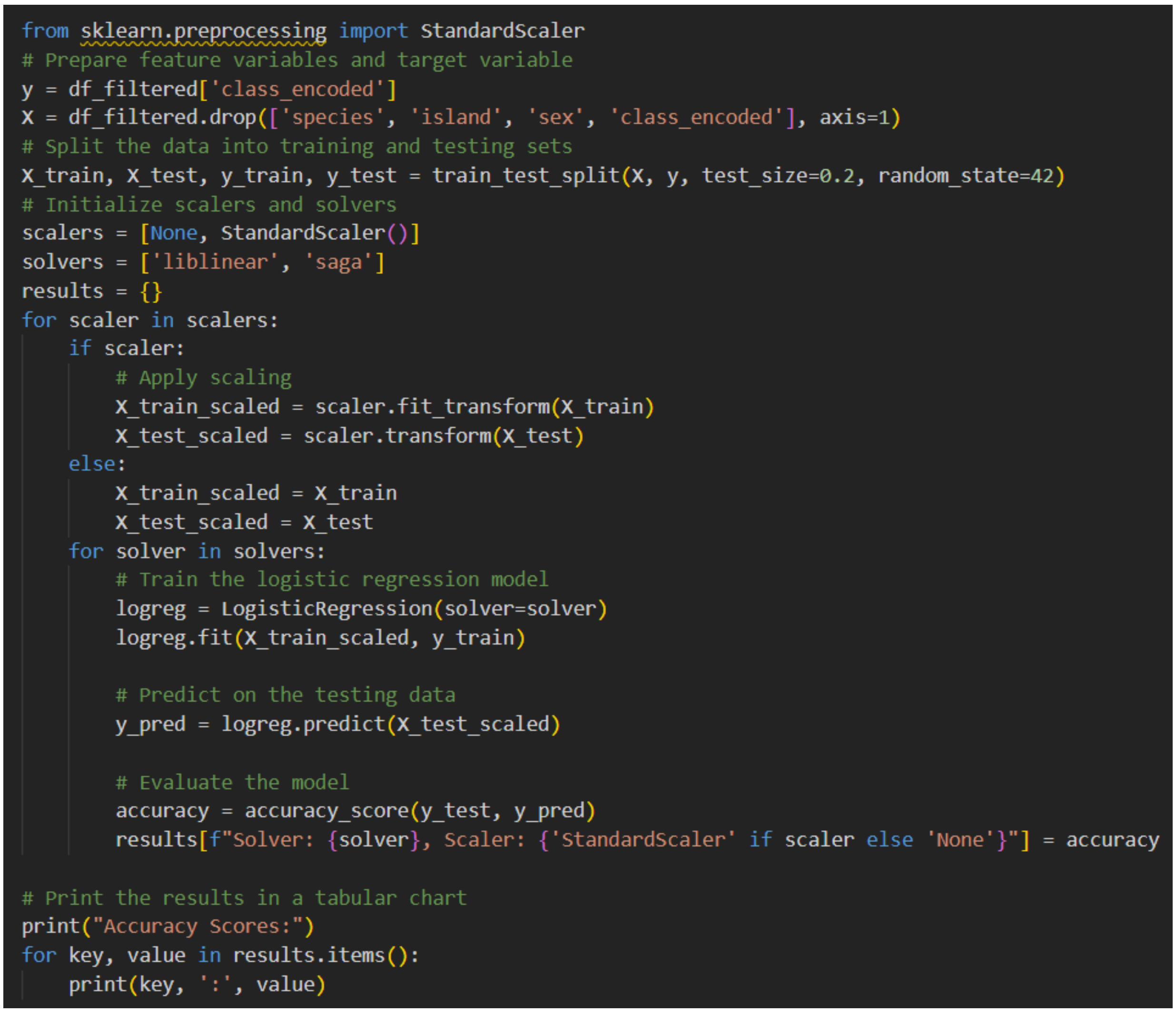
If there is a significant difference in the accuracy with and without feature scaling, what is the reason for that.

Solver Sensitivity:

- **saga** : Sensitive to feature scaling due to its optimization approach. It uses stochastic gradient descent, which benefits from features being on the same scale to ensure stable and effective gradient updates.
- **liblinear** : Less sensitive to feature scaling as it uses a coordinate descent algorithm which might handle varying feature scales better or converge effectively regardless of feature scaling.

Optimization and Convergence:

- Scaling Benefits: Feature scaling normalizes the range of features, making the optimization process more stable. For saga, this helps in achieving faster and more reliable convergence.
- Unscaled Features: For saga, unscaled features can lead to skewed gradients, making the optimization less efficient and potentially causing the model to converge to suboptimal solutions.

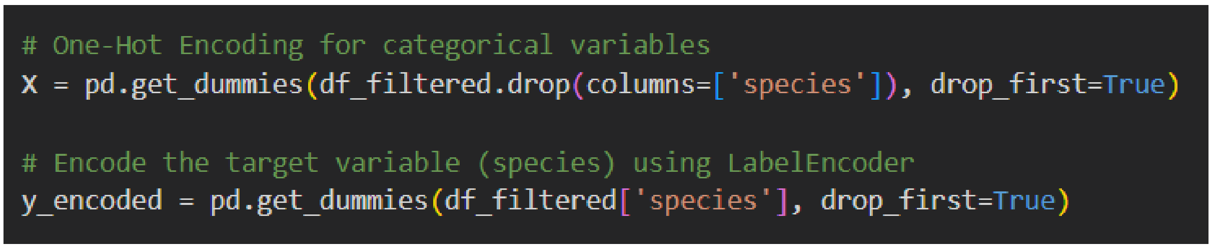


```python
from sklearn.preprocessing import StandardScaler
# Prepare feature variables and target variable
y = df_filtered['class_encoded']
X = df_filtered.drop(['species', 'island', 'sex', 'class_encoded'], axis=1)
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
# Initialize scalers and solvers
scalers = [None, StandardScaler()]
solvers = ['liblinear', 'saga']
results = {}
for scaler in scalers:
    if scaler:
        # Apply scaling
        X_train_scaled = scaler.fit_transform(X_train)
        X_test_scaled = scaler.transform(X_test)
    else:
        X_train_scaled = X_train
        X_test_scaled = X_test
    for solver in solvers:
        # Train the logistic regression model
        logreg = LogisticRegression(solver=solver)
        logreg.fit(X_train_scaled, y_train)

        # Predict on the testing data
        y_pred = logreg.predict(X_test_scaled)

        # Evaluate the model
        accuracy = accuracy_score(y_test, y_pred)
        results[f"Solver: {solver}, Scaler: {'StandardScaler' if scaler else 'None'}"] = accuracy

# Print the results in a tabular chart
print("Accuracy Scores:")
for key, value in results.items():
    print(key, ':', value)
```

11. What is the problem of this code and how to solve this?

This code fails because it includes unencoded categorical variables.

Solution- Encode categorial varibales using one hot encoding.



```python
# One-Hot Encoding for categorical variables
X = pd.get_dummies(df_filtered.drop(columns=['species']), drop_first=True)

# Encode the target variable (species) using LabelEncoder
y_encoded = pd.get_dummies(df_filtered['species'], drop_first=True)
```

12. Why Label Encoding Followed by Scaling is Not Ideal?

Misinterpretation of Labels:

- Label encoding assigns integer values to categories (e.g., red=0, blue=1, green=2). These integers imply an ordinal relationship which doesn't actually exist in nominal categorical features.

Scaling Assumptions:

- Feature scaling methods like Standard Scaling and Min-Max Scaling are designed for continuous numeric features. Applying these methods to label-encoded features can lead to misleading results because scaling assumes that the feature values have a meaningful numerical relationship, which isn't true for nominal categories.

What is the Proposed Approach?

One-Hot Encoding:

- Convert categorical variables into binary columns, where each column represents one possible value of the feature.
- Advantages: One-hot encoding ensures that each category is treated independently and does not impose any ordinal relationship. It is a common approach for handling categorical features in machine learning.

Question 2

$$P = \frac{1}{1 + e^{-(w_0 + w_1 x_1 + w_2 x_2)}}$$

① $z = w_0 + w_1 x_1 + w_2 x_2$

$z = -5.9 + 0.06 \times 50 + 1.5 \times 3.6$

$z = 2.5$

$$P = \frac{1}{1 + e^{-2.5}} = 0.924 = 92.4\%$$

② $$0.6 = \frac{1}{1 + e^{-z}}$$

$z = 0.405$

$z = w_0 + w_1 x_1 + w_2 x_2$

$0.405 = -5.9 + 0.06 x_1 + 1.5 \times 3.6$

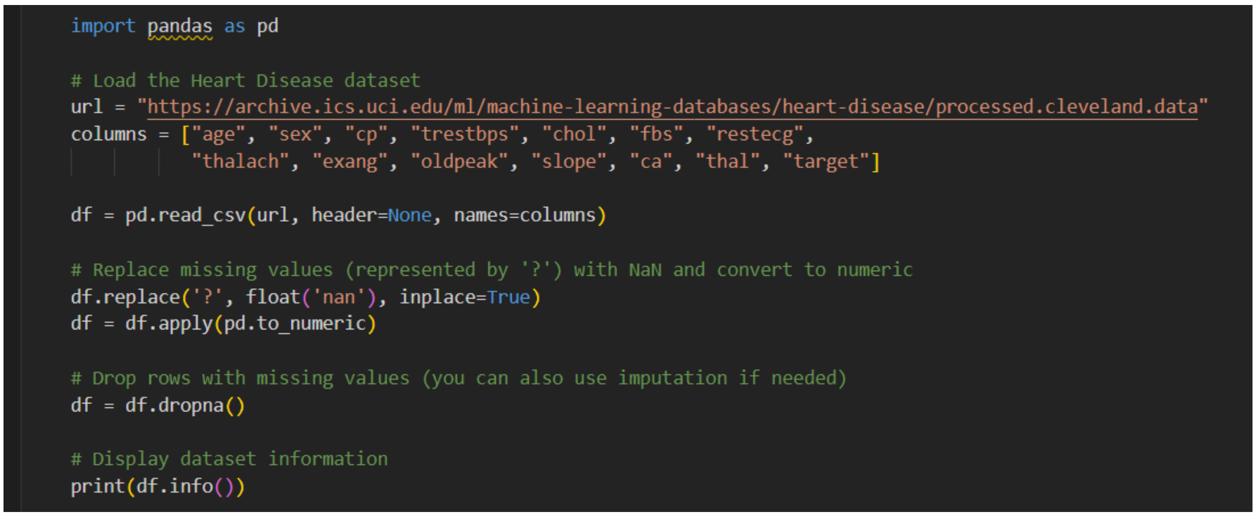
$x_1 = 15.08$

ans $\rightarrow$ 15.08 hrs

2 Logistic regression on real world data

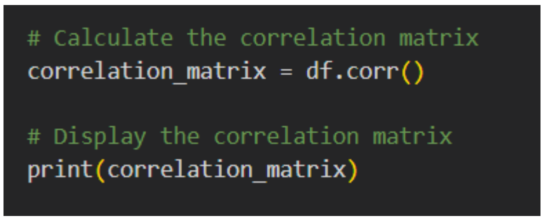
1.Choose a data set from UCI Machine Learning Repository that is appropriate for logistic regression.

Heart Disease dataset is chosen for logistic regression because it contains multiple features that can be used to predict the presence or absence of heart disease. The dataset is well-suited for binary classification, as the target variable (`target`) represents whether the patient has heart disease (value 1) or not (value 0).

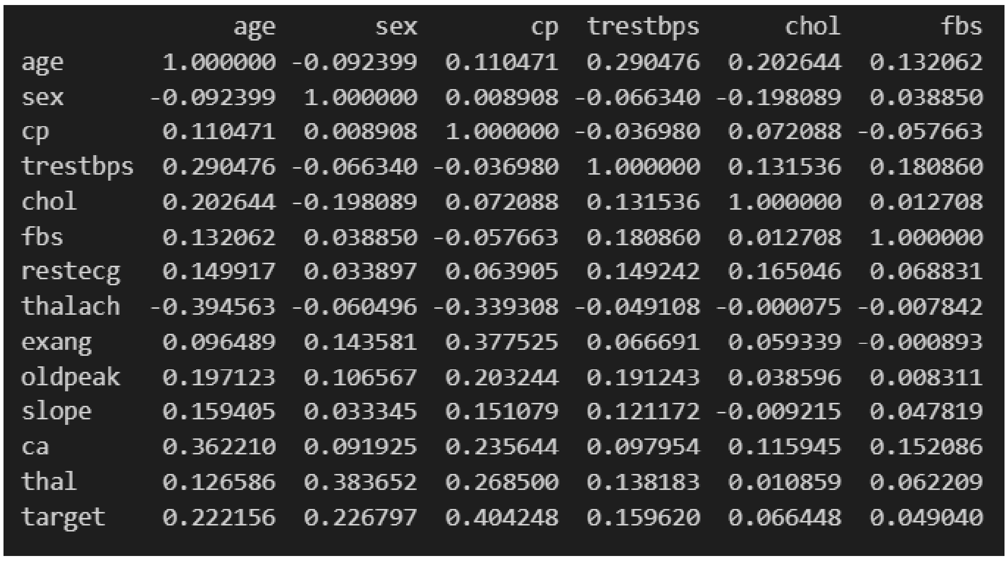


```python
import pandas as pd

# Load the Heart Disease dataset
url = "https://archive.ics.uci.edu/ml/machine-learning-databases/heart-disease/processed.cleveland.data"
columns = ["age", "sex", "cp", "trestbps", "chol", "fbs", "restecg",
           "thalach", "exang", "oldpeak", "slope", "ca", "thal", "target"]

df = pd.read_csv(url, header=None, names=columns)

# Replace missing values (represented by '?') with NaN and convert to numeric
df.replace('?', float('nan'), inplace=True)
df = df.apply(pd.to_numeric)

# Drop rows with missing values (you can also use imputation if needed)
df = df.dropna()

# Display dataset information
print(df.info())
```

2.Obtain the correlation matrix



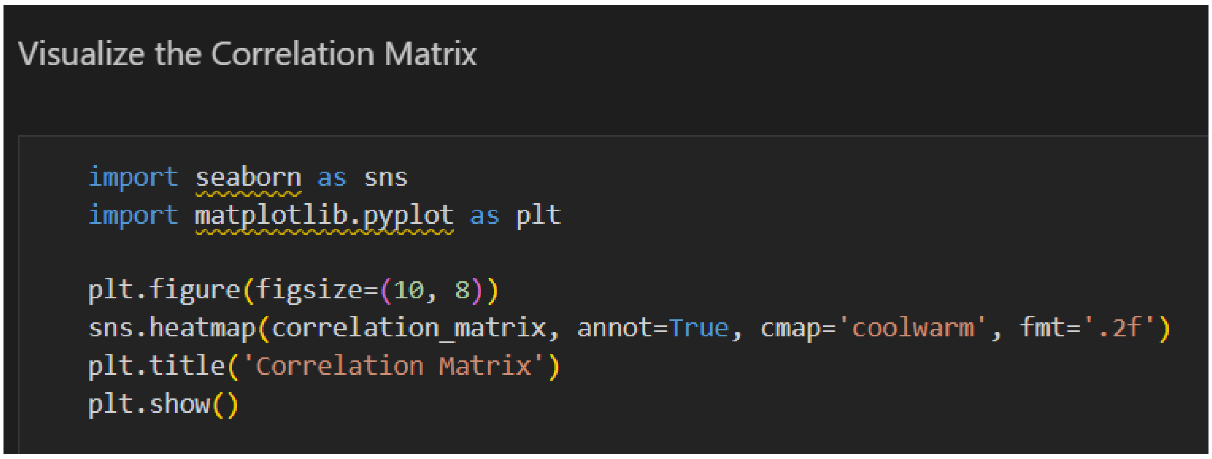
```python
# Calculate the correlation matrix
correlation_matrix = df.corr()

# Display the correlation matrix
print(correlation_matrix)
```



```
               age       sex        cp  trestbps      chol       fbs
age       1.000000 -0.092399  0.110471  0.290476  0.202644  0.132062
sex      -0.092399  1.000000  0.008908 -0.066340 -0.198089  0.038850
cp        0.110471  0.008908  1.000000 -0.036980  0.072088 -0.057663
trestbps  0.290476 -0.066340 -0.036980  1.000000  0.131536  0.180860
chol      0.202644 -0.198089  0.072088  0.131536  1.000000  0.012708
fbs       0.132062  0.038850 -0.057663  0.180860  0.012708  1.000000
restecg   0.149917  0.033897  0.063905  0.149242  0.165046  0.068831
thalach  -0.394563 -0.060496 -0.339308 -0.049108 -0.000075 -0.007842
exang     0.096489  0.143581  0.377525  0.066691  0.059339 -0.000893
oldpeak   0.197123  0.106567  0.203244  0.191243  0.038596  0.008311
slope     0.159405  0.033345  0.151079  0.121172 -0.009215  0.047819
ca        0.362210  0.091925  0.235644  0.097954  0.115945  0.152086
thal      0.126586  0.383652  0.268500  0.138183  0.010859  0.062209
target    0.222156  0.226797  0.404248  0.159620  0.066448  0.049040
```

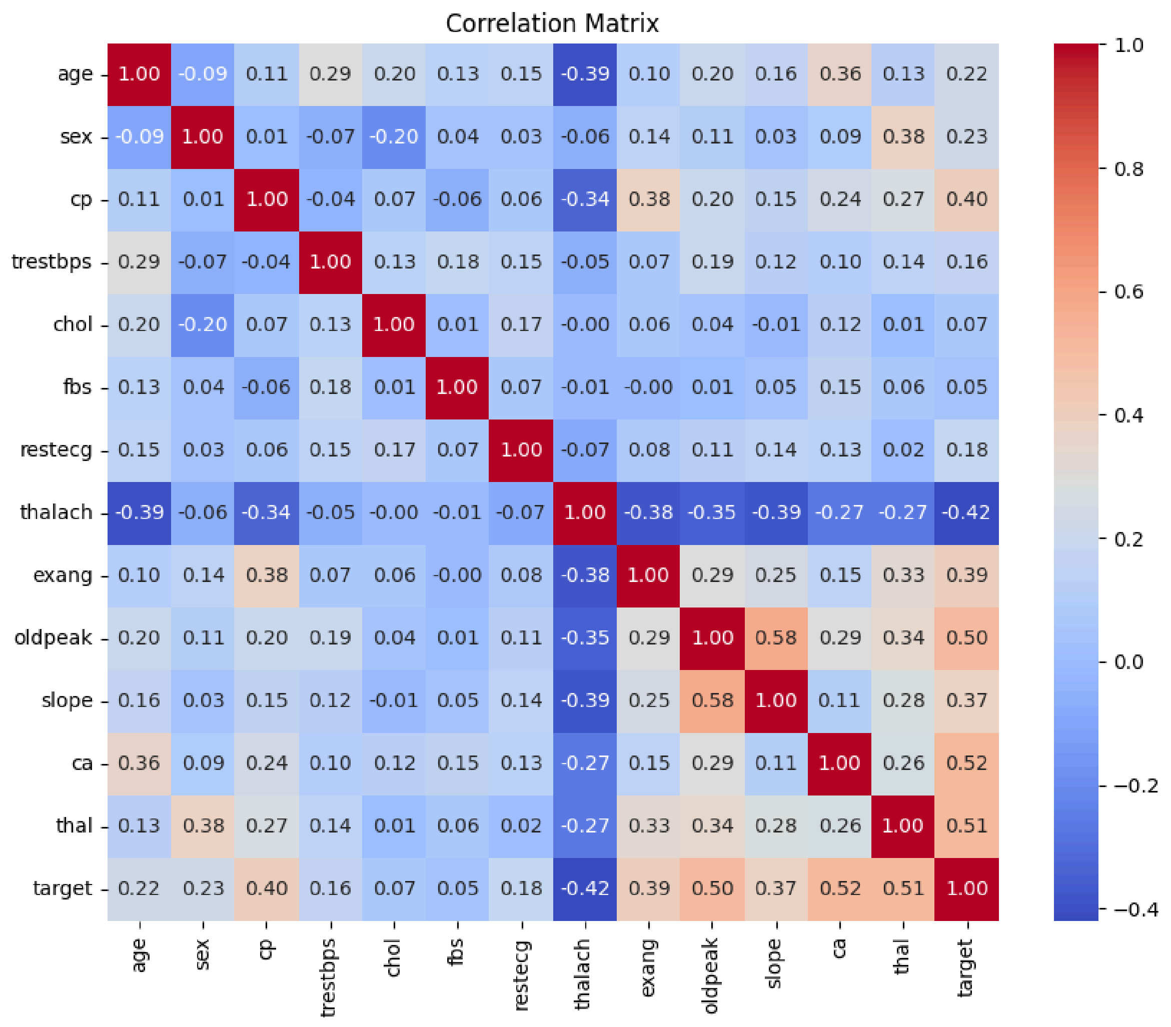
Correlation Matrix: The correlation matrix shows the strength and direction of relationships between variables (values range from -1 to 1)

- A positive correlation indicates that as one variable increases, the other also increases.
- A negative correlation indicates an inverse relationship.
- Values close to 0 indicate weak or no linear relationship.



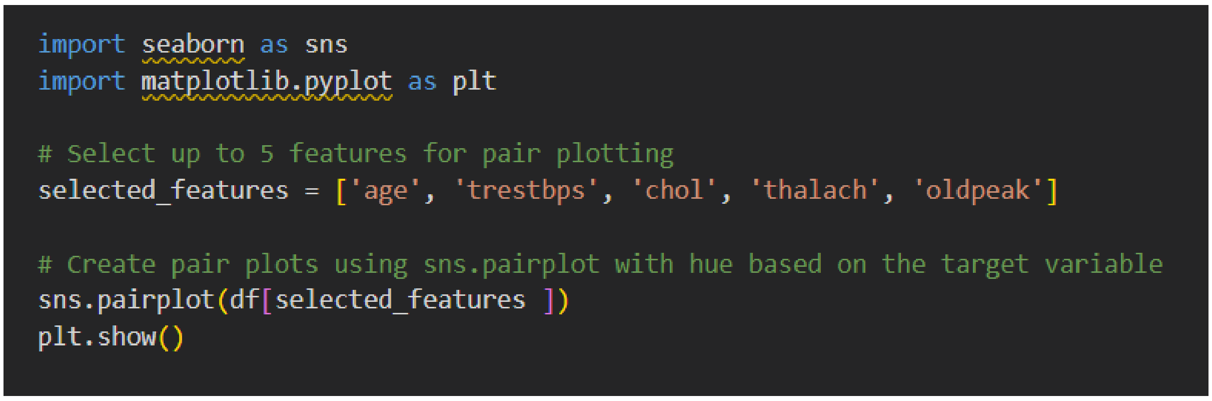
Visualize the Correlation Matrix

```python
import seaborn as sns
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt='.2f')
plt.title('Correlation Matrix')
plt.show()
```

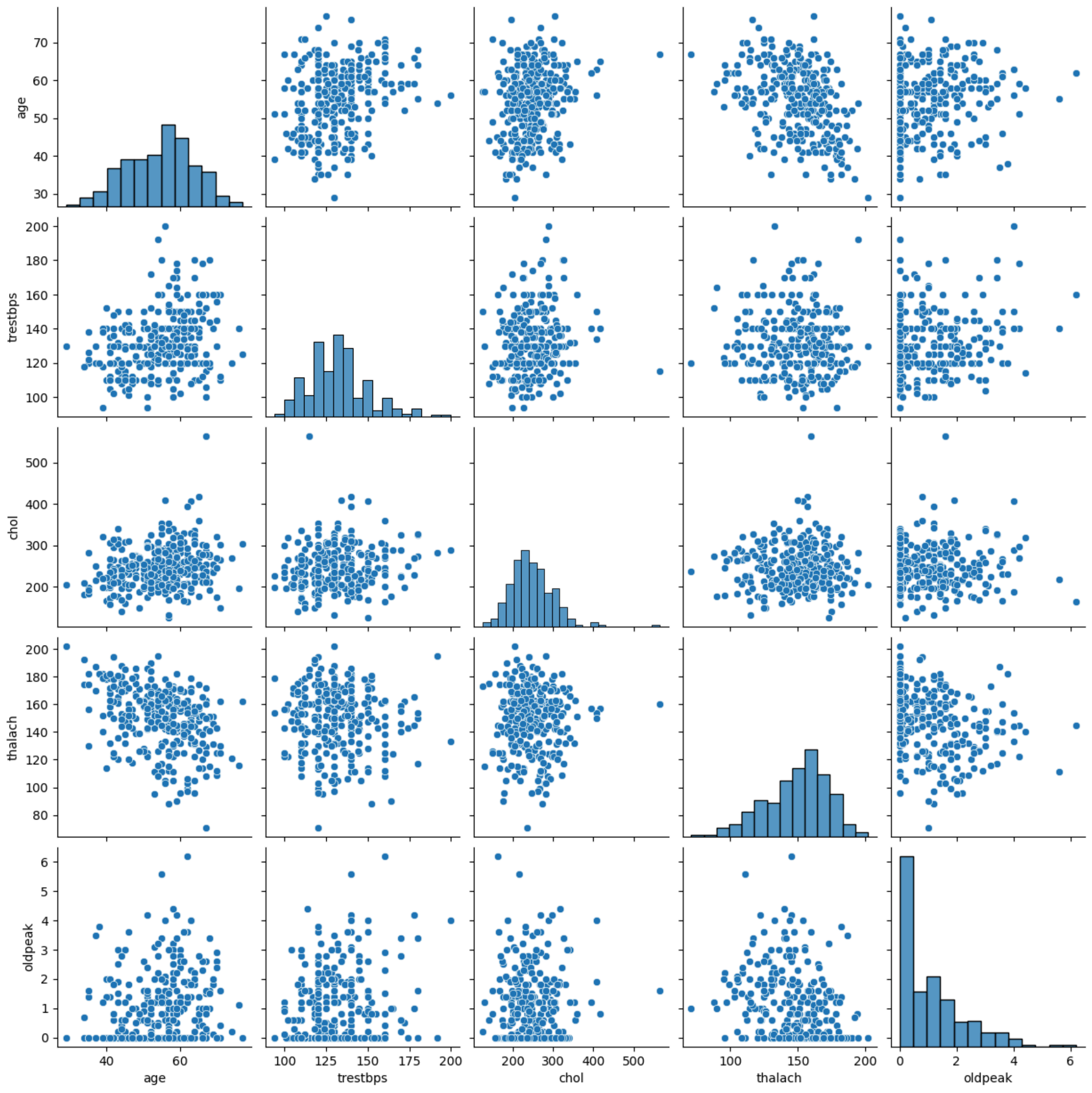


	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
age	1.00	-0.09	0.11	0.29	0.20	0.13	0.15	-0.39	0.10	0.20	0.16	0.36	0.13	0.22
sex	-0.09	1.00	0.01	-0.07	-0.20	0.04	0.03	-0.06	0.14	0.11	0.03	0.09	0.38	0.23
cp	0.11	0.01	1.00	-0.04	0.07	-0.06	0.06	-0.34	0.38	0.20	0.15	0.24	0.27	0.40
trestbps	0.29	-0.07	-0.04	1.00	0.13	0.18	0.15	-0.05	0.07	0.19	0.12	0.10	0.14	0.16
chol	0.20	-0.20	0.07	0.13	1.00	0.01	0.17	-0.00	0.06	0.04	-0.01	0.12	0.01	0.07
fbs	0.13	0.04	-0.06	0.18	0.01	1.00	0.07	-0.01	-0.00	0.01	0.05	0.15	0.06	0.05
restecg	0.15	0.03	0.06	0.15	0.17	0.07	1.00	-0.07	0.08	0.11	0.14	0.13	0.02	0.18
thalach	-0.39	-0.06	-0.34	-0.05	-0.00	-0.01	-0.07	1.00	-0.38	-0.35	-0.39	-0.27	-0.27	-0.42
exang	0.10	0.14	0.38	0.07	0.06	-0.00	0.08	-0.38	1.00	0.29	0.25	0.15	0.33	0.39
oldpeak	0.20	0.11	0.20	0.19	0.04	0.01	0.11	-0.35	0.29	1.00	0.58	0.29	0.34	0.50
slope	0.16	0.03	0.15	0.12	-0.01	0.05	0.14	-0.39	0.25	0.58	1.00	0.11	0.28	0.37
ca	0.36	0.09	0.24	0.10	0.12	0.15	0.13	-0.27	0.15	0.29	0.11	1.00	0.26	0.52
thal	0.13	0.38	0.27	0.14	0.01	0.06	0.02	-0.27	0.33	0.34	0.28	0.26	1.00	0.51
target	0.22	0.23	0.40	0.16	0.07	0.05	0.18	-0.42	0.39	0.50	0.37	0.52	0.51	1.00

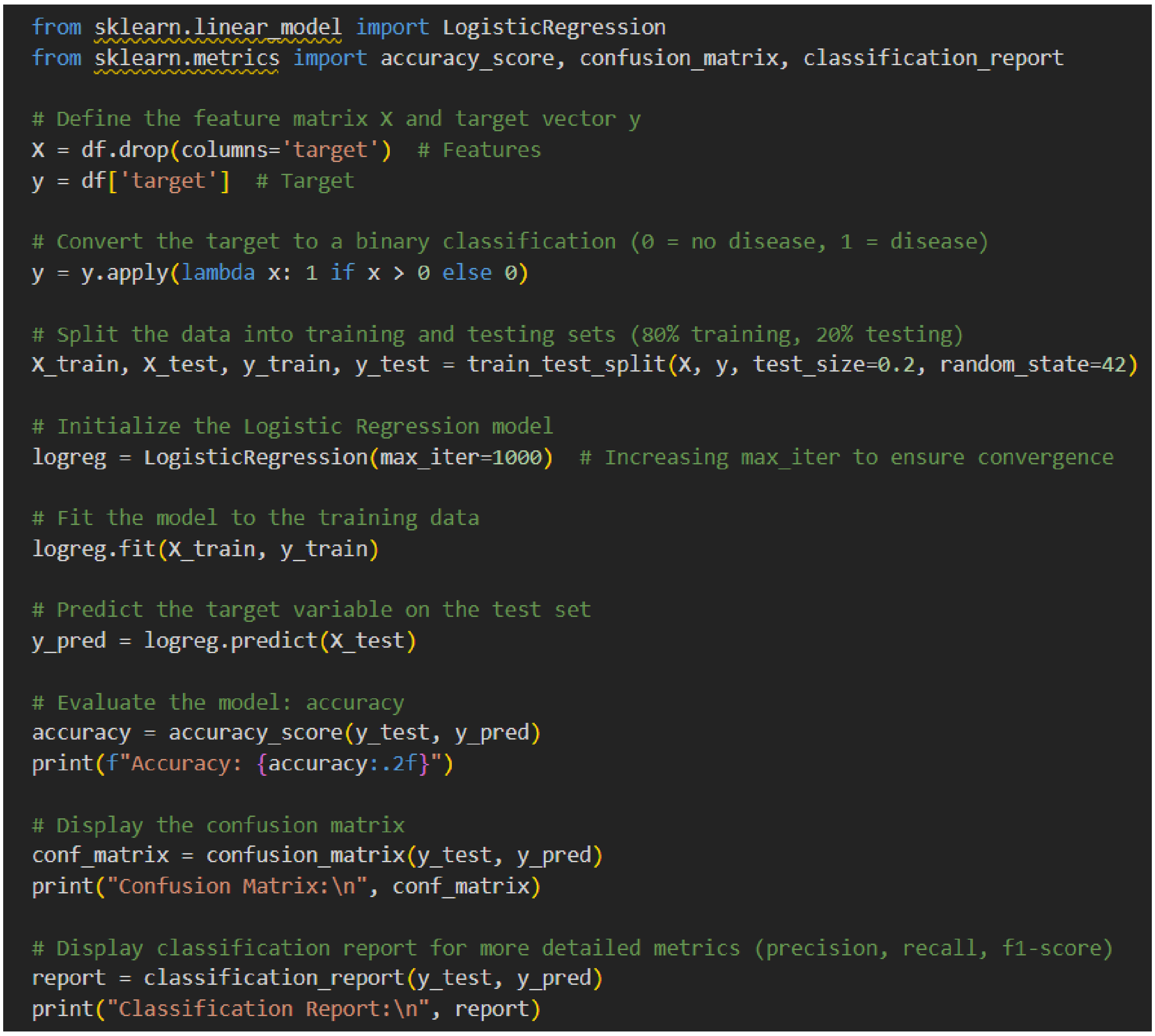
Obtain pair plots using sns.pairplot.



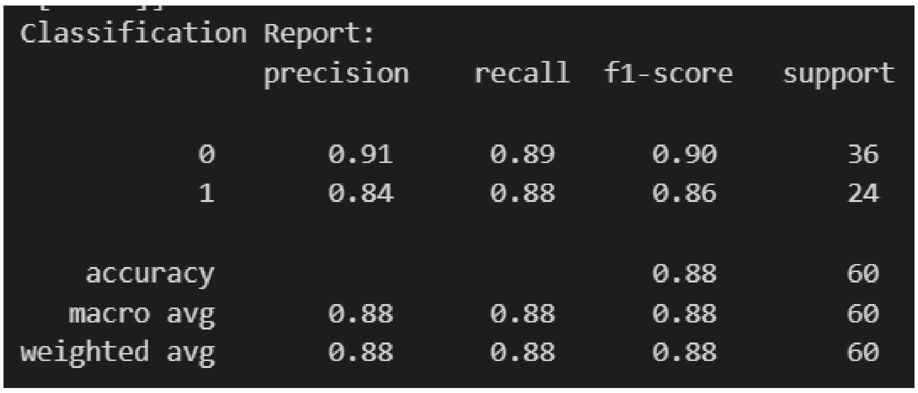
```
import seaborn as sns
import matplotlib.pyplot as plt

# Select up to 5 features for pair plotting
selected_features = ['age', 'trestbps', 'chol', 'thalach', 'oldpeak']

# Create pair plots using sns.pairplot with hue based on the target variable
sns.pairplot(df[selected_features ])
plt.show()
```



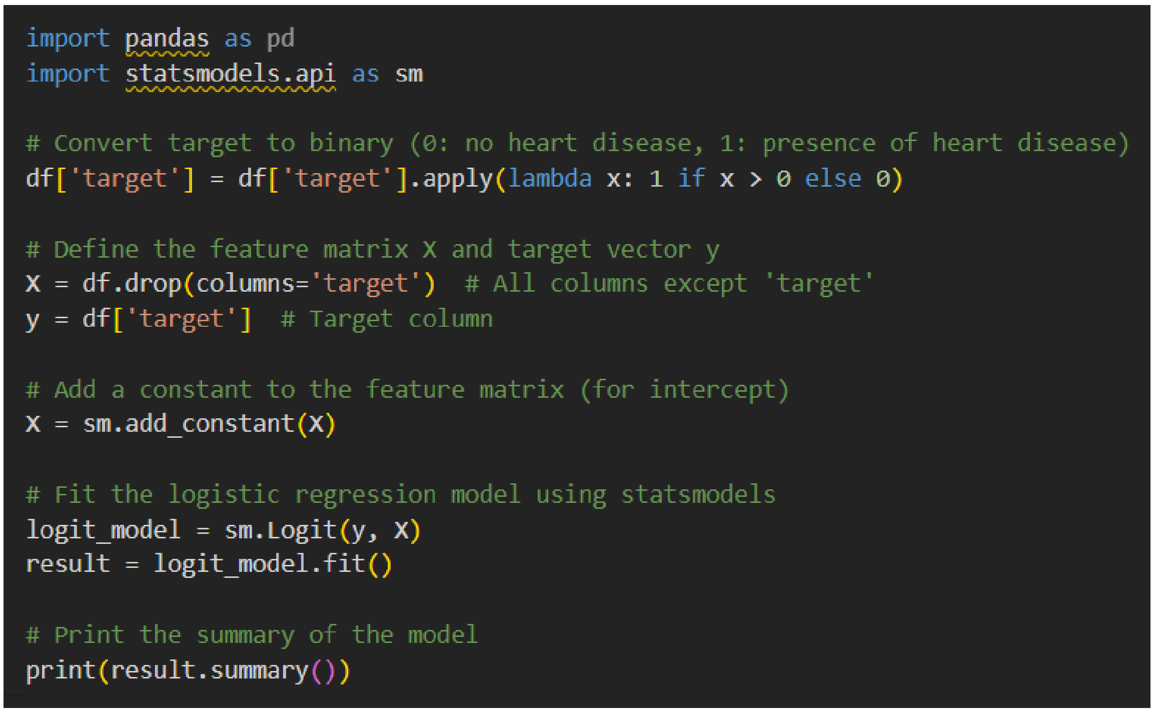
3.Fit a logistic regression model. Evaluate the model's performance.



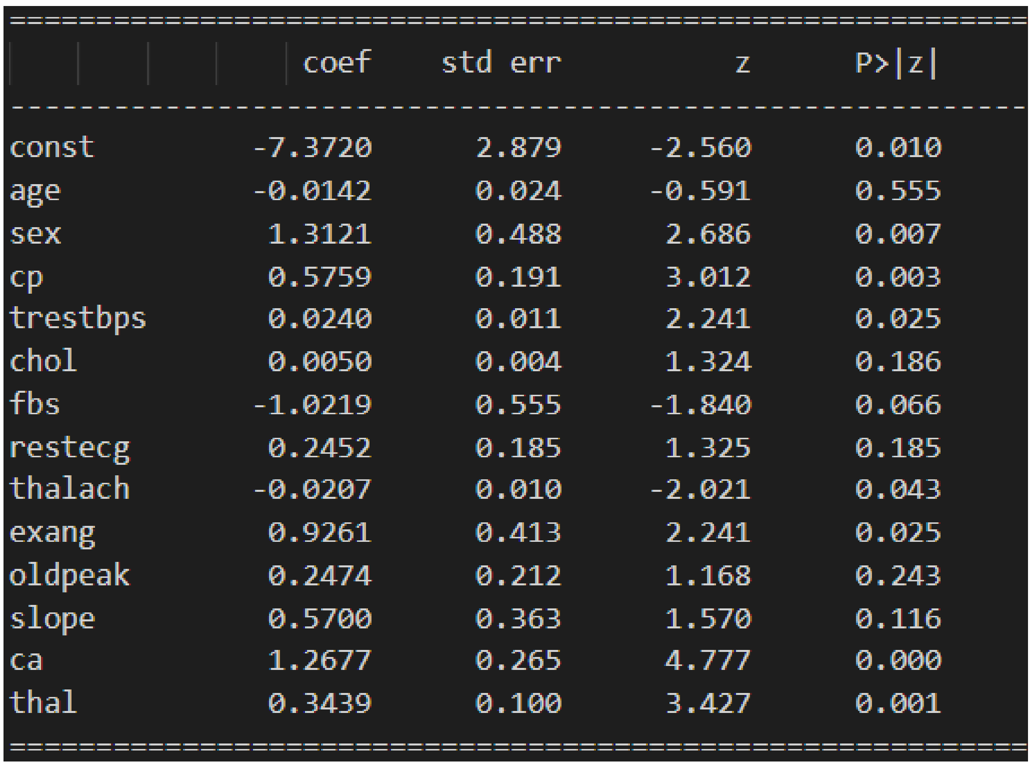
```python
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

# Define the feature matrix X and target vector y
X = df.drop(columns='target')  # Features
y = df['target']  # Target

# Convert the target to a binary classification (0 = no disease, 1 = disease)
y = y.apply(lambda x: 1 if x > 0 else 0)

# Split the data into training and testing sets (80% training, 20% testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Initialize the Logistic Regression model
logreg = LogisticRegression(max_iter=1000)  # Increasing max_iter to ensure convergence

# Fit the model to the training data
logreg.fit(X_train, y_train)

# Predict the target variable on the test set
y_pred = logreg.predict(X_test)

# Evaluate the model: accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")

# Display the confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:\n", conf_matrix)

# Display classification report for more detailed metrics (precision, recall, f1-score)
report = classification_report(y_test, y_pred)
print("Classification Report:\n", report)
```



```
Classification Report:
               precision    recall  f1-score   support

           0       0.91      0.89      0.90        36
           1       0.84      0.88      0.86        24

    accuracy                           0.88        60
   macro avg       0.88      0.88      0.88        60
weighted avg       0.88      0.88      0.88        60
```



```python
import pandas as pd
import statsmodels.api as sm

# Convert target to binary (0: no heart disease, 1: presence of heart disease)
df['target'] = df['target'].apply(lambda x: 1 if x > 0 else 0)

# Define the feature matrix X and target vector y
X = df.drop(columns='target')  # All columns except 'target'
y = df['target']  # Target column

# Add a constant to the feature matrix (for intercept)
X = sm.add_constant(X)

# Fit the logistic regression model using statsmodels
logit_model = sm.Logit(y, X)
result = logit_model.fit()

# Print the summary of the model
print(result.summary())
```



```
==============================================================
                coef    std err          z      P>|z|
--------------------------------------------------------------
const        -7.3720      2.879     -2.560      0.010
age          -0.0142      0.024     -0.591      0.555
sex           1.3121      0.488      2.686      0.007
cp            0.5759      0.191      3.012      0.003
trestbps      0.0240      0.011      2.241      0.025
chol          0.0050      0.004      1.324      0.186
fbs          -1.0219      0.555     -1.840      0.066
restecg       0.2452      0.185      1.325      0.185
thalach      -0.0207      0.010     -2.021      0.043
exang         0.9261      0.413      2.241      0.025
oldpeak       0.2474      0.212      1.168      0.243
slope         0.5700      0.363      1.570      0.116
ca            1.2677      0.265      4.777      0.000
thal          0.3439      0.100      3.427      0.001
==============================================================
```

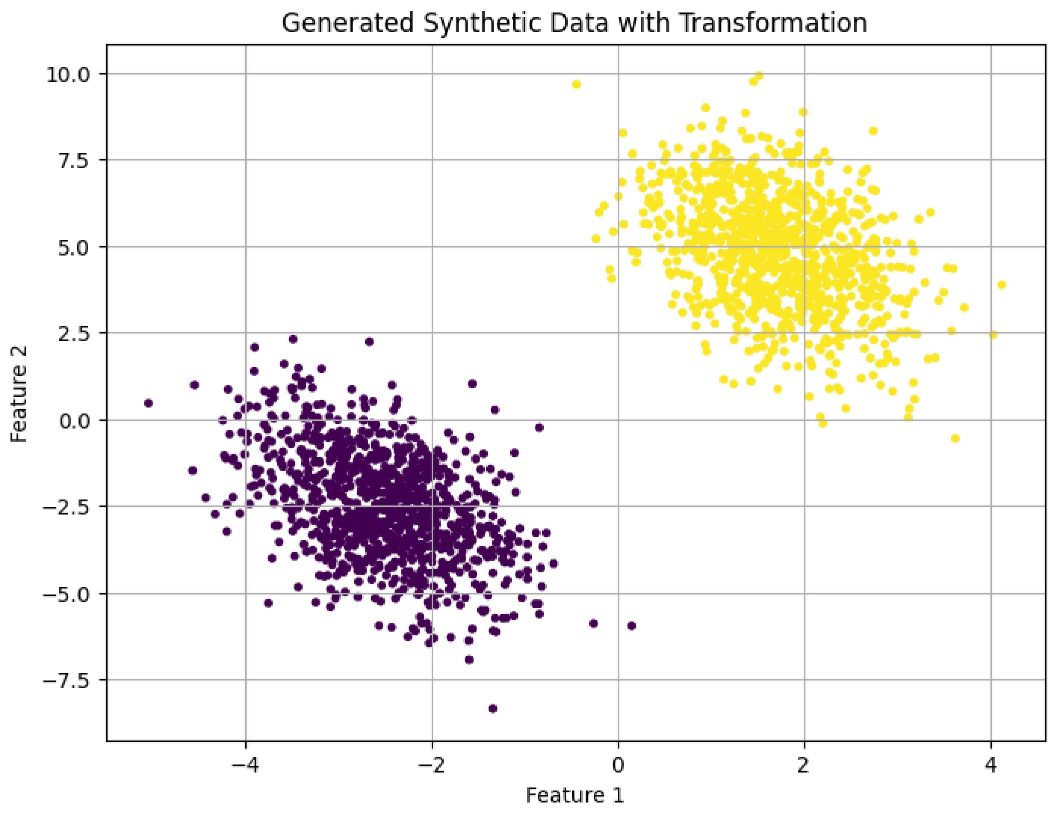
The p-values for each feature in the regression summary indicate the significance of that feature in predicting the target variable.

Features with p-values less than 0.05 are considered statistically significant. (const,sex,cp,trestbps,thalach,exang,ca,thal)

Features with p-values greater than 0.05 can often be discarded, as they do not significantly contribute to the model's predictive power.(age,chol,fbs,restecg,oldpeak,slope)

3 Logistic regression First/Second-Order Methods

1. Use the code given in listing 4 to generate data



2.State the method used to initialize the weights and reason for your selection.

Initializing the weights to zeros. Starting with zeros allows the model to treat all features equally at the beginning of training, avoiding biases towards any feature initially. This is particularly effective when the features are standardized.

3.Specify the loss function you have used and state reason for your selection.

Loss function- Binary Cross-Entropy Loss (Log Loss)

$$\text{Loss}(y, \hat{y}) = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)\right]$$

N is the number of samples.

yi is the true label for sample i (0 or 1).

y^i is the predicted probability of the positive class for sample i.

Reasons for Selection

Suitability for Binary Classification:

- The Binary Cross-Entropy Loss is designed specifically for binary classification tasks, making it an appropriate choice when the output is a probability that indicates whether a sample belongs to a particular class.

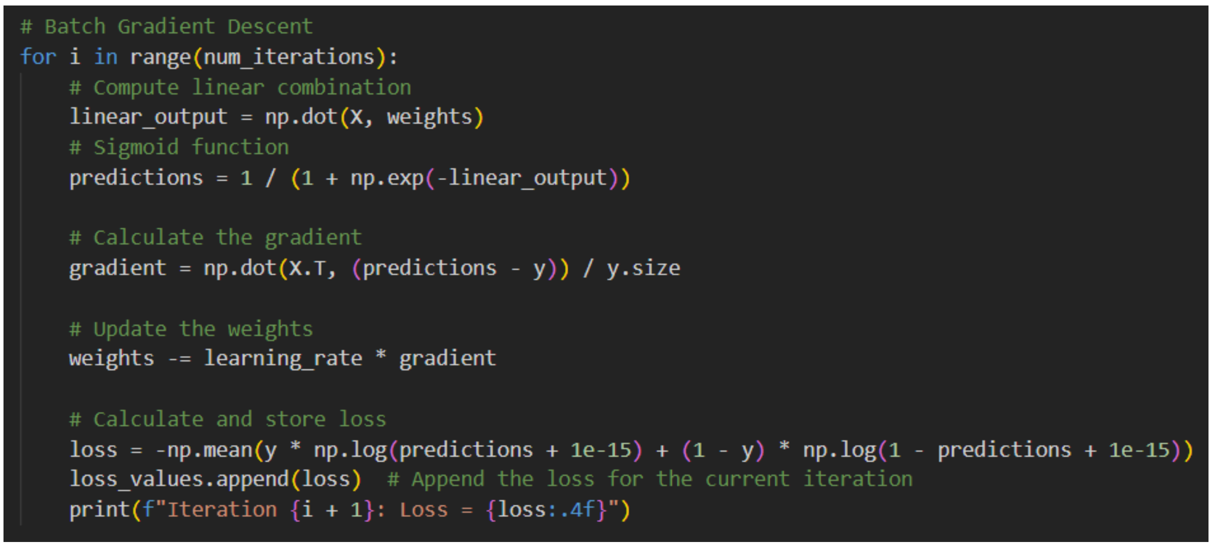
Probability Interpretation:

- Since logistic regression outputs probabilities via the sigmoid function, Binary Cross-Entropy directly measures how well the predicted probabilities match the actual class labels.

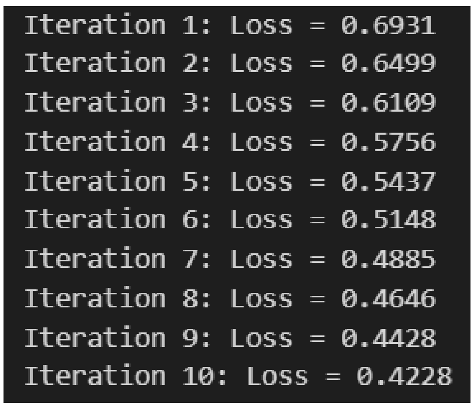
Minimization Objective:

- The goal of logistic regression is to minimize the loss function, and using Binary Cross-Entropy ensures that the optimization process is aligned with the probabilistic interpretation of the model outputs.

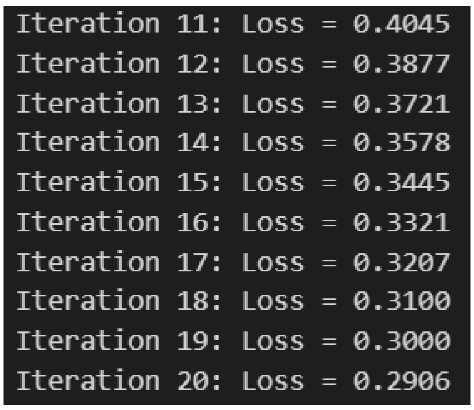
4. Plot the loss with respect to number of iterations



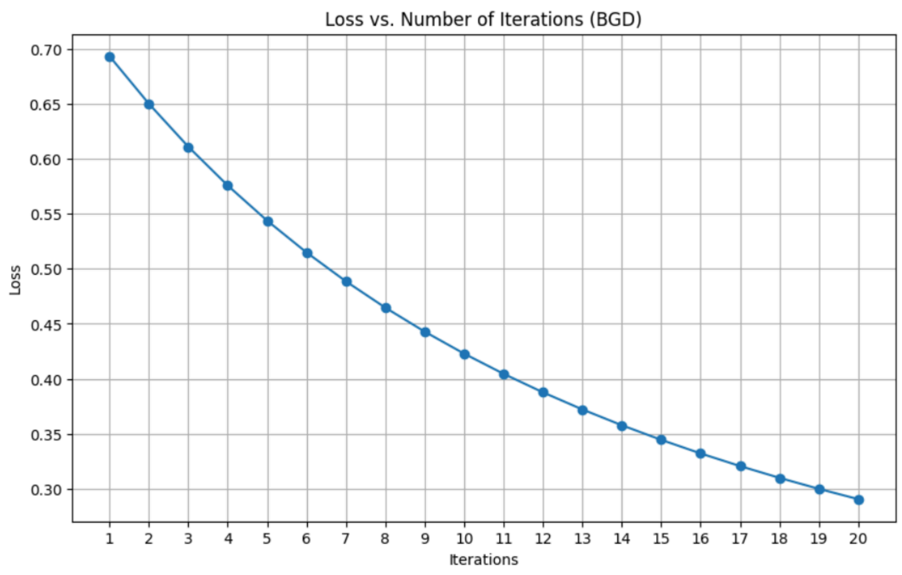
```python
# Batch Gradient Descent
for i in range(num_iterations):
    # Compute linear combination
    linear_output = np.dot(X, weights)
    # Sigmoid function
    predictions = 1 / (1 + np.exp(-linear_output))

    # Calculate the gradient
    gradient = np.dot(X.T, (predictions - y)) / y.size

    # Update the weights
    weights -= learning_rate * gradient

    # Calculate and store loss
    loss = -np.mean(y * np.log(predictions + 1e-15) + (1 - y) * np.log(1 - predictions + 1e-15))
    loss_values.append(loss)  # Append the loss for the current iteration
    print(f"Iteration {i + 1}: Loss = {loss:.4f}")
```



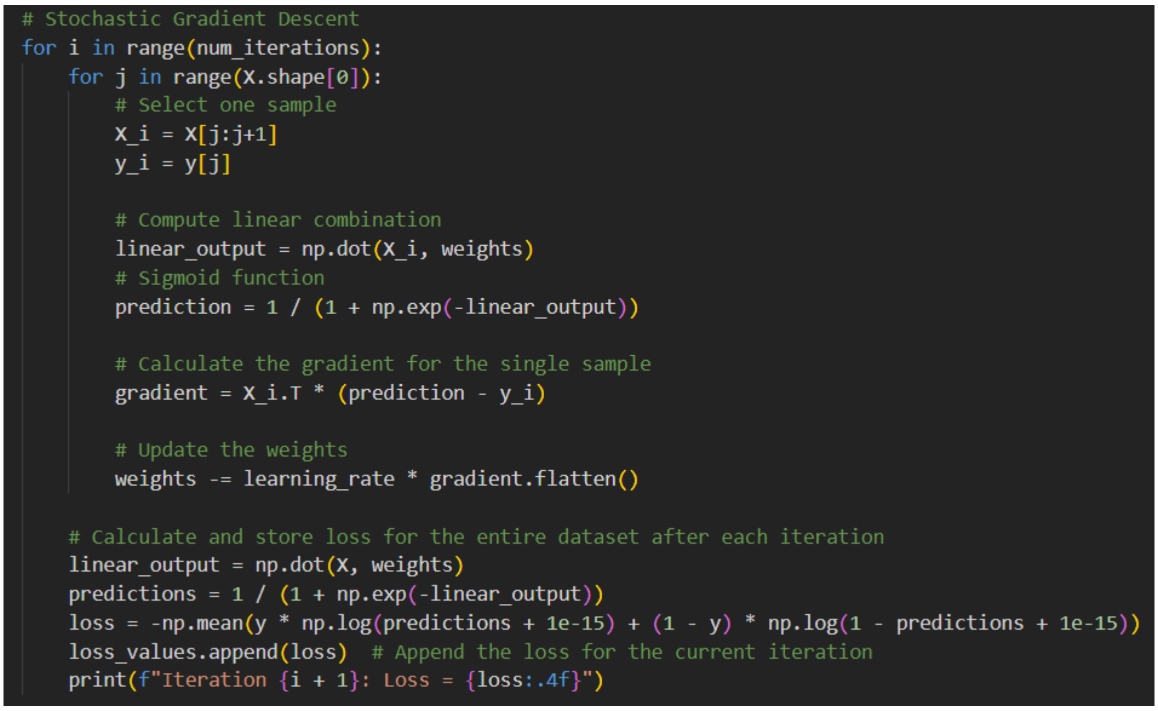
```
Iteration 1: Loss = 0.6931
Iteration 2: Loss = 0.6499
Iteration 3: Loss = 0.6109
Iteration 4: Loss = 0.5756
Iteration 5: Loss = 0.5437
Iteration 6: Loss = 0.5148
Iteration 7: Loss = 0.4885
Iteration 8: Loss = 0.4646
Iteration 9: Loss = 0.4428
Iteration 10: Loss = 0.4228
```



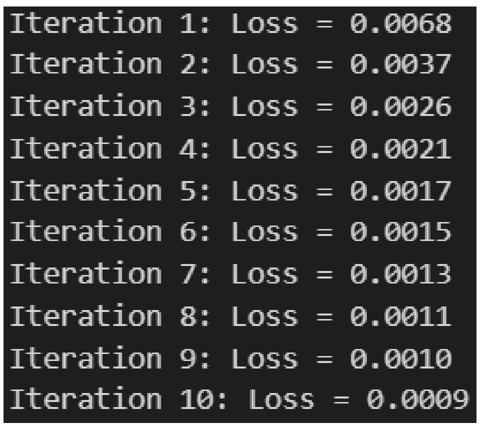
```
Iteration 11: Loss = 0.4045
Iteration 12: Loss = 0.3877
Iteration 13: Loss = 0.3721
Iteration 14: Loss = 0.3578
Iteration 15: Loss = 0.3445
Iteration 16: Loss = 0.3321
Iteration 17: Loss = 0.3207
Iteration 18: Loss = 0.3100
Iteration 19: Loss = 0.3000
Iteration 20: Loss = 0.2906
```



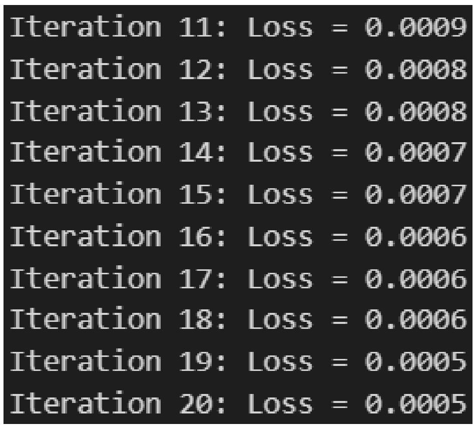
5. Repeat step 2 for stochastic Gradient descent.



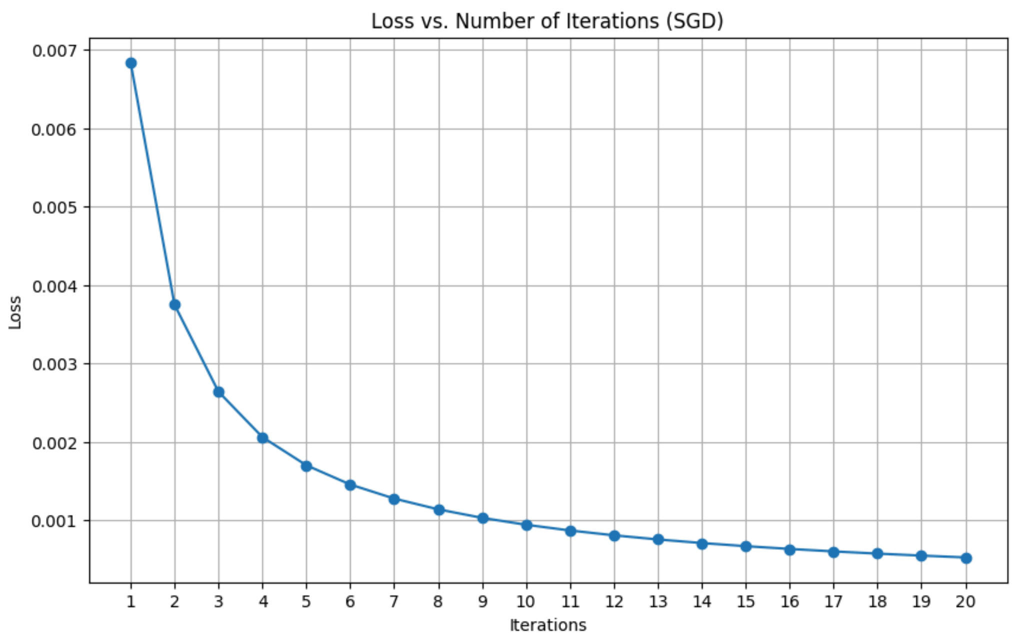
```
# Stochastic Gradient Descent
for i in range(num_iterations):
    for j in range(X.shape[0]):
        # Select one sample
        X_i = X[j:j+1]
        y_i = y[j]

        # Compute linear combination
        linear_output = np.dot(X_i, weights)
        # Sigmoid function
        prediction = 1 / (1 + np.exp(-linear_output))

        # Calculate the gradient for the single sample
        gradient = X_i.T * (prediction - y_i)

        # Update the weights
        weights -= learning_rate * gradient.flatten()

    # Calculate and store loss for the entire dataset after each iteration
    linear_output = np.dot(X, weights)
    predictions = 1 / (1 + np.exp(-linear_output))
    loss = -np.mean(y * np.log(predictions + 1e-15) + (1 - y) * np.log(1 - predictions + 1e-15))
    loss_values.append(loss)  # Append the loss for the current iteration
    print(f"Iteration {i + 1}: Loss = {loss:.4f}")
```



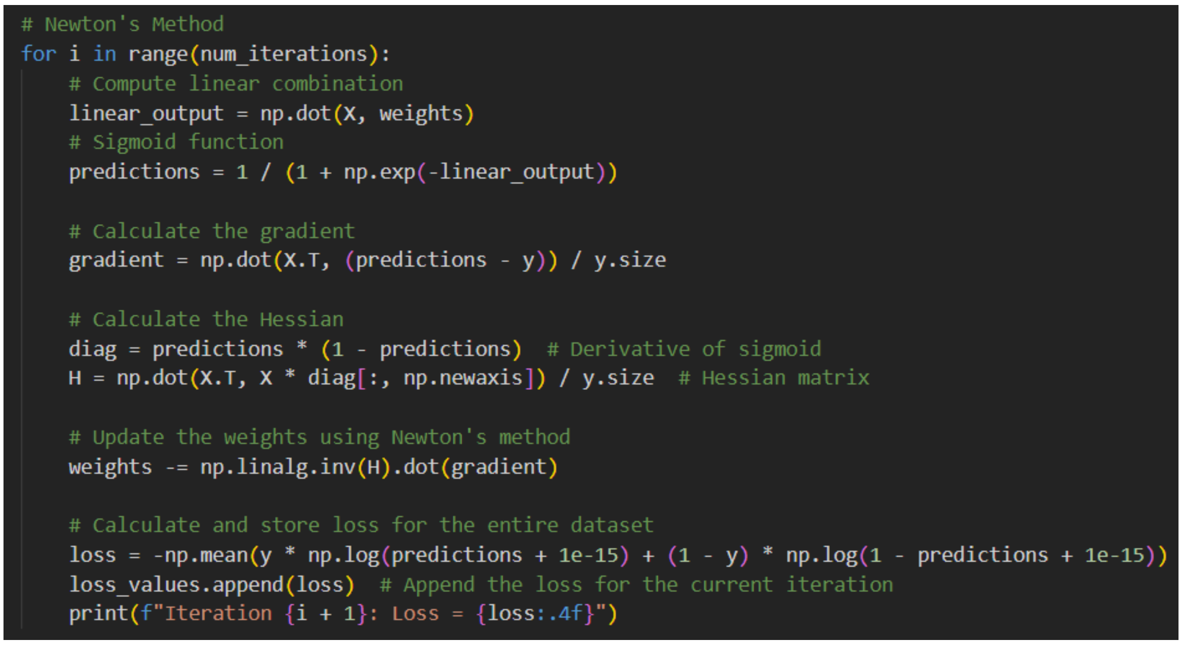
```
Iteration 1: Loss = 0.0068
Iteration 2: Loss = 0.0037
Iteration 3: Loss = 0.0026
Iteration 4: Loss = 0.0021
Iteration 5: Loss = 0.0017
Iteration 6: Loss = 0.0015
Iteration 7: Loss = 0.0013
Iteration 8: Loss = 0.0011
Iteration 9: Loss = 0.0010
Iteration 10: Loss = 0.0009
```



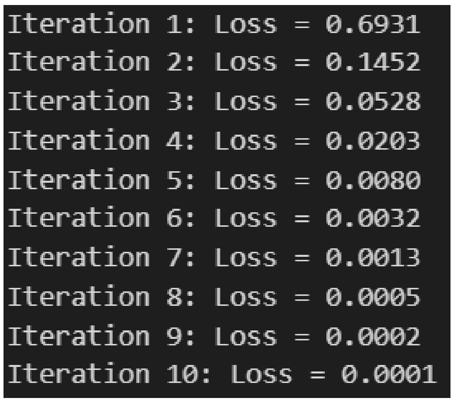
```
Iteration 11: Loss = 0.0009
Iteration 12: Loss = 0.0008
Iteration 13: Loss = 0.0008
Iteration 14: Loss = 0.0007
Iteration 15: Loss = 0.0007
Iteration 16: Loss = 0.0006
Iteration 17: Loss = 0.0006
Iteration 18: Loss = 0.0006
Iteration 19: Loss = 0.0005
Iteration 20: Loss = 0.0005
```



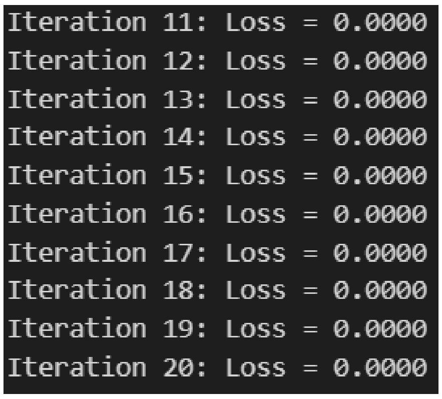
6. Implement Newton's method to update the weights for the given dataset over 20 iterations.



```
# Newton's Method
for i in range(num_iterations):
    # Compute linear combination
    linear_output = np.dot(X, weights)
    # Sigmoid function
    predictions = 1 / (1 + np.exp(-linear_output))

    # Calculate the gradient
    gradient = np.dot(X.T, (predictions - y)) / y.size

    # Calculate the Hessian
    diag = predictions * (1 - predictions)  # Derivative of sigmoid
    H = np.dot(X.T, X * diag[:, np.newaxis]) / y.size  # Hessian matrix

    # Update the weights using Newton's method
    weights -= np.linalg.inv(H).dot(gradient)

    # Calculate and store loss for the entire dataset
    loss = -np.mean(y * np.log(predictions + 1e-15) + (1 - y) * np.log(1 - predictions + 1e-15))
    loss_values.append(loss)  # Append the loss for the current iteration
    print(f"Iteration {i + 1}: Loss = {loss:.4f}")
```

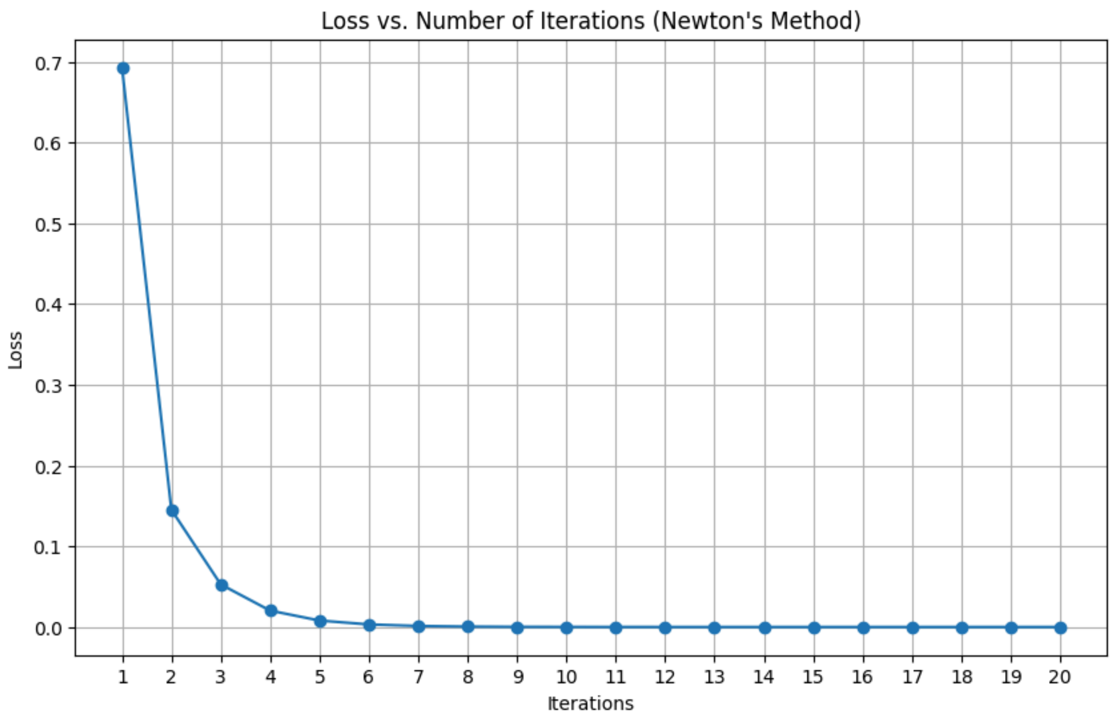


```
Iteration 1: Loss = 0.6931
Iteration 2: Loss = 0.1452
Iteration 3: Loss = 0.0528
Iteration 4: Loss = 0.0203
Iteration 5: Loss = 0.0080
Iteration 6: Loss = 0.0032
Iteration 7: Loss = 0.0013
Iteration 8: Loss = 0.0005
Iteration 9: Loss = 0.0002
Iteration 10: Loss = 0.0001
```

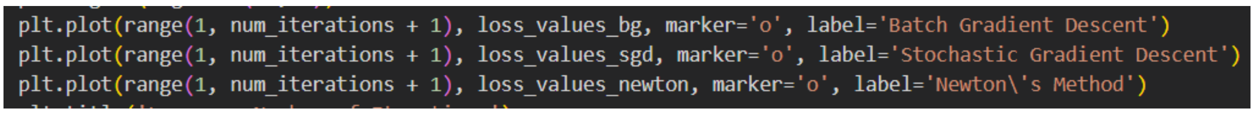


```
Iteration 11: Loss = 0.0000
Iteration 12: Loss = 0.0000
Iteration 13: Loss = 0.0000
Iteration 14: Loss = 0.0000
Iteration 15: Loss = 0.0000
Iteration 16: Loss = 0.0000
Iteration 17: Loss = 0.0000
Iteration 18: Loss = 0.0000
Iteration 19: Loss = 0.0000
Iteration 20: Loss = 0.0000
```

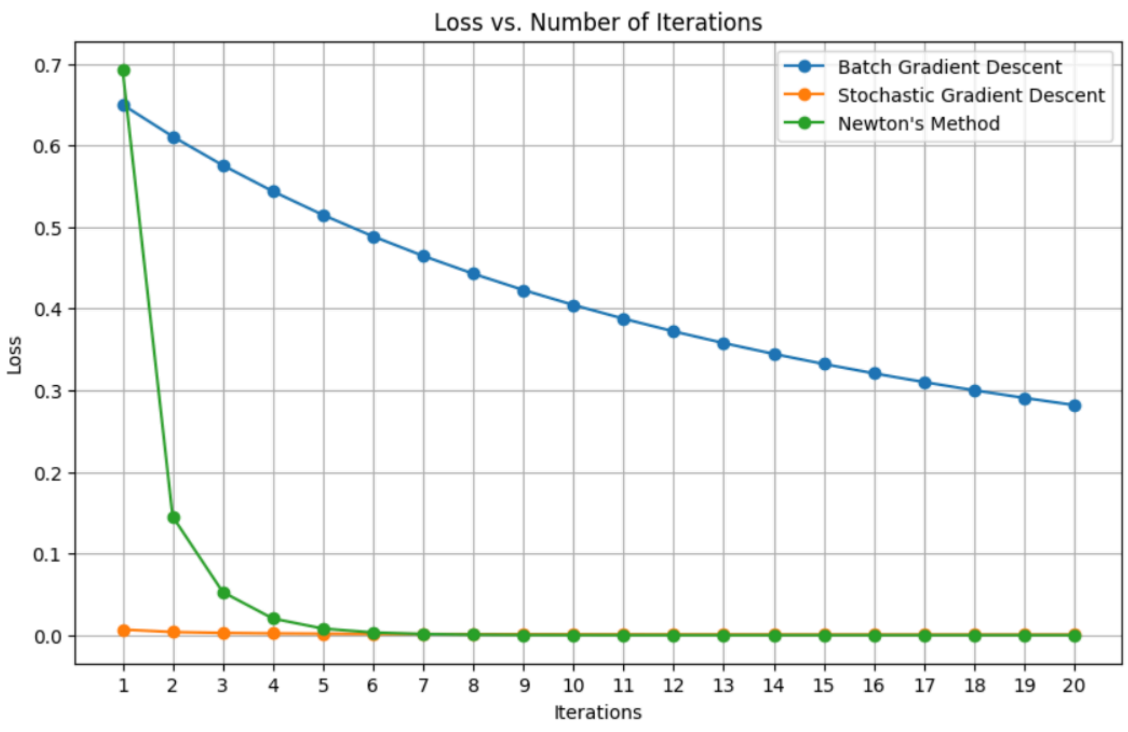
7. Plot the loss with respect to number of iterations.



8.Plot the loss with respect to number of iterations for Gradient descent, stochastic Gradient descent and Newton method's in a single plot. Comment on your results.



```
plt.plot(range(1, num_iterations + 1), loss_values_bg, marker='o', label='Batch Gradient Descent')
plt.plot(range(1, num_iterations + 1), loss_values_sgd, marker='o', label='Stochastic Gradient Descent')
plt.plot(range(1, num_iterations + 1), loss_values_newton, marker='o', label='Newton\'s Method')
```



Batch Gradient Descent (BGD):

- Loss decreases steadily but at a slower rate compared to the other two methods.
- It starts with a high loss and gradually decreases over the iterations, showing a typical smooth curve for BGD. This method computes the gradient over the entire dataset, which explains the steady convergence but with slower initial improvement.

Stochastic Gradient Descent (SGD):

- SGD converges much faster, reaching a very low loss by the 4th iteration.
- The loss rapidly drops within the first few iterations, indicating its faster initial convergence due to the use of single data points but it is noisier and fluctuates slightly at the beginning.

Newton's Method:

- Newton's Method achieves the lowest loss the fastest, with almost instantaneous convergence by the 3rd iteration.
- Its convergence speed is much faster than the gradient-based methods, owing to the second-order information (Hessian) it uses, which allows for quicker jumps towards the minimum.

9. Propose two approaches to decide number of iterations for Gradient descent and Newton's method.

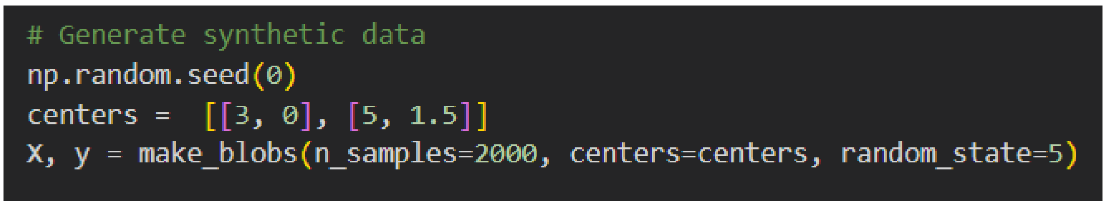
Convergence Threshold Approach

- Instead of pre-defining a fixed number of iterations, monitor the rate of decrease in loss (or cost function) and stop the optimization when the reduction in loss between consecutive iterations becomes very small.

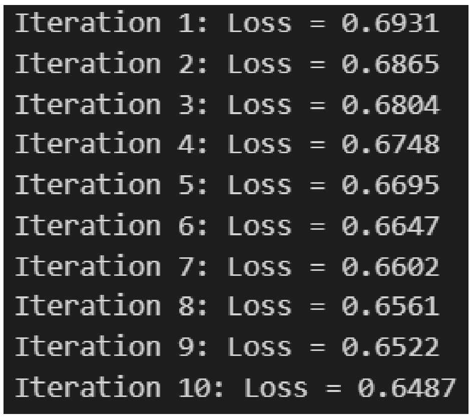
Cross-Validation Approach

- Use a validation set (different from the training set) to evaluate the performance of the model at various points during the training. The idea is to stop the algorithm when the validation loss starts increasing indicating no further meaningful improvement.

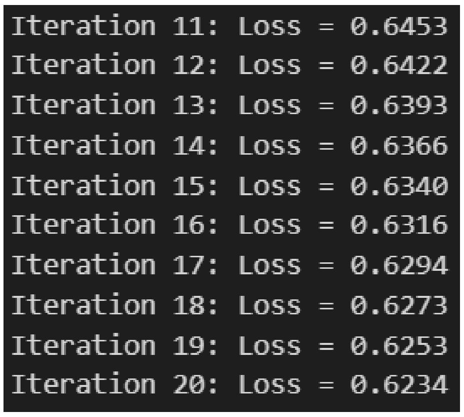
10. Suppose the centers in in listing 4 are changed to centers = [[3, 0], [5, 1.5]]. Use batch gradient descent to update the weights for this new configuration.Analyze the convergence behavior of the algorithm with this updated data, and provide an explanation for convergence behavior.



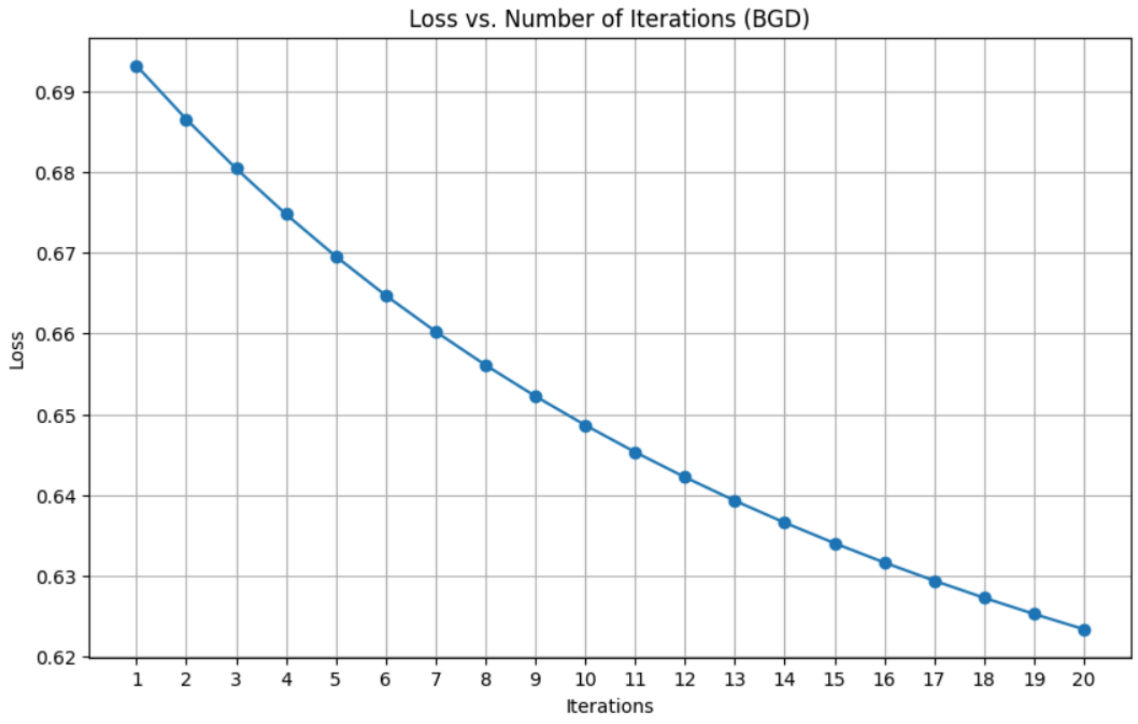
```
# Generate synthetic data
np.random.seed(0)
centers =  [[3, 0], [5, 1.5]]
X, y = make_blobs(n_samples=2000, centers=centers, random_state=5)
```



```
Iteration 1: Loss = 0.6931
Iteration 2: Loss = 0.6865
Iteration 3: Loss = 0.6804
Iteration 4: Loss = 0.6748
Iteration 5: Loss = 0.6695
Iteration 6: Loss = 0.6647
Iteration 7: Loss = 0.6602
Iteration 8: Loss = 0.6561
Iteration 9: Loss = 0.6522
Iteration 10: Loss = 0.6487
```



```
Iteration 11: Loss = 0.6453
Iteration 12: Loss = 0.6422
Iteration 13: Loss = 0.6393
Iteration 14: Loss = 0.6366
Iteration 15: Loss = 0.6340
Iteration 16: Loss = 0.6316
Iteration 17: Loss = 0.6294
Iteration 18: Loss = 0.6273
Iteration 19: Loss = 0.6253
Iteration 20: Loss = 0.6234
```



Centers [[3, 0], [5, 1.5]]

The reduction in loss is slower and more gradual, with small improvements from one iteration to the next.

The centers [3, 0] and [5, 1.5] are closer together, which results in less separability between the two classes. The decision boundary is less clear, which makes it more challenging for gradient descent to converge quickly

Centers [[−5, 0], [5, 1.5]]

The loss drops significantly faster.This shows a much more rapid decline in loss.

The centers [-5, 0] and [5, 1.5] are far apart, meaning the two classes are well-separated in the feature space.The decision boundary is more easily identified, allowing gradient descent to converge more quickly.