EN3150 -Pattern Recognition

Assignment 02

Learning from data and related challenges and classification

W.C.A Wickramaarachchi
210699K
01/10/2024

1) Logistic Regression

1) Load the data

2) The line in the code both fits the Label Encoder and transforms the “species” column into numerical labels. This process assigns unique integers to each species name 'Adelie' and 'Chinstrap'. This process of converting String label to numeric label is essential for making logistic regression models. Other thing is because this “species” feature is the target variable of our model we must have numerical inputs.

3) This given line is the one create the feature matrix X by excluding the columns “species, island and sex” from the data frame df. Because the species is the target variable for modeling, we must exclude it from X matrix. “Island” and “sex“ may not be necessary for predictions. “axis=1” in the code specifies the things should be dropped are columns, not rows.

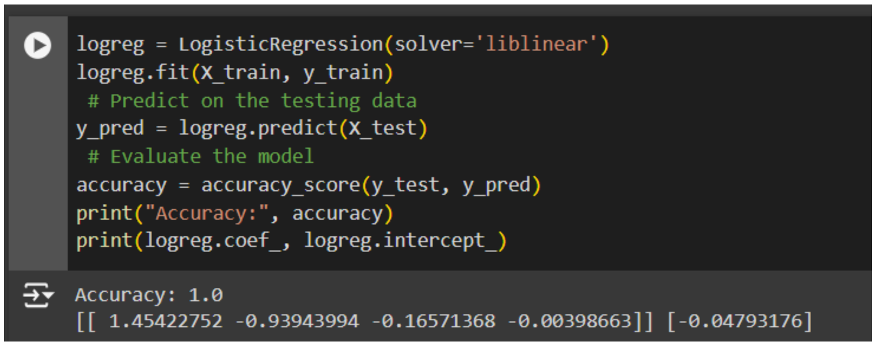
4) First thing is “island and “sex” features are categorical variables. Logistic regression requires numerical inputs, and before feeding categorical variables to the model wehave to encode them. Additionally these features might be considered less directly relevant because this model is train by morphological measurements. Removing less important features can help in simplifying the model, reducing overfitting effects.

5) training logistic regression model.

6) The random state parameter in the train_test_split function ensures the consistent data splits, Reproducibility and No Data leakages happen. While maintaining the integrity of testing data by ensuring it is randomly selected but consistent across different runs, it prevent any accidental data leakage that might be happen during training phases.

7) Logistic regression model “saga” is sensitive to feature scaling. If the numerical features have very different ranges, the model will not perform well. It is better to use “standardScaller” or similar method before training. The other thing which affect to the “saga” regression model is, it is designed for large datasets and handles L1 or elastic-net regulations. If the regularization is not needed or the data set is small, it might not perform well. Theres a convergence warning saying that the solver struggle with the current feature scales, coefficients didn’t converge.

8)



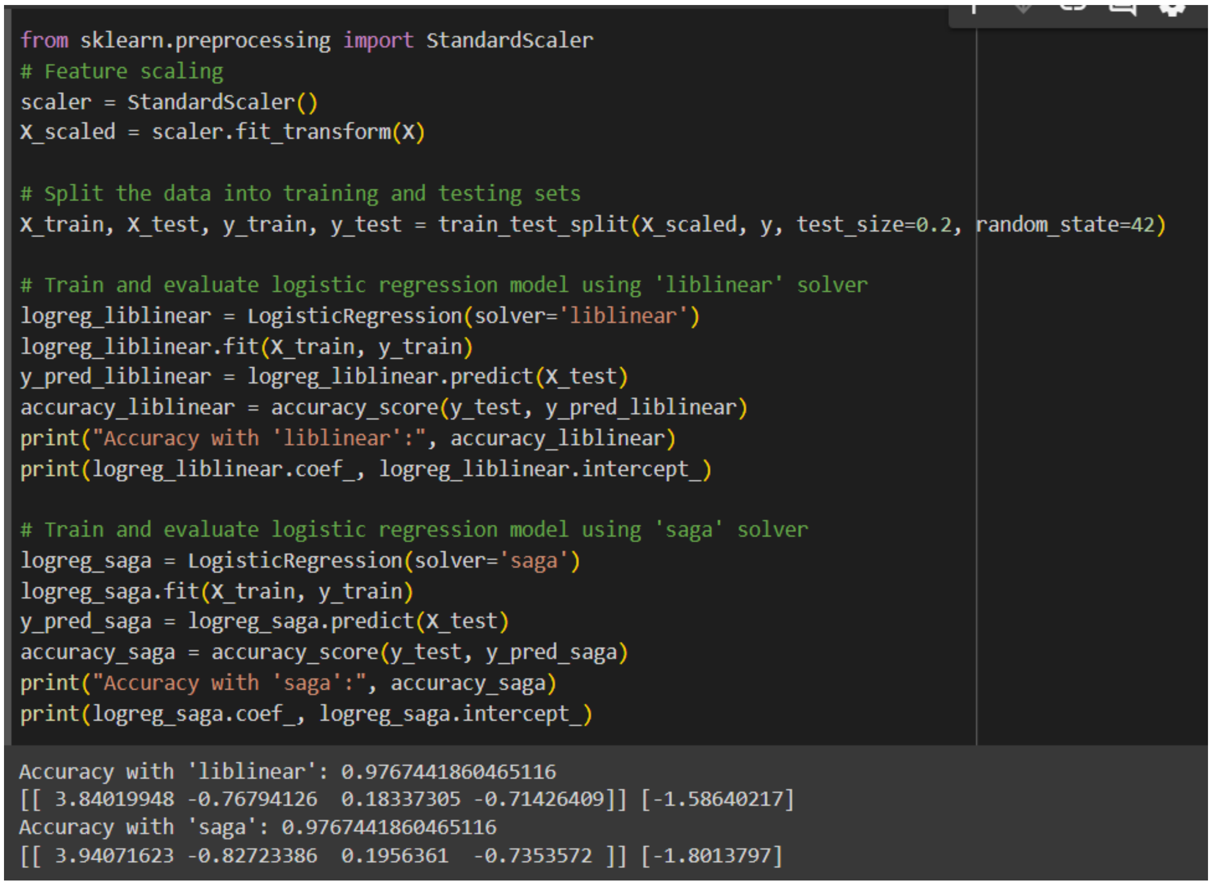
```
logreg = LogisticRegression(solver='liblinear')
logreg.fit(X_train, y_train)
 # Predict on the testing data
y_pred = logreg.predict(X_test)
 # Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
print(logreg.coef_, logreg.intercept_)
```

```
Accuracy: 1.0
[[ 1.45422752 -0.93943994 -0.16571368 -0.00398663]] [-0.04793176]
```

Accuracy:- 1.0

9) The “liblinear” solver is better for small data sets because it is optimized for binary classification . Saga is not good in smaller or less complex datasets. Other thing is liblinear solver don’t need feature scaling methods . But because “saga” solver use stochastic gradient descent it need feature scaling. Other thing is the way ‘liblinear” and “saga:” handle regularization can affect their performances. The default regularization in logistic regression is L2 and “liblinea” might converge to a more suitable solution with L2 regularization.

10)



```
from sklearn.preprocessing import StandardScaler
# Feature scaling
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

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

# Train and evaluate logistic regression model using 'liblinear' solver
logreg_liblinear = LogisticRegression(solver='liblinear')
logreg_liblinear.fit(X_train, y_train)
y_pred_liblinear = logreg_liblinear.predict(X_test)
accuracy_liblinear = accuracy_score(y_test, y_pred_liblinear)
print("Accuracy with 'liblinear':", accuracy_liblinear)
print(logreg_liblinear.coef_, logreg_liblinear.intercept_)

# Train and evaluate logistic regression model using 'saga' solver
logreg_saga = LogisticRegression(solver='saga')
logreg_saga.fit(X_train, y_train)
y_pred_saga = logreg_saga.predict(X_test)
accuracy_saga = accuracy_score(y_test, y_pred_saga)
print("Accuracy with 'saga':", accuracy_saga)
print(logreg_saga.coef_, logreg_saga.intercept_)
```

```
Accuracy with 'liblinear': 0.9767441860465116
[[ 3.84019948 -0.76794126  0.18337305 -0.71426409]] [-1.58640217]
Accuracy with 'saga': 0.9767441860465116
[[ 3.94071623 -0.82723386  0.1956361  -0.7353572 ]] [-1.8013797]
```

Solver	Unscaled Accuracy	Scaled Accuracy
Saga	0.5813953488372093	0.9767441860465116
Liblinear	1.0	0.9767441860465116

It is clear that because of standard scaling the accuracy of the model increase to 97% using saga solver. But the liblinear solver model not benefits due to scaling and the accuracy get reduced to 97%. Anyway after feature scaling both solvers recorded a similar accuracy. This highlights , if we use gradient descent-based optimization method for modelling, feature scaling is very much important for the accuracy of the model.

11) In the code they again separate the Y value (target variable) and X values (X matrix) from the data set and in the code they didn’t remove the “island” and “sex” features which are categorical variables which are not transform to numeric values. Because they are not in

numeric format we can’t applied them directly with logistic regression. If we have to use those features we have to encode them using a method like one-hot encoding. Otherwise we should drop the “island and “sex” columns from X matrix.

12) This approach is not correct.

Label Encoding convert categorical features in to arbitrary numerical value. As an example, “red”, “blue”, “green” might to encoded as 0,1,2 respectively. This encoding implies an ordinal relationship that may not exist. (as ‘green’>’blue’>’red’). When we scale these numbers, it essentially normalizing these arbitrary distinctions, not actual meaningful distances or relationships. This can mislead the machine learning model. Instead of using label encoding we can use one hot encoding to handling categorical variables. This technique transforms each category into a separate binary column, ensuring the model treats each category as an independent feature without implying any ordinality. By doing so, it preserves the unique characteristics of categorical data and prevents the model from mistakenly assigning an unintended order to the categories. This is essential for accurate model training. One-hot encoding also avoids distorting relationships between categories that might arise from scaling, ultimately improving model performance by capturing the distinct influence of each category on the target variable.

Question 2

1) In logistic regression, the probability p of receiving an A+ is given by the following formula.

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

$$w_0 = -5.9 \quad w_1 = 0.06 \quad w_2 = 1.5 \quad x_1 = 50 \quad x_2 = 3.6$$

$$P = \frac{1}{1 + e^{-(-5.9 + 0.06 * 50 + 1.5 * 3.6)}}$$

$$P = 0.9241$$

The estimated probability that the student will receive an A+ is 92.4%.

2)

$$0.6 = \frac{1}{1 + e^{-(w_0 + w_1 x_1 + w_2 x_2)}}$$

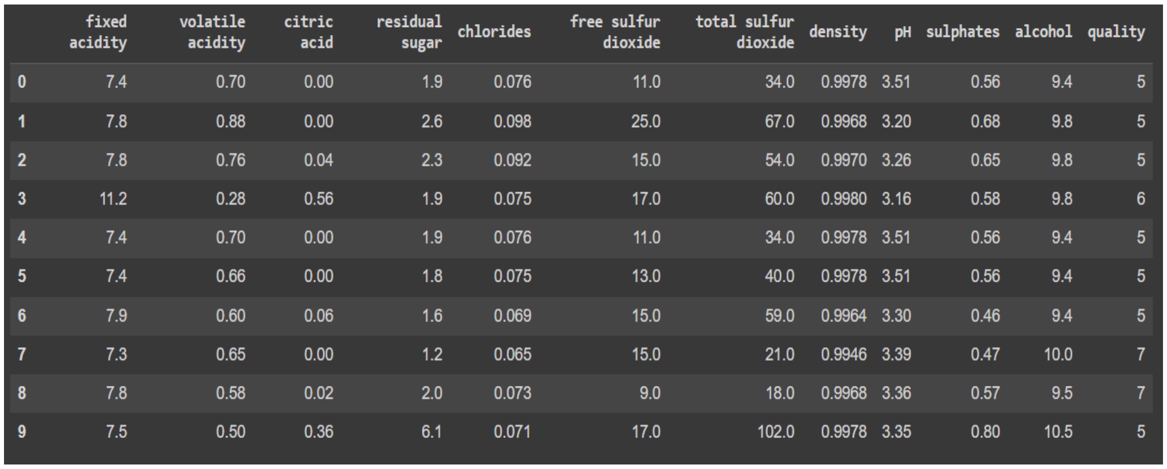
$$0.6 = \frac{1}{1 + e^{-(-5.9 + 0.06 x_1 + 1.5 * 3.6)}}$$

$$x_1 = 15.091$$

Student should study 15.091 hours to achieve a 60% chance of receiving an A+

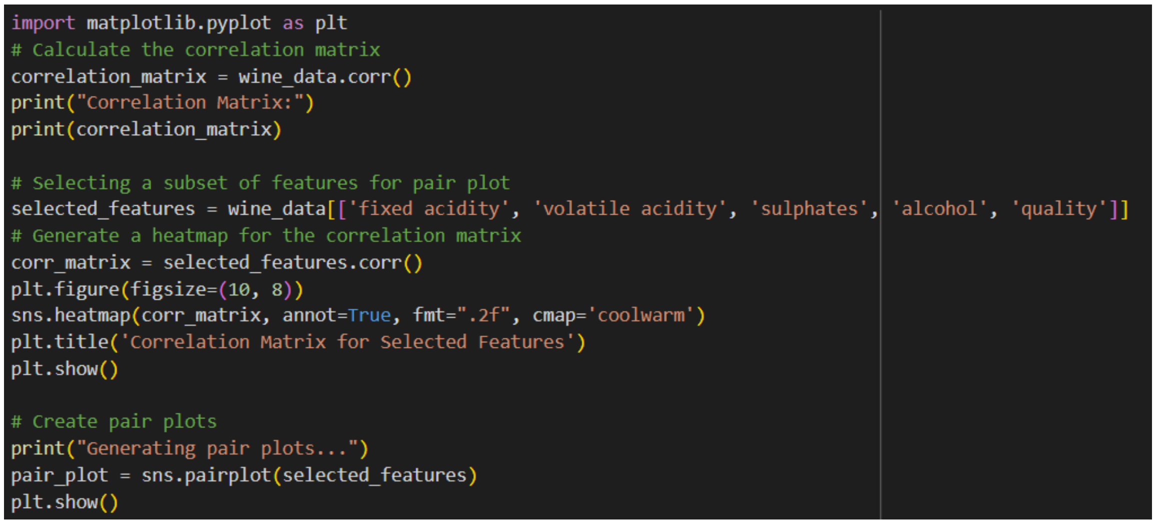
2) Logistic regression on real world data

1) I chose Wine Quality data set



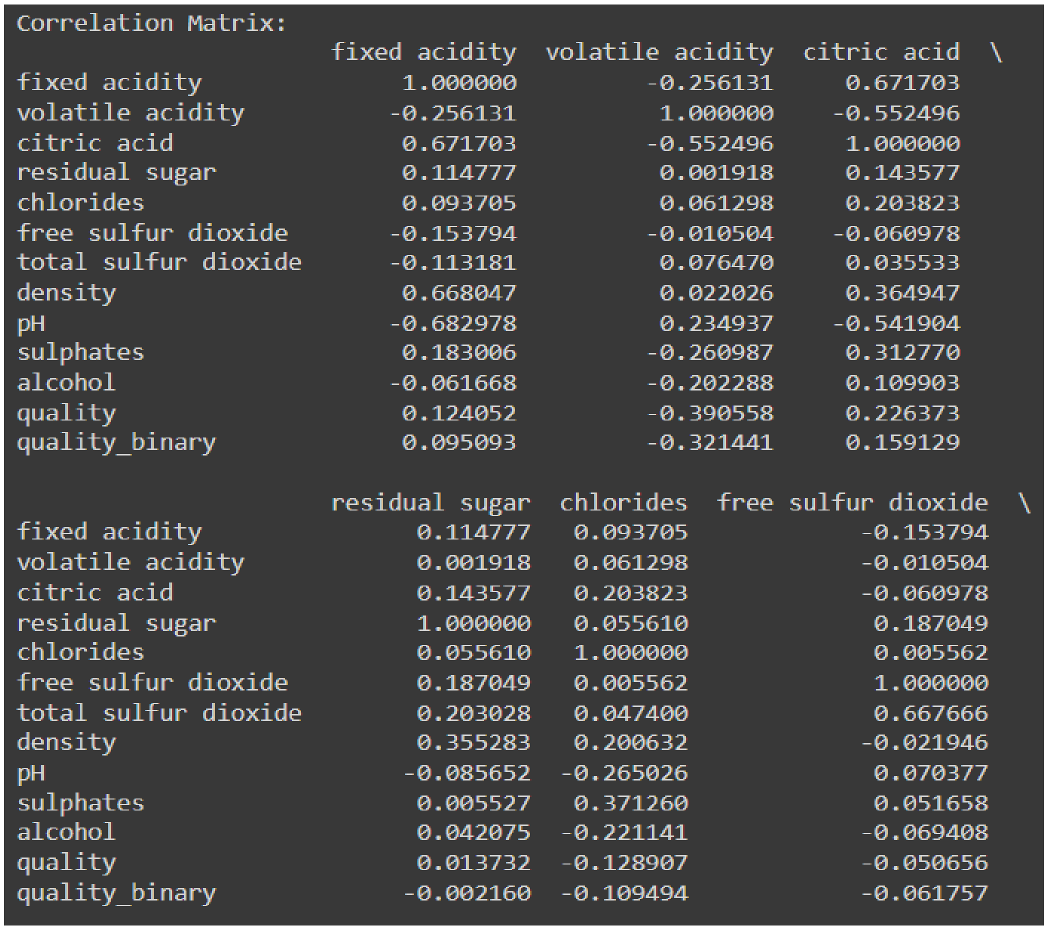
	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	5
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	0.68	9.8	5
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	0.65	9.8	5
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	0.58	9.8	6
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	5
5	7.4	0.66	0.00	1.8	0.075	13.0	40.0	0.9978	3.51	0.56	9.4	5
6	7.9	0.60	0.06	1.6	0.069	15.0	59.0	0.9964	3.30	0.46	9.4	5
7	7.3	0.65	0.00	1.2	0.065	15.0	21.0	0.9946	3.39	0.47	10.0	7
8	7.8	0.58	0.02	2.0	0.073	9.0	18.0	0.9968	3.36	0.57	9.5	7
9	7.5	0.50	0.36	6.1	0.071	17.0	102.0	0.9978	3.35	0.80	10.5	5

2)

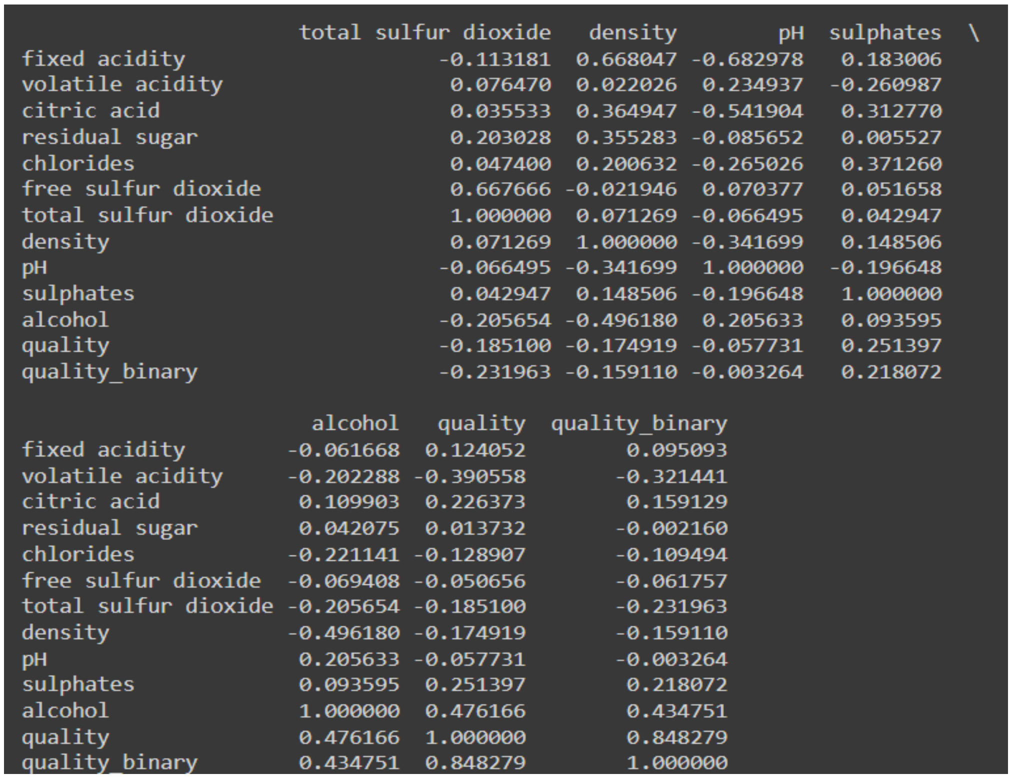


```python
import matplotlib.pyplot as plt
# Calculate the correlation matrix
correlation_matrix = wine_data.corr()
print("Correlation Matrix:")
print(correlation_matrix)

# Selecting a subset of features for pair plot
selected_features = wine_data[['fixed acidity', 'volatile acidity', 'sulphates', 'alcohol', 'quality']]
# Generate a heatmap for the correlation matrix
corr_matrix = selected_features.corr()
plt.figure(figsize=(10, 8))
sns.heatmap(corr_matrix, annot=True, fmt=".2f", cmap='coolwarm')
plt.title('Correlation Matrix for Selected Features')
plt.show()

# Create pair plots
print("Generating pair plots...")
pair_plot = sns.pairplot(selected_features)
plt.show()
```

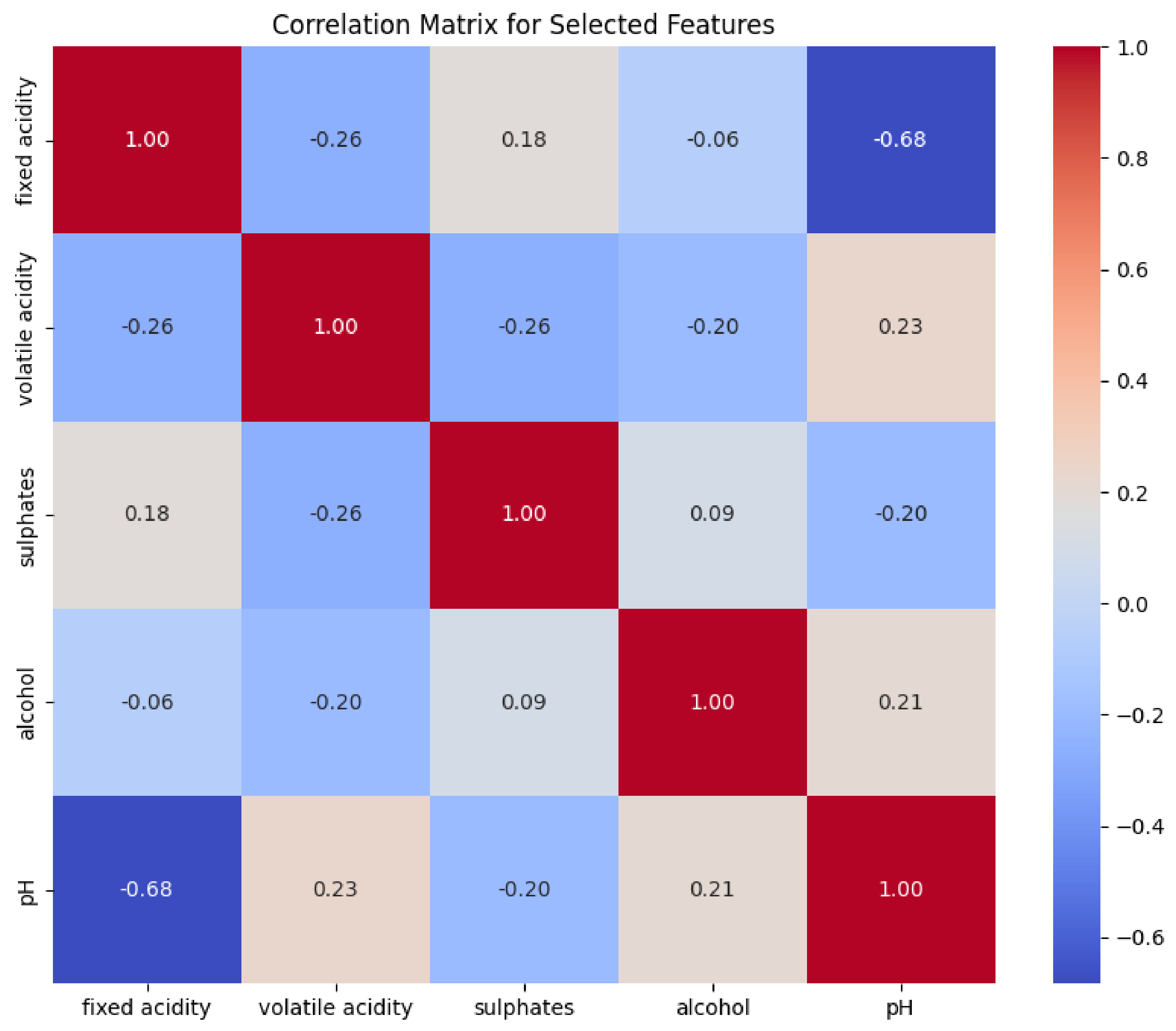
Correlation Matrix :-



```
Correlation Matrix:
                      fixed acidity  volatile acidity  citric acid  \
fixed acidity              1.000000         -0.256131     0.671703   
volatile acidity          -0.256131          1.000000    -0.552496   
citric acid                0.671703         -0.552496     1.000000   
residual sugar             0.114777          0.001918     0.143577   
chlorides                  0.093705          0.061298     0.203823   
free sulfur dioxide       -0.153794         -0.010504    -0.060978   
total sulfur dioxide      -0.113181          0.076470     0.035533   
density                    0.668047          0.022026     0.364947   
pH                        -0.682978          0.234937    -0.541904   
sulphates                  0.183006         -0.260987     0.312770   
alcohol                   -0.061668         -0.202288     0.109903   
quality                    0.124052         -0.390558     0.226373   
quality_binary             0.095093         -0.321441     0.159129   

                      residual sugar  chlorides  free sulfur dioxide  \
fixed acidity               0.114777   0.093705            -0.153794   
volatile acidity            0.001918   0.061298            -0.010504   
citric acid                 0.143577   0.203823            -0.060978   
residual sugar              1.000000   0.055610             0.187049   
chlorides                   0.055610   1.000000             0.005562   
free sulfur dioxide         0.187049   0.005562             1.000000   
total sulfur dioxide        0.203028   0.047400             0.667666   
density                     0.355283   0.200632            -0.021946   
pH                         -0.085652  -0.265026             0.070377   
sulphates                   0.005527   0.371260             0.051658   
alcohol                     0.042075  -0.221141            -0.069408   
quality                     0.013732  -0.128907            -0.050656   
quality_binary             -0.002160  -0.109494            -0.061757   
```



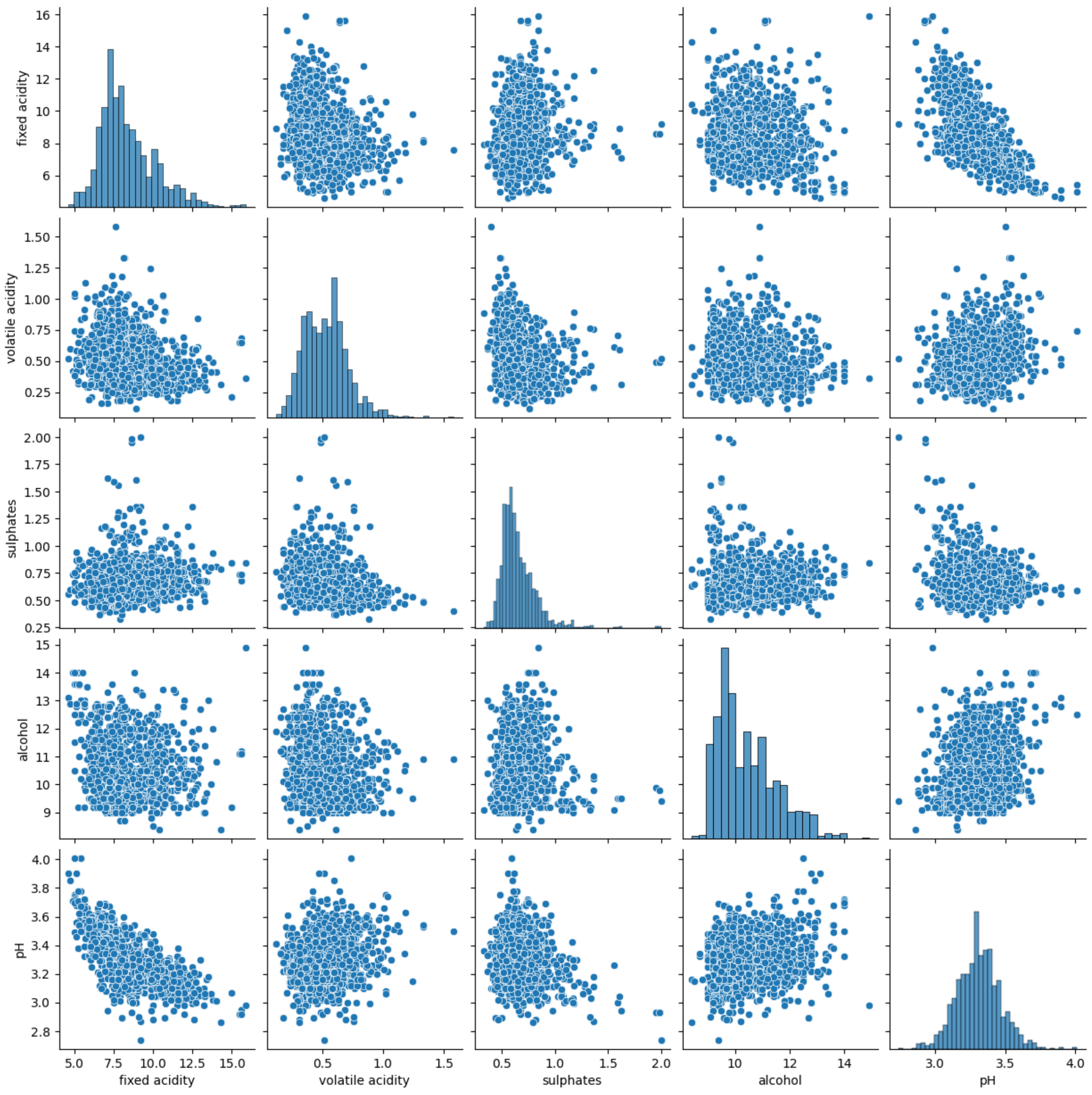
```
                      total sulfur dioxide   density        pH  sulphates  \
fixed acidity                    -0.113181  0.668047 -0.682978   0.183006   
volatile acidity                  0.076470  0.022026  0.234937  -0.260987   
citric acid                       0.035533  0.364947 -0.541904   0.312770   
residual sugar                    0.203028  0.355283 -0.085652   0.005527   
chlorides                         0.047400  0.200632 -0.265026   0.371260   
free sulfur dioxide               0.667666 -0.021946  0.070377   0.051658   
total sulfur dioxide              1.000000  0.071269 -0.066495   0.042947   
density                           0.071269  1.000000 -0.341699   0.148506   
pH                               -0.066495 -0.341699  1.000000  -0.196648   
sulphates                         0.042947  0.148506 -0.196648   1.000000   
alcohol                          -0.205654 -0.496180  0.205633   0.093595   
quality                          -0.185100 -0.174919 -0.057731   0.251397   
quality_binary                   -0.231963 -0.159110 -0.003264   0.218072   

                       alcohol   quality  quality_binary  
fixed acidity        -0.061668  0.124052        0.095093  
volatile acidity     -0.202288 -0.390558       -0.321441  
citric acid           0.109903  0.226373        0.159129  
residual sugar        0.042075  0.013732       -0.002160  
chlorides            -0.221141 -0.128907       -0.109494  
free sulfur dioxide  -0.069408 -0.050656       -0.061757  
total sulfur dioxide -0.205654 -0.185100       -0.231963  
density              -0.496180 -0.174919       -0.159110  
pH                    0.205633 -0.057731       -0.003264  
sulphates             0.093595  0.251397        0.218072  
alcohol               1.000000  0.476166        0.434751  
quality               0.476166  1.000000        0.848279  
quality_binary        0.434751  0.848279        1.000000  
```



	fixed acidity	volatile acidity	sulphates	alcohol	pH
fixed acidity	1.00	-0.26	0.18	-0.06	-0.68
volatile acidity	-0.26	1.00	-0.26	-0.20	0.23
sulphates	0.18	-0.26	1.00	0.09	-0.20
alcohol	-0.06	-0.20	0.09	1.00	0.21
pH	-0.68	0.23	-0.20	0.21	1.00

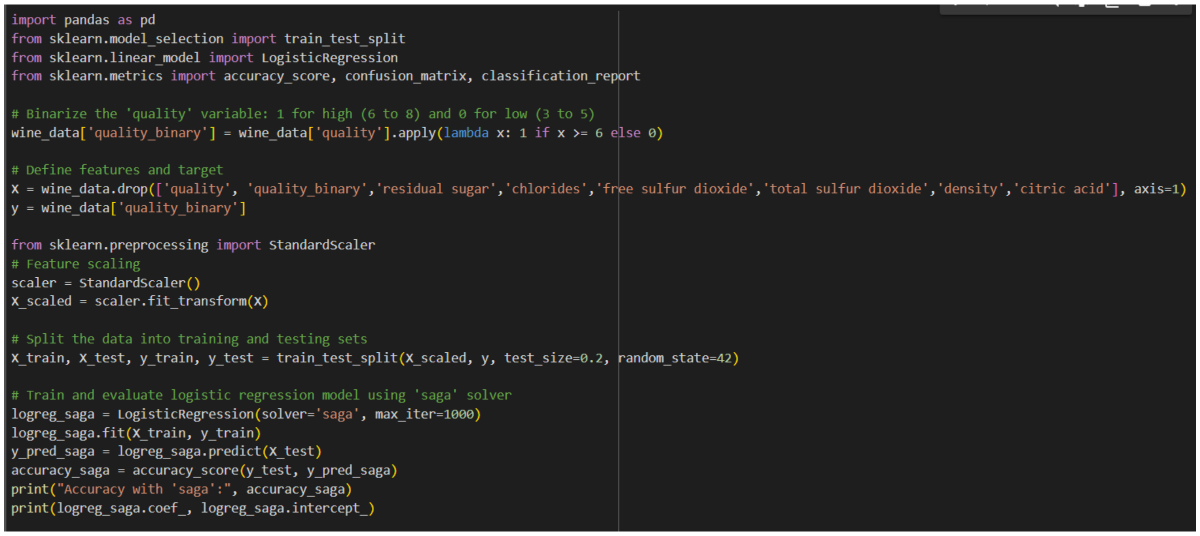
I selected below 5 features to analyze the wine quality data set.

- Fixed acidity
- Volatile acidity
- Sulphates
- Alcohol
- quality



Pair plot of Selected feature

3)

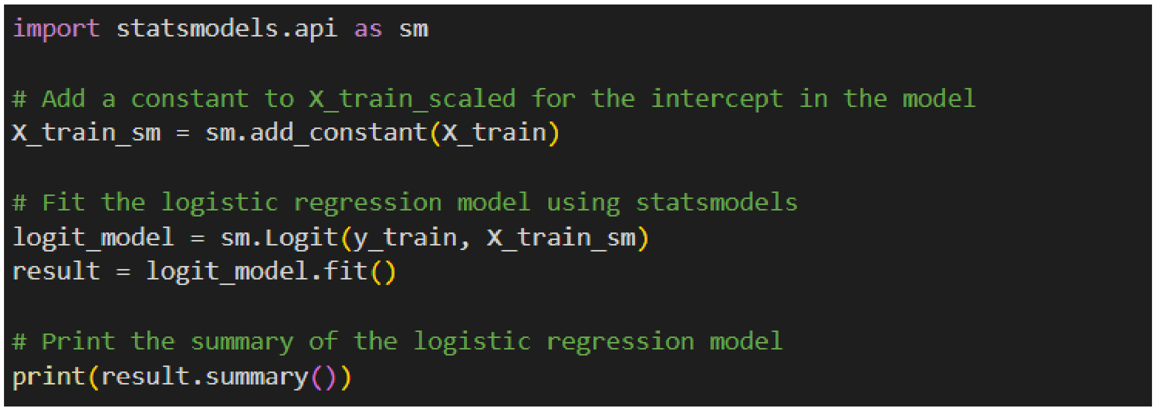


```python
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

# Binarize the 'quality' variable: 1 for high (6 to 8) and 0 for low (3 to 5)
wine_data['quality_binary'] = wine_data['quality'].apply(lambda x: 1 if x >= 6 else 0)

# Define features and target
X = wine_data.drop(['quality', 'quality_binary','residual sugar','chlorides','free sulfur dioxide','total sulfur dioxide','density','citric acid'], axis=1)
y = wine_data['quality_binary']

from sklearn.preprocessing import StandardScaler
# Feature scaling
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

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

# Train and evaluate logistic regression model using 'saga' solver
logreg_saga = LogisticRegression(solver='saga', max_iter=1000)
logreg_saga.fit(X_train, y_train)
y_pred_saga = logreg_saga.predict(X_test)
accuracy_saga = accuracy_score(y_test, y_pred_saga)
print("Accuracy with 'saga':", accuracy_saga)
print(logreg_saga.coef_, logreg_saga.intercept_)
```

In training my logistic regression model, I found no missing values in the dataset, eliminating the need for data cleaning to remove rows, which helped maintain model integrity. I selected five integer features for the model, and since there were no categorical features, no feature encoding was necessary. Given the large size and complexity of my dataset, I chose the 'saga' solver, which is well-suited for such conditions. To prevent any single feature from dominating due to its scale, I implemented feature scaling using the 'StandardScaler'. Notably, this scaling improved the model's accuracy. Before scaling, the accuracy was 72.5%, and after applying standard scaling, it increased to 73.125%.

Another one special thing is in my target variable there are multiple categories regarding quality of wine. So I recategorize those values when the quality is greater then 6 it converts to 1 indicating high quality. Otherwise 0 indicating low quality.

4)



```python
import statsmodels.api as sm

# Add a constant to X_train_scaled for the intercept in the model
X_train_sm = sm.add_constant(X_train)

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

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

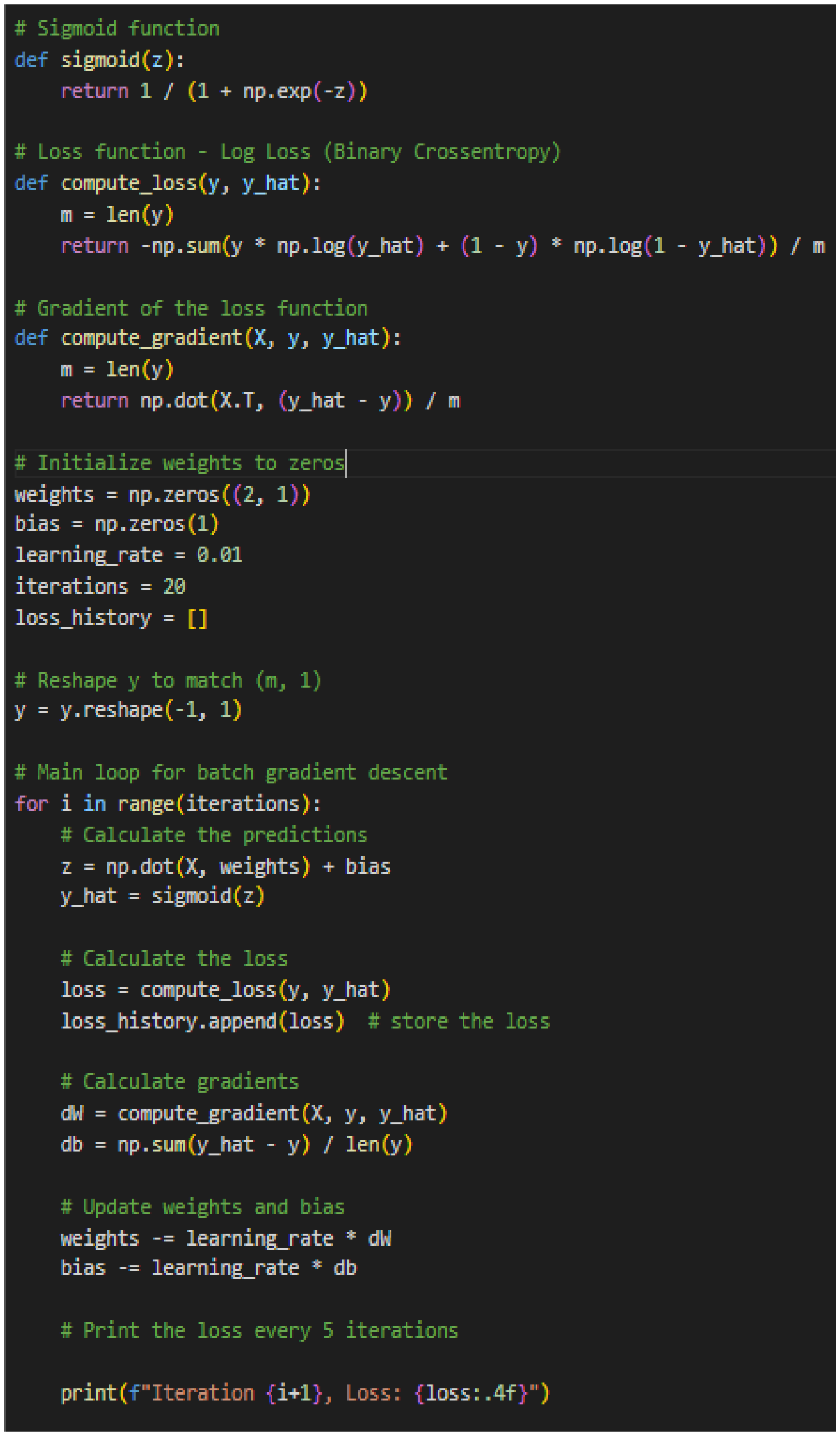
Fixed Acidity	0.052
Volatile Acidity	0.000
PH	0.255
Sulphates	0.000
Alcohol	0.000

Fixed acidity has a p-value that is slightly above the accepted cutoff of 0.05, indicating that it is marginally significant. This implies that it is not statistically significant, even though it might have some effect on the model. The response variable is inversely correlated with volatile acidity, which is extremely significant. In the model, it is an important predictor. The pH is not a significant predictor of the result, as seen by its high p-value. This feature could be removed from the model without compromising its efficacy because it does not statistically add to the model. Additionally, alcohol has a strong beneficial impact on the response variable. It has a big impact on forecasting the result.

3) Logistic regression First/Second-Order Methods

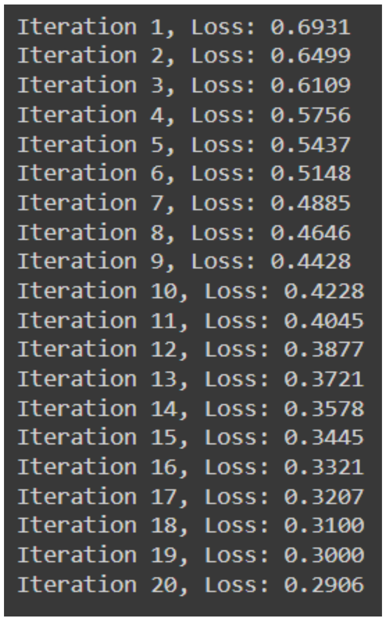
1) Generated the data using code given in listing 4.

2)



```python
# Sigmoid function
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# Loss function - Log Loss (Binary Crossentropy)
def compute_loss(y, y_hat):
    m = len(y)
    return -np.sum(y * np.log(y_hat) + (1 - y) * np.log(1 - y_hat)) / m

# Gradient of the loss function
def compute_gradient(X, y, y_hat):
    m = len(y)
    return np.dot(X.T, (y_hat - y)) / m

# Initialize weights to zeros
weights = np.zeros((2, 1))
bias = np.zeros(1)
learning_rate = 0.01
iterations = 20
loss_history = []

# Reshape y to match (m, 1)
y = y.reshape(-1, 1)

# Main loop for batch gradient descent
for i in range(iterations):
    # Calculate the predictions
    z = np.dot(X, weights) + bias
    y_hat = sigmoid(z)

    # Calculate the loss
    loss = compute_loss(y, y_hat)
    loss_history.append(loss)  # store the loss

    # Calculate gradients
    dW = compute_gradient(X, y, y_hat)
    db = np.sum(y_hat - y) / len(y)

    # Update weights and bias
    weights -= learning_rate * dW
    bias -= learning_rate * db

    # Print the loss every 5 iterations

    print(f"Iteration {i+1}, Loss: {loss:.4f}")
```

Loss in each iteration:



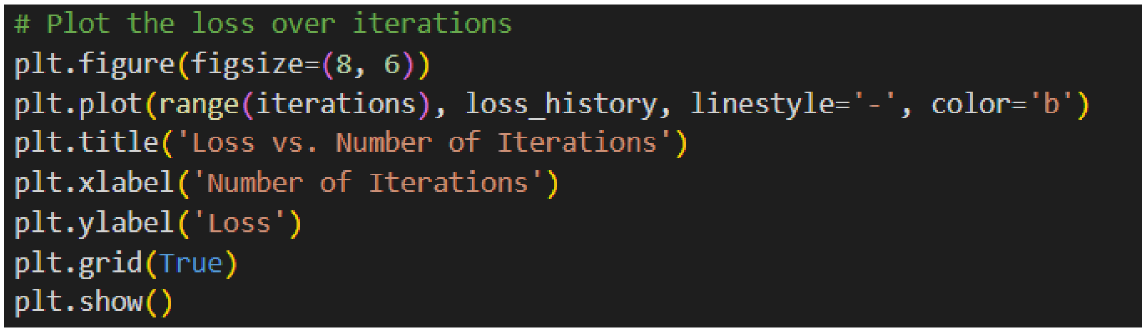
```
Iteration 1, Loss: 0.6931
Iteration 2, Loss: 0.6499
Iteration 3, Loss: 0.6109
Iteration 4, Loss: 0.5756
Iteration 5, Loss: 0.5437
Iteration 6, Loss: 0.5148
Iteration 7, Loss: 0.4885
Iteration 8, Loss: 0.4646
Iteration 9, Loss: 0.4428
Iteration 10, Loss: 0.4228
Iteration 11, Loss: 0.4045
Iteration 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
```

I initialized weights and bias to zeros.

Initializing weights and biases to zeros in logistic regression works well because it ensures symmetry . The cost function is convex, so gradient descent will still find the global minimum, and zero initialization doesn't prevent learning effectively. This simplicity makes zero initialization a good choice for logistic regression.

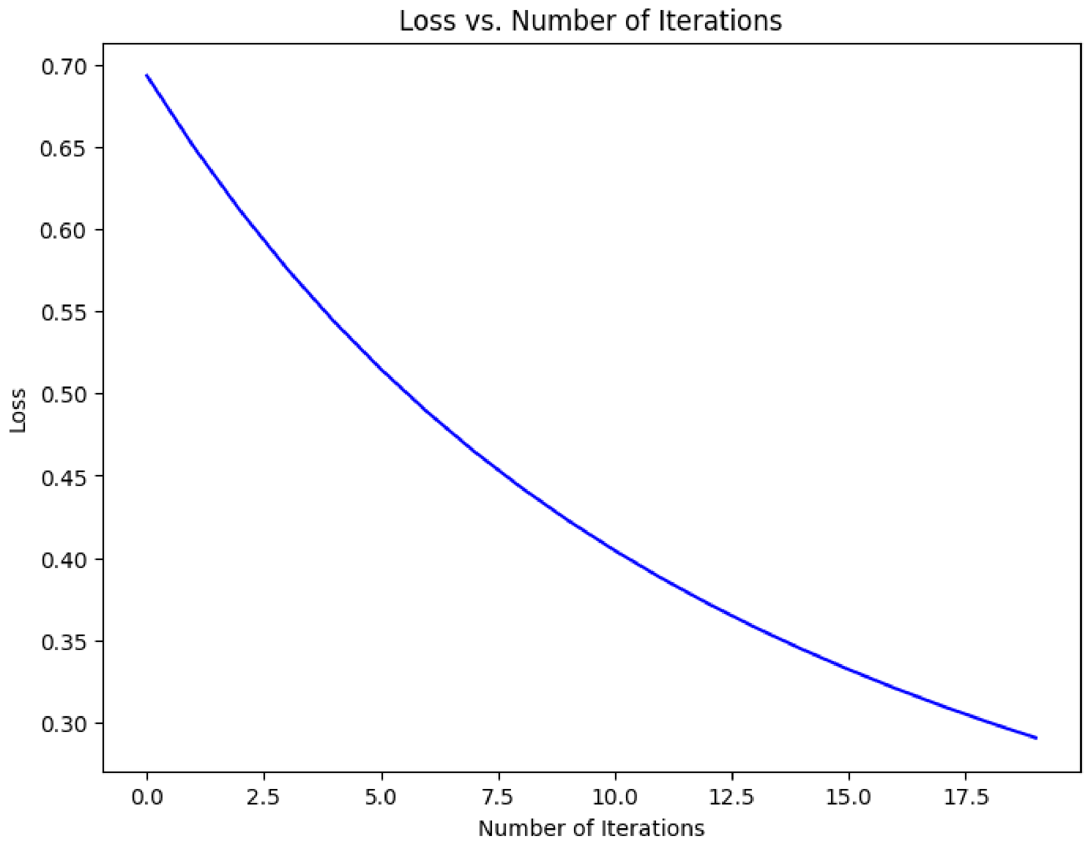
3) I used Logistic Loss function to reduce the error. Why I used Loss function in my binary classification problem is because it measures the difference between predicted probabilities and actual labels (good quality vs. bad quality). It complements the sigmoid function, penalizing incorrect predictions and ensuring smooth, differentiable gradients for batch gradient descent. This allows efficient updates to weights and biases, ensuring stable convergence to an optimal solution, as logistic loss provides a convex cost function with a guaranteed global minimum.

4)

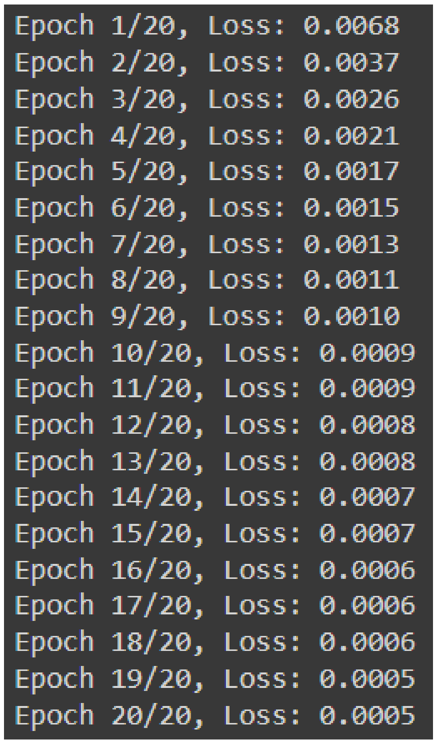


```python
# Plot the loss over iterations
plt.figure(figsize=(8, 6))
plt.plot(range(iterations), loss_history, linestyle='-', color='b')
plt.title('Loss vs. Number of Iterations')
plt.xlabel('Number of Iterations')
plt.ylabel('Loss')
plt.grid(True)
plt.show()
```

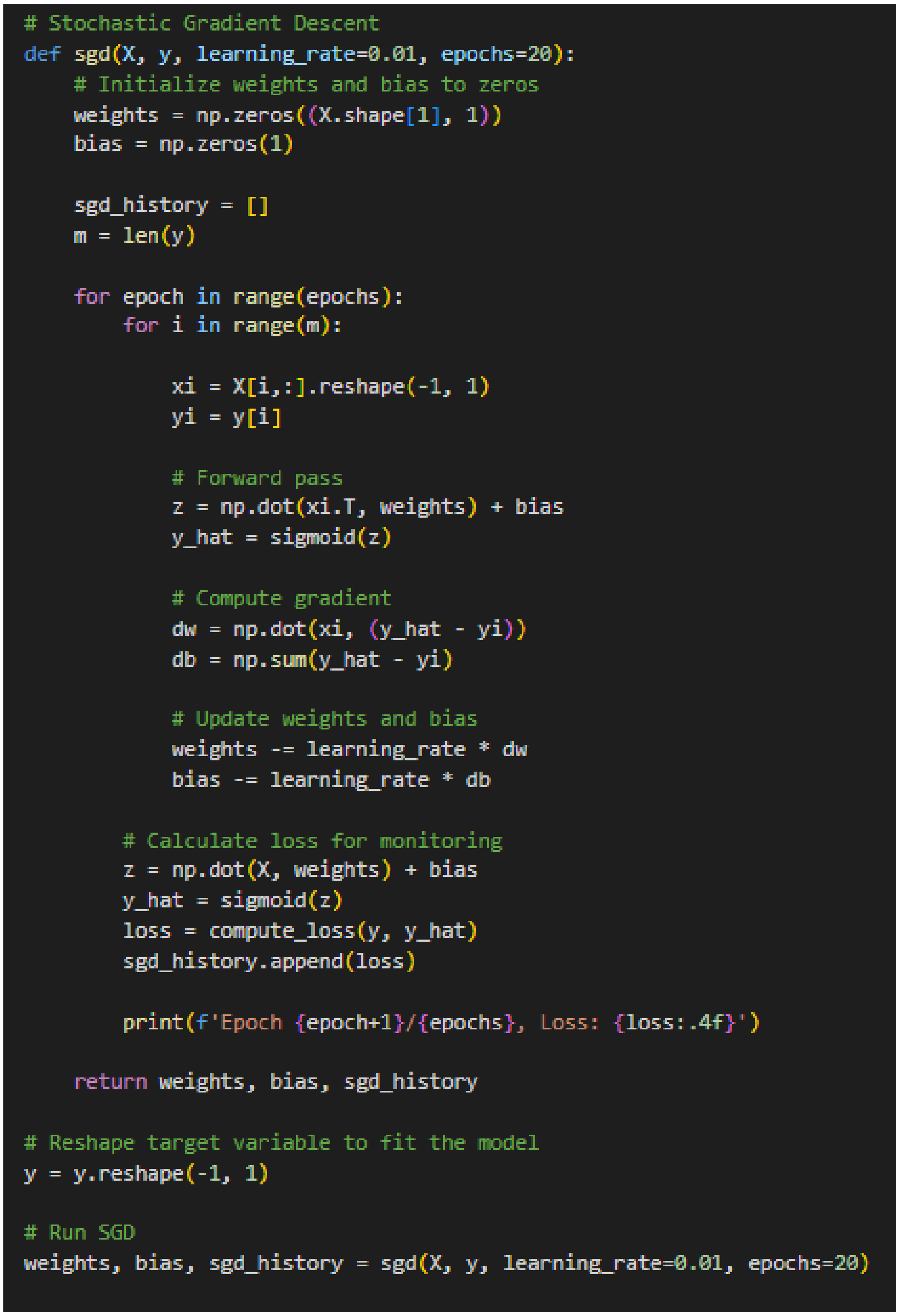
Graph of loses with respect to number of iterations



5) Loss in each iteration:

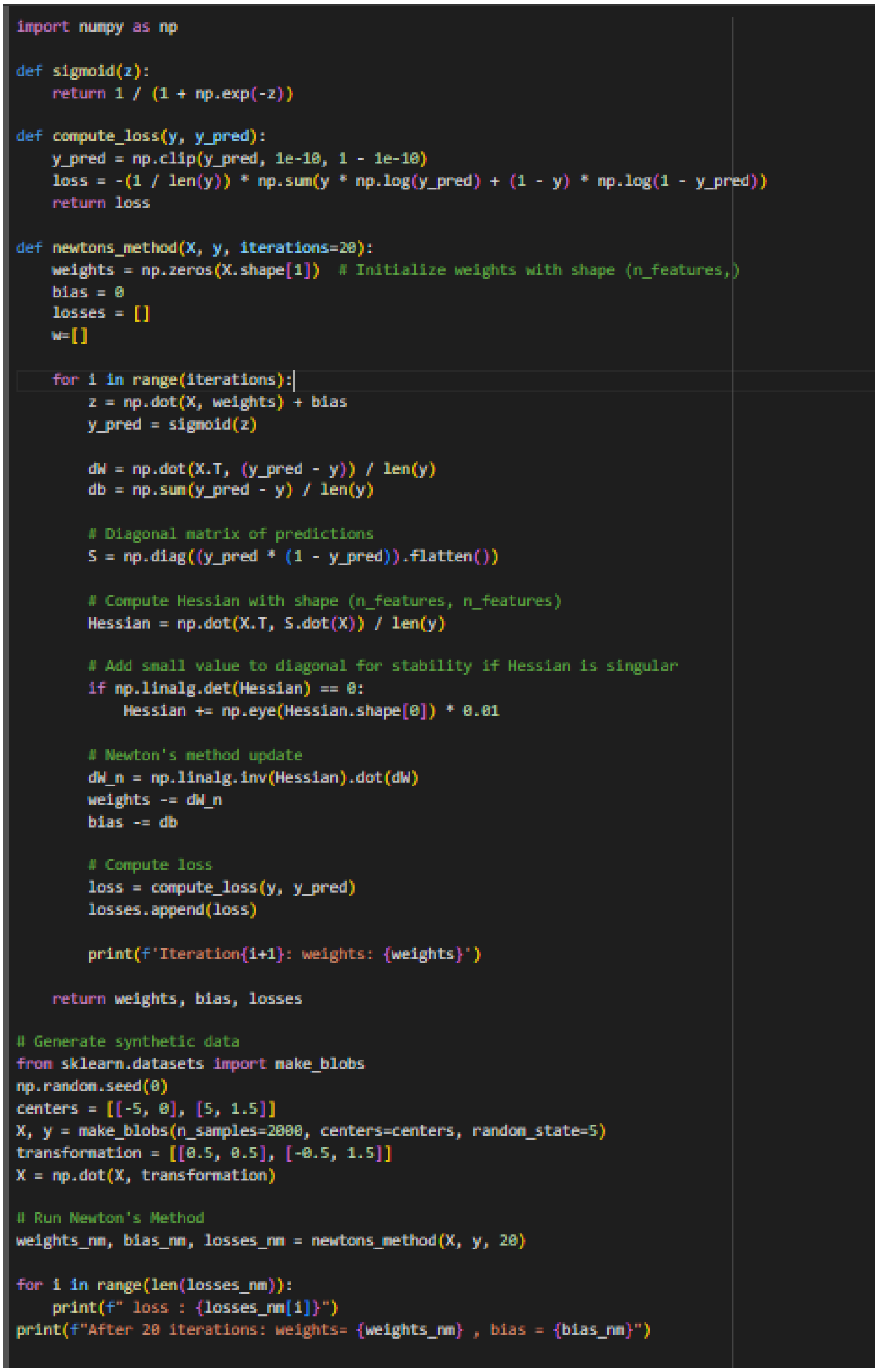


```
Epoch 1/20, Loss: 0.0068
Epoch 2/20, Loss: 0.0037
Epoch 3/20, Loss: 0.0026
Epoch 4/20, Loss: 0.0021
Epoch 5/20, Loss: 0.0017
Epoch 6/20, Loss: 0.0015
Epoch 7/20, Loss: 0.0013
Epoch 8/20, Loss: 0.0011
Epoch 9/20, Loss: 0.0010
Epoch 10/20, Loss: 0.0009
Epoch 11/20, Loss: 0.0009
Epoch 12/20, Loss: 0.0008
Epoch 13/20, Loss: 0.0008
Epoch 14/20, Loss: 0.0007
Epoch 15/20, Loss: 0.0007
Epoch 16/20, Loss: 0.0006
Epoch 17/20, Loss: 0.0006
Epoch 18/20, Loss: 0.0006
Epoch 19/20, Loss: 0.0005
Epoch 20/20, Loss: 0.0005
```



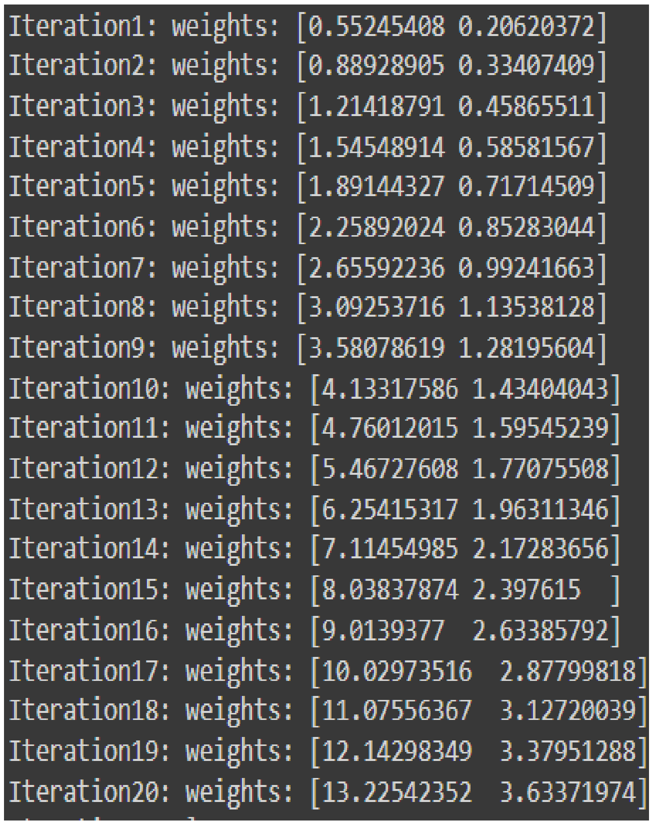
```
# Stochastic Gradient Descent
def sgd(X, y, learning_rate=0.01, epochs=20):
    # Initialize weights and bias to zeros
    weights = np.zeros((X.shape[1], 1))
    bias = np.zeros(1)

    sgd_history = []
    m = len(y)

    for epoch in range(epochs):
        for i in range(m):

            xi = X[i,:].reshape(-1, 1)
            yi = y[i]

            # Forward pass
            z = np.dot(xi.T, weights) + bias
            y_hat = sigmoid(z)

            # Compute gradient
            dw = np.dot(xi, (y_hat - yi))
            db = np.sum(y_hat - yi)

            # Update weights and bias
            weights -= learning_rate * dw
            bias -= learning_rate * db

        # Calculate loss for monitoring
        z = np.dot(X, weights) + bias
        y_hat = sigmoid(z)
        loss = compute_loss(y, y_hat)
        sgd_history.append(loss)

        print(f'Epoch {epoch+1}/{epochs}, Loss: {loss:.4f}')

    return weights, bias, sgd_history

# Reshape target variable to fit the model
y = y.reshape(-1, 1)

# Run SGD
weights, bias, sgd_history = sgd(X, y, learning_rate=0.01, epochs=20)
```

6)

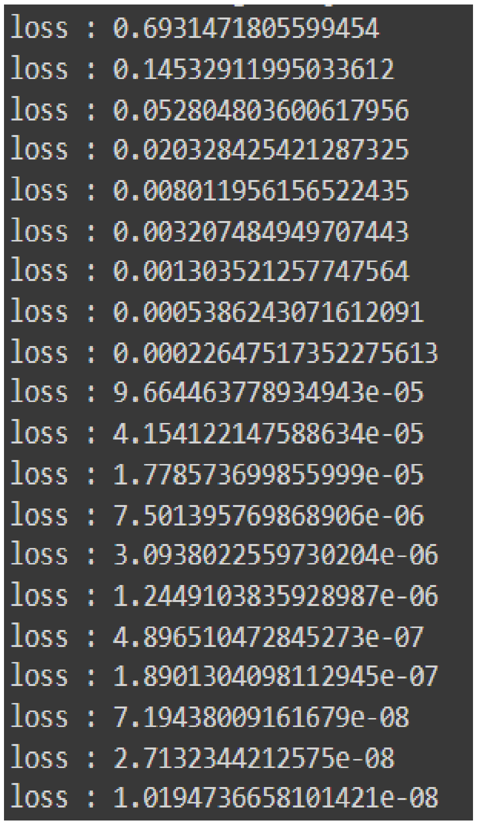


```python
import numpy as np

def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def compute_loss(y, y_pred):
    y_pred = np.clip(y_pred, 1e-10, 1 - 1e-10)
    loss = -(1 / len(y)) * np.sum(y * np.log(y_pred) + (1 - y) * np.log(1 - y_pred))
    return loss

def newtons_method(X, y, iterations=20):
    weights = np.zeros(X.shape[1])  # Initialize weights with shape (n_features,)
    bias = 0
    losses = []
    w=[]

    for i in range(iterations):
        z = np.dot(X, weights) + bias
        y_pred = sigmoid(z)

        dW = np.dot(X.T, (y_pred - y)) / len(y)
        db = np.sum(y_pred - y) / len(y)

        # Diagonal matrix of predictions
        S = np.diag((y_pred * (1 - y_pred)).flatten())

        # Compute Hessian with shape (n_features, n_features)
        Hessian = np.dot(X.T, S.dot(X)) / len(y)

        # Add small value to diagonal for stability if Hessian is singular
        if np.linalg.det(Hessian) == 0:
            Hessian += np.eye(Hessian.shape[0]) * 0.01

        # Newton's method update
        dW_n = np.linalg.inv(Hessian).dot(dW)
        weights -= dW_n
        bias -= db

        # Compute loss
        loss = compute_loss(y, y_pred)
        losses.append(loss)

        print(f'Iteration{i+1}: weights: {weights}')

    return weights, bias, losses

# Generate synthetic data
from sklearn.datasets import make_blobs
np.random.seed(0)
centers = [[-5, 0], [5, 1.5]]
X, y = make_blobs(n_samples=2000, centers=centers, random_state=5)
transformation = [[0.5, 0.5], [-0.5, 1.5]]
X = np.dot(X, transformation)

# Run Newton's Method
weights_nm, bias_nm, losses_nm = newtons_method(X, y, 20)

for i in range(len(losses_nm)):
    print(f" loss : {losses_nm[i]}")
print(f"After 20 iterations: weights= {weights_nm} , bias = {bias_nm}")
```

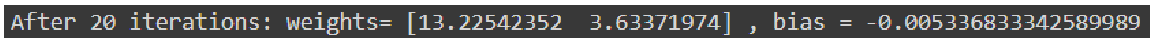
Loss of Each iterations:
Weights in each iteration:



```
Iteration1: weights: [0.55245408 0.20620372]
Iteration2: weights: [0.88928905 0.33407409]
Iteration3: weights: [1.21418791 0.45865511]
Iteration4: weights: [1.54548914 0.58581567]
Iteration5: weights: [1.89144327 0.71714509]
Iteration6: weights: [2.25892024 0.85283044]
Iteration7: weights: [2.65592236 0.99241663]
Iteration8: weights: [3.09253716 1.13538128]
Iteration9: weights: [3.58078619 1.28195604]
Iteration10: weights: [4.13317586 1.43404043]
Iteration11: weights: [4.76012015 1.59545239]
Iteration12: weights: [5.46727608 1.77075508]
Iteration13: weights: [6.25415317 1.96311346]
Iteration14: weights: [7.11454985 2.17283656]
Iteration15: weights: [8.03837874 2.397615  ]
Iteration16: weights: [9.0139377  2.63385792]
Iteration17: weights: [10.02973516  2.87799818]
Iteration18: weights: [11.07556367  3.12720039]
Iteration19: weights: [12.14298349  3.37951288]
Iteration20: weights: [13.22542352  3.63371974]
```

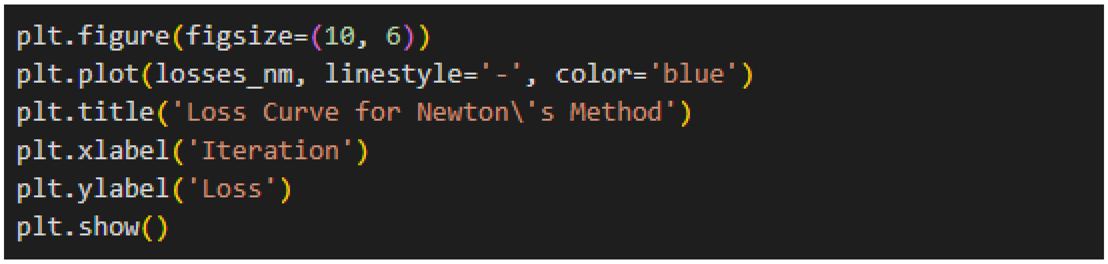


```
loss : 0.6931471805599454
loss : 0.14532911995033612
loss : 0.052804803600617956
loss : 0.020328425421287325
loss : 0.008011956156522435
loss : 0.003207484949707443
loss : 0.001303521257747564
loss : 0.0005386243071612091
loss : 0.00022647517352275613
loss : 9.664463778934943e-05
loss : 4.154122147588634e-05
loss : 1.778573699855999e-05
loss : 7.501395769868906e-06
loss : 3.0938022559730204e-06
loss : 1.2449103835928987e-06
loss : 4.896510472845273e-07
loss : 1.8901304098112945e-07
loss : 7.19438009161679e-08
loss : 2.7132344212575e-08
loss : 1.0194736658101421e-08
```



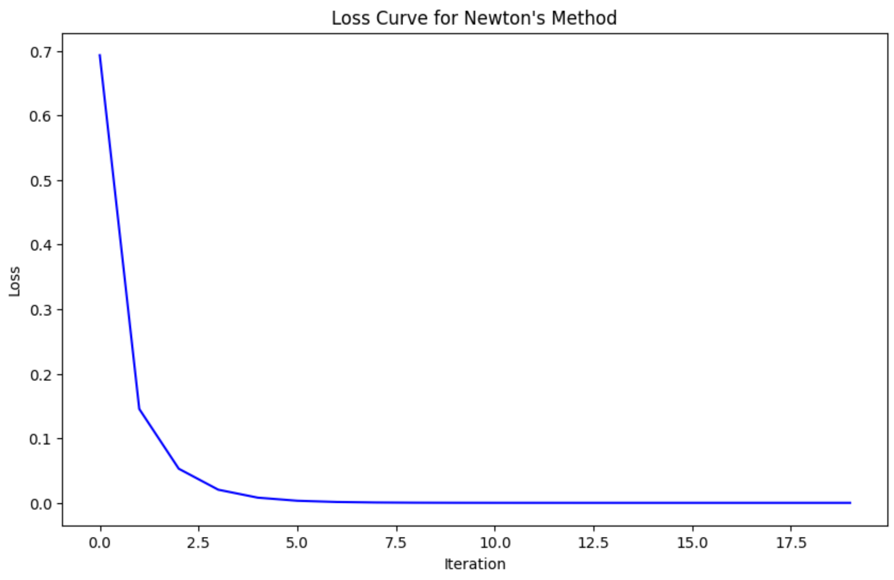
```
After 20 iterations: weights= [13.22542352  3.63371974] , bias = -0.005336833342589989
```

7)

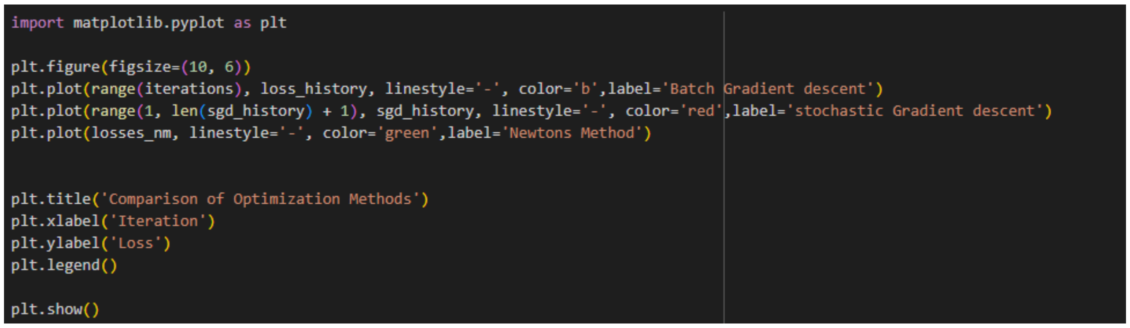


```
plt.figure(figsize=(10, 6))
plt.plot(losses_nm, linestyle='-', color='blue')
plt.title('Loss Curve for Newton\'s Method')
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.show()
```

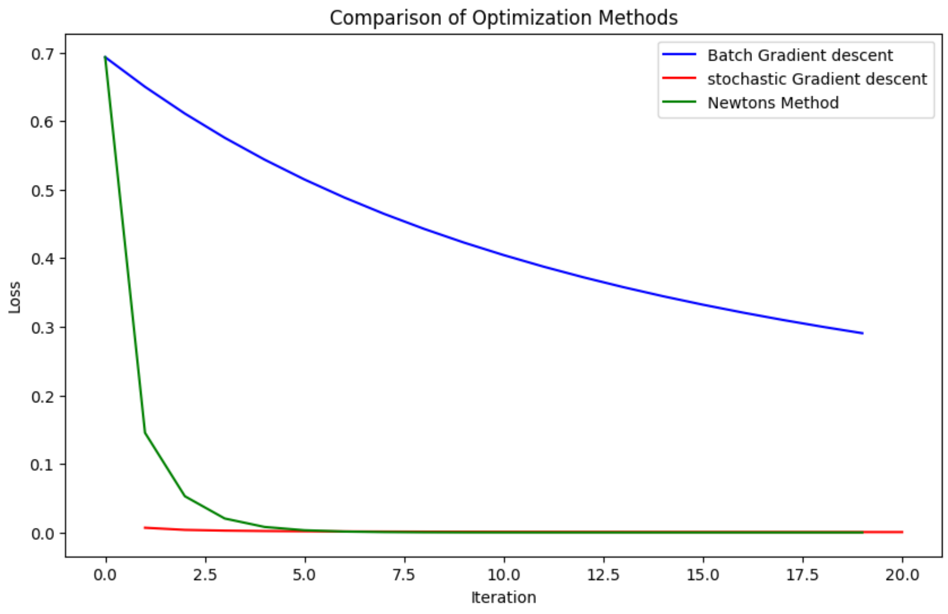
Graph of losses with respect to number of iterations



8)



```
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 6))
plt.plot(range(iterations), loss_history, linestyle='-', color='b',label='Batch Gradient descent')
plt.plot(range(1, len(sgd_history) + 1), sgd_history, linestyle='-', color='red',label='stochastic Gradient descent')
plt.plot(losses_nm, linestyle='-', color='green',label='Newtons Method')


plt.title('Comparison of Optimization Methods')
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.legend()

plt.show()
```



Newton’s method exhibits the fastest convergence among the three, quickly reducing the loss and stabilizing near zero within just a few iterations. This demonstrates the method's efficiency, as it uses second-order derivatives to make more precise weight updates.

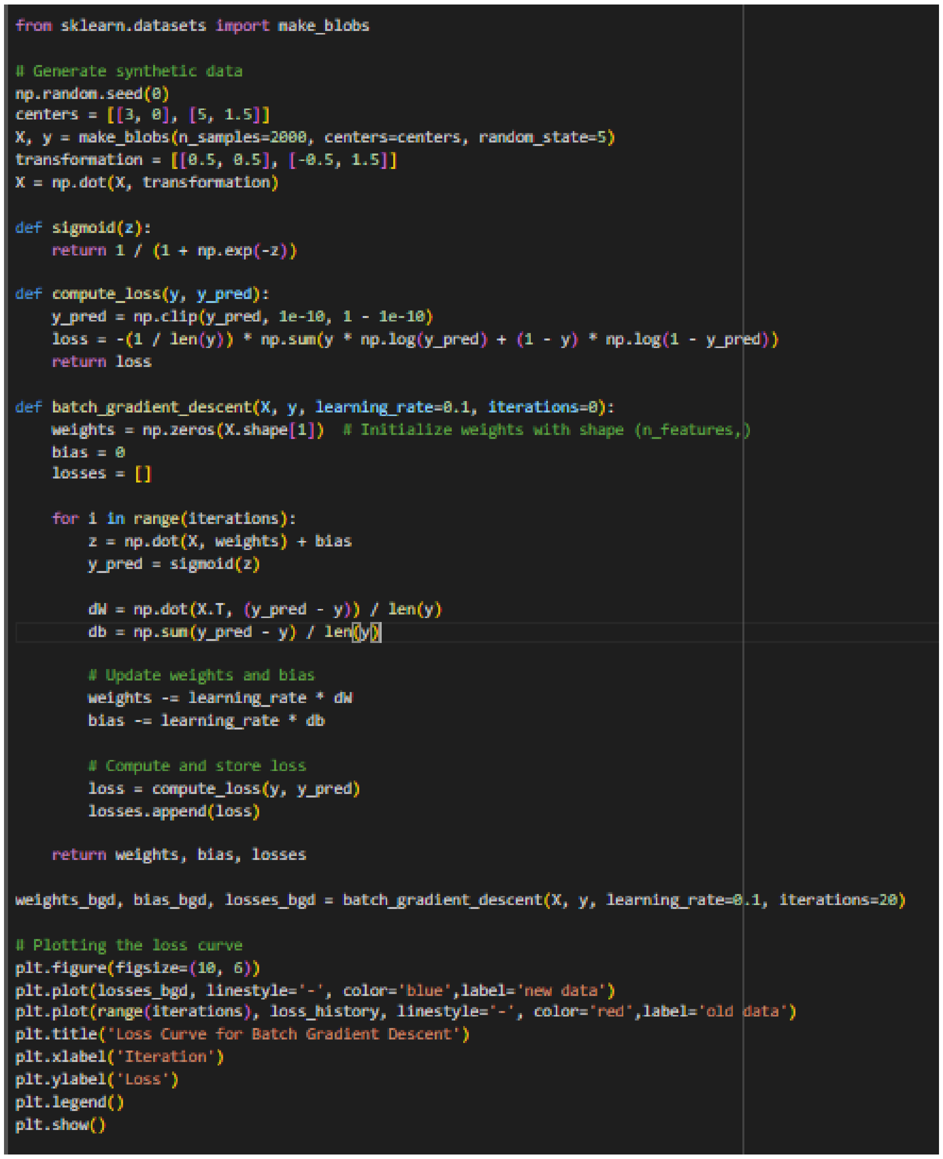
Stochastic Gradient Descent (SGD) converges more quickly than Batch Gradient Descent, with the loss dropping from 0.0068 in the first epoch to 0.0005 by the 20th. This rapid progress is typical of SGD, as it updates weights after processing each data point.

Batch Gradient Descent shows the slowest convergence, with the loss decreasing steadily but more gradually than the other methods. This is expected since BGD calculates gradients over the entire dataset in each iteration, leading to stable but slower movement towards the optimal solution.

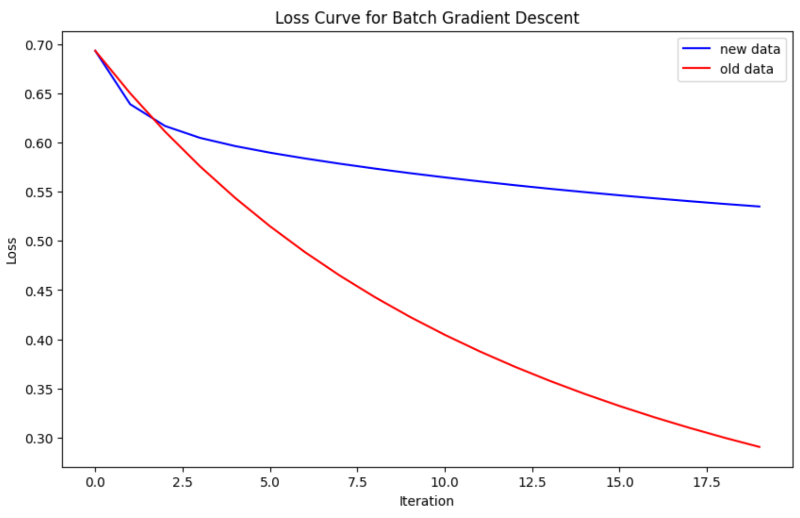
9) **For Gradient Descent:** Two useful methods can be applied to identify the ideal number of iterations for Gradient Descent. In the first, a convergence threshold is determined by taking the loss tolerance into account. This technique maximizes computer resources by stopping the process when improvements become insignificant, when the difference in loss between the next iteration drops below a predefined modest number. In the second method, model performance is measured during training using a validation set. When validation performance declines or stops getting better, suggesting possible overfitting or convergence, the training is stopped. This technique guarantees that the model's generalization is maximized while also aiding in the prevention of overfitting.

For Newton's Method: For Newton’s Method, choosing the correct number of iterations can be based on the properties of the Hessian matrix. One method is to keep an eye on the size of the elements in the Hessian matrix and stop the iterations when the norm drops below a certain threshold. This usually signifies that the loss function's minimum is getting closer. Another technique involves observing the condition number of the Hessian matrix, a high condition number may signify numerical instability or poor convergence, requiring an adjustment or termination of iterations. These methods help to improve the stability and efficiency of the optimization process and are based on the theoretical foundations of Newton's Method.

10) The provided loss curve comparison for Batch Gradient Descent using updated centers shows that the new data, with centers closer together, converges slower and plateaus at a higher loss compared to the old data. This behavior likely results from the reduced separability in the new dataset configuration, where the proximity of the clusters makes it more challenging for the logistic regression model to effectively distinguish between classes. Consequently, the gradient updates do not decrease the loss as efficiently, resulting in a higher final loss and indicating the impact of data distribution on the convergence and effectiveness of the learning algorithm. This underscores the need for considering data characteristics in model training and possibly adopting more sophisticated models or preprocessing techniques to handle less separable data efficiently.



```
from sklearn.datasets import make_blobs

# 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)
transformation = [[0.5, 0.5], [-0.5, 1.5]]
X = np.dot(X, transformation)

def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def compute_loss(y, y_pred):
    y_pred = np.clip(y_pred, 1e-10, 1 - 1e-10)
    loss = -(1 / len(y)) * np.sum(y * np.log(y_pred) + (1 - y) * np.log(1 - y_pred))
    return loss

def batch_gradient_descent(X, y, learning_rate=0.1, iterations=0):
    weights = np.zeros(X.shape[1])  # Initialize weights with shape (n_features,)
    bias = 0
    losses = []

    for i in range(iterations):
        z = np.dot(X, weights) + bias
        y_pred = sigmoid(z)

        dW = np.dot(X.T, (y_pred - y)) / len(y)
        db = np.sum(y_pred - y) / len(y)

        # Update weights and bias
        weights -= learning_rate * dW
        bias -= learning_rate * db

        # Compute and store loss
        loss = compute_loss(y, y_pred)
        losses.append(loss)

    return weights, bias, losses

weights_bgd, bias_bgd, losses_bgd = batch_gradient_descent(X, y, learning_rate=0.1, iterations=20)

# Plotting the loss curve
plt.figure(figsize=(10, 6))
plt.plot(losses_bgd, linestyle='-', color='blue',label='new data')
plt.plot(range(iterations), loss_history, linestyle='-', color='red',label='old data')
plt.title('Loss Curve for Batch Gradient Descent')
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.legend()
plt.show()
```


Loss Curve for Batch Gradient Descent
new data
old data
Loss
Iteration