In [258…

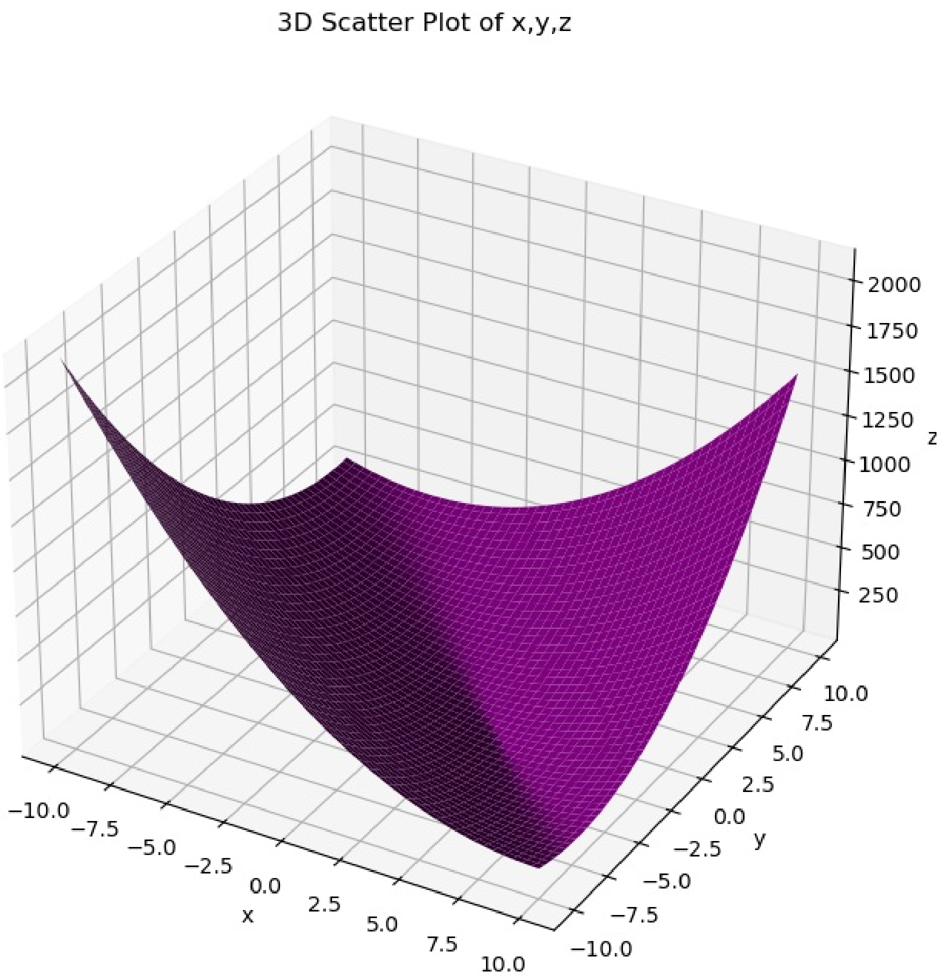
```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from mpl_toolkits.mplot3d import Axes3D
from sklearn.metrics import confusion_matrix
```

1.

In [ ]:

In [3]:

```python
# Define the function
def f(x, y):
    return (x + 2 * y) ** 2 + (2 * x + y - 5) ** 2

# Generate a grid of x and y values
x_range = np.linspace(-10, 10, 200)
y_range = np.linspace(-10, 10, 200)
x, y = np.meshgrid(x_range, y_range)  # Creates 2D arrays for x and y

z = f(x, y)  # Apply the function to the entire grid

fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')


surface = ax.plot_surface(x, y, z, color='purple')

ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.set_title('3D Scatter Plot of x,y,z')

plt.show()
```



In [ ]:

b.

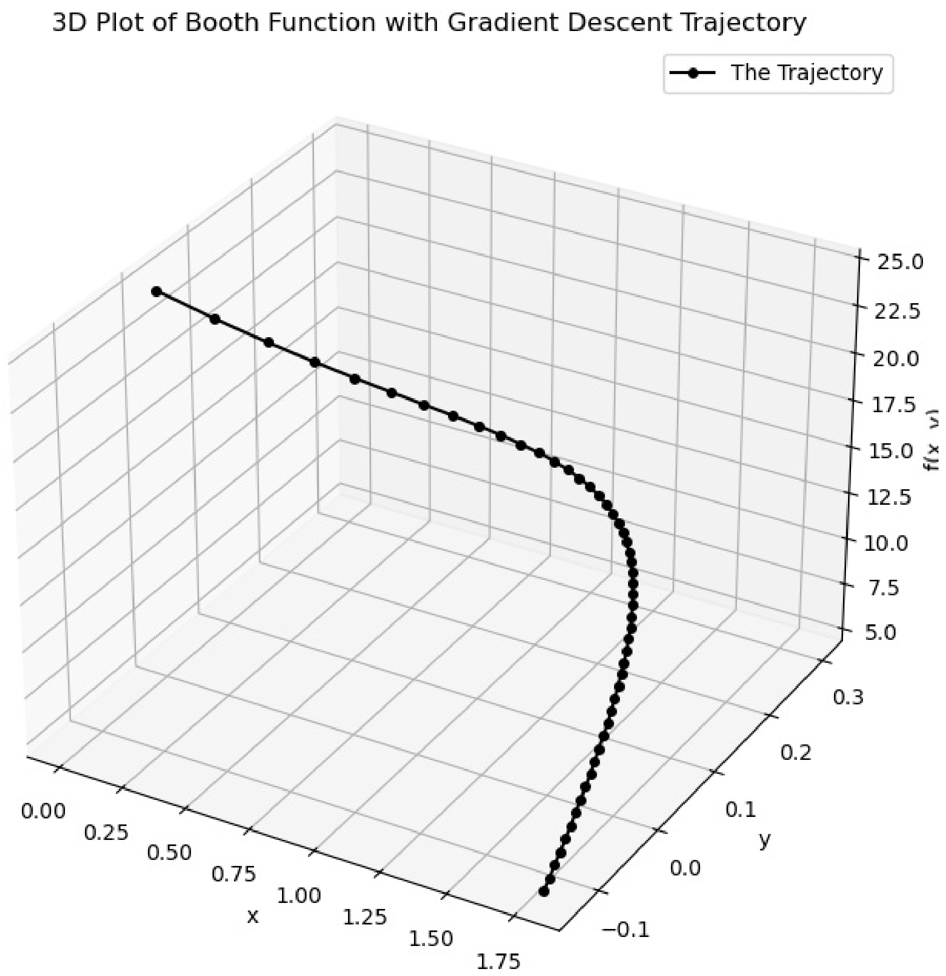
In [ ]:

In [6]:
```python
# Booth function definition
def booth_function(x, y):
    return (x + 2 * y) ** 2 + (2 * x + y - 5) ** 2

# Partial derivatives
def booth_gradient(x, y):
    df_dx = 5 * x + 4 * y - 10  # Partial Derivative with respect to x
    df_dy = 4 * x + 4 * y - 5   # Partial Derivative with respect to y
    return np.array([df_dx, df_dy])

# Gradient descent algorithm
def gradient_descent(learning_rate=0.01, epochs=10):
    beta = np.array([0, 0])

    for _ in range(epochs):
        grad = booth_gradient(beta[0], beta[1])
        beta = beta - learning_rate * grad
        #print(path)
      #  points.append(point)
    return (beta)

gradient_descent()
```

Out[6]: array([0.74624509, 0.28396592])

In [ ]:

In [278…
```python
# Booth function definition
def booth_function(x, y):
    return (x + 2 * y) ** 2 + (2 * x + y - 5) ** 2

# Partial derivatives (gradient) of the Booth function
def booth_gradient(x, y):
    df_dx = 5 * x + 4 * y - 10  # Partial Derivative with respect to x
    df_dy = 4 * x + 4 * y - 5   # Partial Derivative with respect to y
    return np.array([df_dx, df_dy])

def gradient_descent(learning_rate=0.01, epochs=50):
    beta = np.array([0.0, 0.0])  # Starting point
    trajectory = [beta.copy()]  # List to store each point in the trajectory

    for _ in range(epochs):
        grad = booth_gradient(beta[0], beta[1])
        beta = beta - learning_rate * grad
        trajectory.append(beta.copy())  # Save current position in the trajectory

    return np.array(trajectory)

trajectory = gradient_descent()

# Plotting the Booth function with the trajectory
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')

# Extracting the x, y, and z values from the trajectory
traj_x = trajectory[:, 0]
traj_y = trajectory[:, 1]
traj_z = booth_function(traj_x, traj_y)

# Plot the trajectory on the 3D surface
ax.plot(traj_x, traj_y, traj_z, 'o-', color='black', markersize=4, label="The Trajectory")

ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('f(x, y)')
ax.set_title('3D Plot of Booth Function with Gradient Descent Trajectory')

plt.legend()
plt.show()
```



In [ ]:

Step Length Function

In [286…

```
def prediction(x, beta):
    z = np.dot(x, beta)
    return z

def mean_square_loss(x, y, beta):
    predictions = np.dot(x, beta)
    return np.mean((y - predictions) ** 2)

def steplength(x, y, beta, lr, plus=1.1, minus=0.5, iterations=250):

    y_hat = prediction(x, beta)  # Initial prediction
    l_old = mean_square_loss(x, y, beta)  # Initial loss

    for i in range(iterations):
        # Calculate the gradient
        grad_beta = -2 * np.dot(x.T, (y - y_hat)) / len(y)

        # Update beta
        beta += lr * grad_beta

        # Calculate new prediction and loss
        y_hat = prediction(x, beta)
        l = mean_square_loss(x, y, beta)

        # Adjust learning rate based on loss
        if l < l_old:
            lr *= plus
        else:
            lr *= minus
            return lr  # Return early if the loss increases

        # Update the previous loss
        l_old = l

    return lr
```

2. LOGISTIC REGRESSION

a.)

In [71]:
```
class Optimization:
    def __init__(self,x,y):
        self.x = x  # Variables
        self.y = y

    def mean_square_loss(self, theta):
        predictions = np.dot(self.x,theta)
        return np.mean((self.y - predictions) ** 2)

    def mean_square_gradient(self, theta):
        predictions = np.dot(self.x, theta)
        gradient_b = -2 * np.dot(self.x.T, (self.y - predictions)) / len(self.y)
        return gradient_b

    def mean_square_hessian(self, theta):
        return 2 * np.dot(self.x.T, self.x) / len(self.x)

    def newtons_method(self,theta,lr=0.01, epochs=50):

      #  theta = np.zeros(x.shape[1])  # Initial values for β

        for epoch in range(epochs):
            # Compute gradients
            gradient_b = self.mean_square_gradient(theta)

            # Compute Hessian matrix
            H_b = self.mean_square_hessian(theta)

            # Update parameters using Newton's method
            inv_hessian = np.linalg.inv(H_b)
            theta -= lr * np.dot(inv_hessian, gradient_b)

            # Compute Mean Square Loss
           # loss = self.mean_square_loss(theta)
            return(theta)

            #print(f"Epoch {epoch + 1}, Mean Square Loss: {loss}")

if __name__ == "__main__":

    x = np.random.rand(100, 3)
    y = np.random.rand(100)

    model = Optimization(x,y)

    theta = np.zeros(x.shape[1])
    optimal_theta = model.newtons_method(theta)
    print("Optimal theta:", optimal_theta)
```

```
Optimal theta: [0.00377951 0.00320711 0.00330593]
```

In [ ]:

In [ ]:

In [288…
```
class Loss:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def mean_square_loss(self, theta):
        predictions = np.dot(self.x, theta)
        return np.mean((self.y - predictions) ** 2)

    def mean_square_gradient(self, theta):
        predictions = np.dot(self.x, theta)
        gradient = -2 * np.dot(self.x.T, (self.y - predictions)) / len(self.y)
        return gradient

    def cross_entropy_loss(self, theta):
        predictions = self.sigmoid(np.dot(self.x, theta))
        return -np.mean(self.y * np.log(predictions) + (1 - self.y) * np.log(1 - predictions))

    def cross_entropy_gradient(self, theta):
        predictions = self.sigmoid(np.dot(self.x, theta))
        gradient = np.dot(self.x.T, (predictions - self.y)) / len(self.y)
        return gradient

    @staticmethod
```

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


class Optimization:
    def __init__(self, x, y):
        self.x = x  # Variables
        self.y = y
        self.loss_obj = Loss(x, y)

    def mean_square_hessian(self):
        return 2 * np.dot(self.x.T, self.x) / len(self.x)

    def newtons_method(self, theta, lr=0.01, epochs=30, loss_type="mse"):

        for epoch in range(epochs):
            if loss_type == "mse":
                gradient = self.loss_obj.mean_square_gradient(theta)
                hessian = self.mean_square_hessian()
            elif loss_type == "cross_entropy":
                gradient = self.loss_obj.cross_entropy_gradient(theta)
                hessian = self.mean_square_hessian()

            inv_hessian = np.linalg.inv(hessian)
            theta -= lr * np.dot(inv_hessian, gradient)

            if loss_type == "mse":
                loss = self.loss_obj.mean_square_loss(theta)
            elif loss_type == "cross_entropy":
                loss = self.loss_obj.cross_entropy_loss(theta)
            #print(f"Epoch {epoch + 1}, Loss ({loss_type}): {loss}")

        return theta

if __name__ == "__main__":
    x = np.random.rand(100, 3)
    y = np.random.rand(100)

    model = Optimization(x, y)
    # Initial theta (parameters)
    theta = np.zeros(x.shape[1])

    print("Mean Square Loss:")
    optimal_theta_mse = model.newtons_method(theta, loss_type="mse")
    print("Optimal theta for MSE after 30 iterations:", optimal_theta_mse)

    # Run Newton's Method for Cross Entropy Loss

    y_binary = np.random.randint(0, 2, size=100)
    model_ce = Optimization(x, y_binary)

    print("\Cross Entropy Loss:")
    optimal_theta_ce = model_ce.newtons_method(theta, loss_type="cross_entropy")
    print("Optimal theta for Cross Entropy after 30 iterations:", optimal_theta_ce)
```

```
Mean Square Loss:
Optimal theta for MSE after 30 iterations: [0.0598615  0.08457299 0.08766653]
\Cross Entropy Loss:
Optimal theta for Cross Entropy after 30 iterations: [0.07106551 0.07236868 0.09858609]
```

b.) Mean Squared Loss and Newton's Method

In [180…
```
df = pd.read_csv('regression.csv')
```

Data Analysis

In [8]:
```
df.isna().sum()
```

Out[8]:
```
X1     0
X2     0
X3     0
X4     0
X5     0
X6     0
X7     0
X8     0
X9     0
X10    0
X11    0
Y      0
dtype: int64
```

No missing values

In [53]:
```
### Dividing data into train and test sets

# from sklearn.model_selection import train_test_split
# Xdata = df[['X1','X2','X3','X4','X5','X6','X7','X8','X9','X10','X11']]
# Ydata = df['Y']
# Xdata = (Xdata - Xdata.mean())/Xdata.std()

# x_train, x_test, y_train, y_test =train_test_split(Xdata,Ydata,train_size=0.8, test_size=0.2)
```

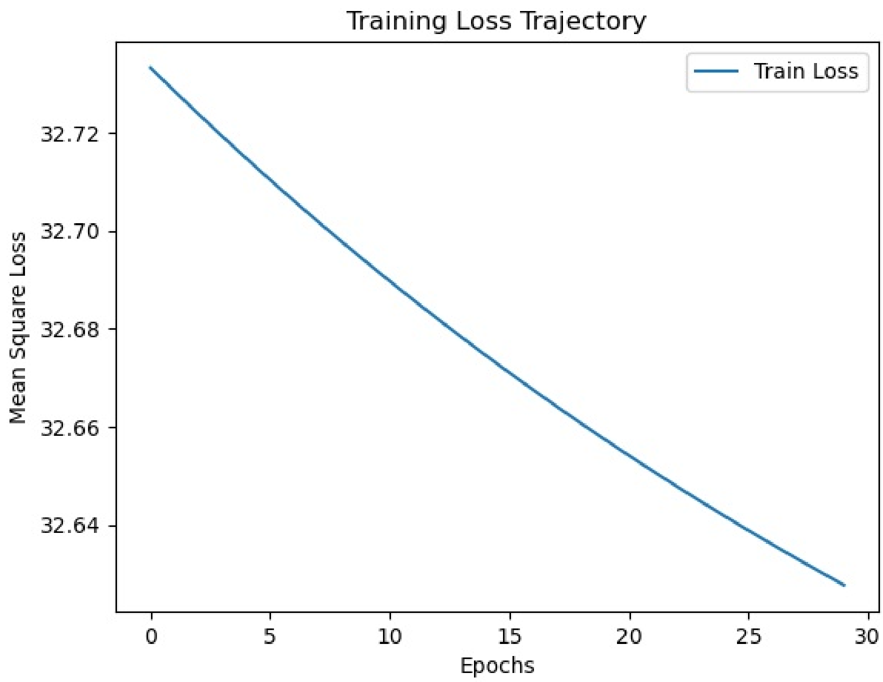
In [ ]:

Task With Classes

In [190…
```
def train_test_split(data):
    # Calculate the split index for 80% of the data
    split_idx = int(len(data) * 0.8)

    # Split the data into training and testing sets
    train_data = data[:split_idx]   # First 80% for training
    test_data = data[split_idx:]    # Last 20% for testing

    return train_data, test_data

train_data, test_data = train_test_split(df)

Xtrain = train_data[['X1','X2','X3','X4','X5','X6','X7','X8','X9','X10','X11']]
Ytrain = train_data['Y']
Xtrainn = (Xtrain - Xtrain.mean())/Xtrain.std()


Xtests = test_data[['X1','X2','X3','X4','X5','X6','X7','X8','X9','X10','X11']]
Ytests = test_data['Y']
Xtestss = (Xtests - Xtests.mean())/Xtests.std()
```

In [191…
```
class Loss:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def mean_square_loss(self, theta):
        predictions = np.dot(self.x, theta)
        return np.mean((self.y - predictions) ** 2)

    def mean_square_gradient(self, theta):
        predictions = np.dot(self.x, theta)
        gradient = -2 * np.dot(self.x.T, (self.y - predictions)) / len(self.y)
        return gradient

    def cross_entropy_loss(self, theta):
        predictions = self.sigmoid(np.dot(self.x, theta))
        return -np.mean(self.y * np.log(predictions) + (1 - self.y) * np.log(1 - predictions))

    def cross_entropy_gradient(self, theta):
        predictions = self.sigmoid(np.dot(self.x, theta))
        gradient = np.dot(self.x.T, (predictions - self.y)) / len(self.y)
        return gradient


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


class Optimization:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.loss_obj = Loss(x, y)

    def mean_square_hessian(self):
        return 2 * np.dot(self.x.T, self.x) / len(self.x)

    def newtons_method(self, theta, lr=0.01, epochs=50, loss_trajectory=None):

        for epoch in range(epochs):
            # Compute gradient and Hessian for Mean Square Loss
            gradient = self.loss_obj.mean_square_gradient(theta)
            hessian = self.mean_square_hessian()

            inv_hessian = np.linalg.inv(hessian)
```

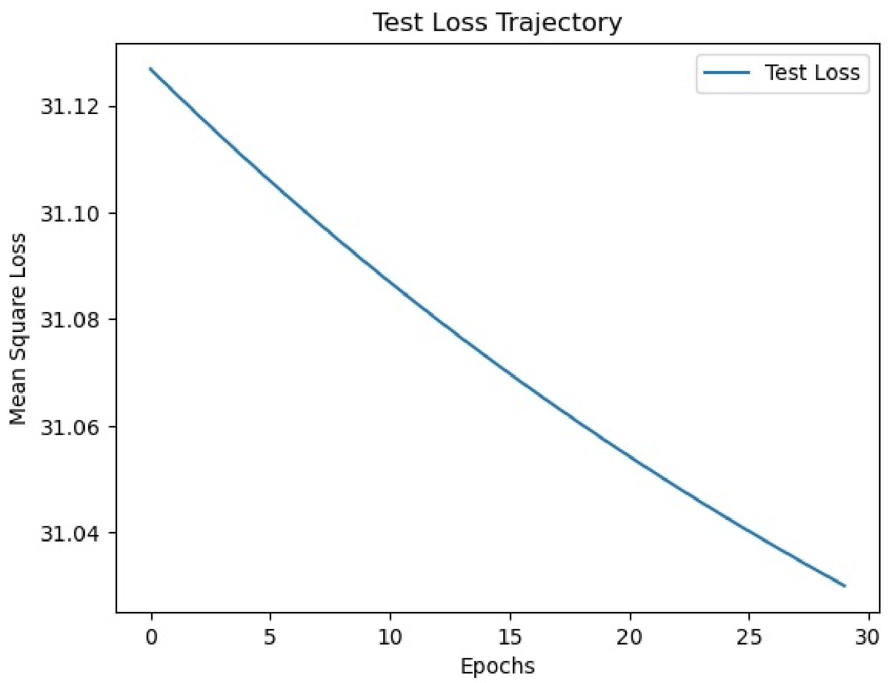
```
            theta -= lr * np.dot(inv_hessian, gradient)

            # Calculate loss for monitoring
            loss = self.loss_obj.mean_square_loss(theta)
            loss_trajectory.append(loss)
           # print(f"Epoch {epoch + 1}, Mean Square Loss: {loss}")

        return theta


class LinearRegression:
    def __init__(self):
        self.theta = None  # Model parameters (coefficients)
        self.train_loss_trajectory = []

    def fit(self, x, y, lr=0.01, epochs=50):
        self.theta = np.zeros(x.shape[1])  # Initialize coefficients (theta)

        # Use the Optimization class to minimize Mean Square Loss
        optimizer = Optimization(x, y)
        self.theta = optimizer.newtons_method(self.theta, lr=lr, epochs=epochs, loss_trajectory=self.train_loss_
        return self.theta

    def predict(self, x):
        return np.dot(x, self.theta)


if __name__ == "__main__":

    model = LinearRegression()
    model.fit(Xtrainn, Ytrain, lr=0.01, epochs=30)


    # Plot training loss trajectory
    plt.plot(model.train_loss_trajectory, label='Train Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Mean Square Loss')
    plt.legend()
    plt.title("Training Loss Trajectory")
    plt.show()
```



In [ ]:

In [192…

```
model1 = LinearRegression()
model1.fit(Xtestss, Ytests, lr=0.01, epochs=30)

#predictions = model.predict(Xtests)

# Plot training loss trajectory
plt.plot(model1.train_loss_trajectory, label='Test Loss')
plt.xlabel('Epochs')
plt.ylabel('Mean Square Loss')
plt.legend()
plt.title("Test Loss Trajectory")
plt.show()
```



It is well known that deep neural networks do not function if the model parameters are initialized to zero. Why is it so? Does this issue also arise while optimizing the loss function for Linear o Logistic Regression? Expl inIn a neural network, each neuron has its own weights and biases that connect it to the neurons in the following layer. When training the network, the objective is to learn unique parameters for each neuron to capture various features of the data. However, if all weights are initialized to zero, the gradients calculated during backpropagation will be identical for each weight. This means that there wold be No Diversity in Feature Learning: Since all weights start with the same value and receive identical updates, each neuron in a given layer will compute the same output as others. As a result, the network fails to learn diverse features, as all neurons in a layer will effectively be "mirrors" of each other. Learning Stagnation: With all neurons in a layer contributing the same output, there is no incremental learning. The network gets stuck in a situation where it cannot differentiate between different patterns or features, causing training to fail. No, this issue does not arise in linear or logistic regression because of its: Linear Structure Convex Optimization: The loss functions for linear regression (mean squared error) and logistic regression (cross-entropy loss) are convex. This means there is only one global minimum, and gradient-based optimization techniques can reach this minimum regardless of whether the weights start at zero or a small random value.

In [ ]:
```
# def mean_square_loss(x, y, b, c):
#     # Predictions
#     predictions = np.dot(x, b) + c
#     # Mean Square Loss
#     return np.mean((y - predictions) ** 2)

# def mean_square_gradient(x, y, b, c):
#     # Predictions
#     predictions = np.dot(x, b) + c
#     # Gradient w.r.t. β
#     gradient_b = -2 * np.dot(x.T, (y - predictions)) / len(y)
#     # Gradient w.r.t. intercept c
#     gradient_c = -2 * np.sum(y - predictions) / len(y)
#     return gradient_b, gradient_c

# def mean_square_hessian(x):
#     # 2nd order derivative
#     return 2 * np.dot(x.T, x) / len(x)

# def newton_method(x, y, lr=0.01, epochs=50, c=-0.5):

#     # Initialize parameters
#     b = np.zeros(x.shape[1]) # Initial values for β
#     train_loss_trajectory = []

#     for epoch in range(epochs):
#         # Step 1: Compute gradients
#         gradient_b, gradient_c = mean_square_gradient(x, y, b, c)

#         # Step 2: Compute Hessian matrix
#         H_b = mean_square_hessian(x)

#         # Step 3: Update parameters
#         inv_hessian = np.linalg.inv(H_b)
#         b -= lr * np.dot(inv_hessian, gradient_b)
#         c -= lr * gradient_c

#         # Compute Mean Square Loss
#         ell = mean_square_loss(x, y, b, c)
#         train_loss_trajectory.append(ell)

#         #print(f"Epoch {epoch+1}, Mean Square Loss: {ell}")

#     return b, c, ell
```

```
# # def predict(x,b,c):
# #     return np.dot(np.dot(x, b) + c)

# if __name__ == "__main__":

#     x = np.array(x_train)  # Feature matrix
#     y = np.array(y_train)  # Labels

#     b, c, ell = newton_method(x, y)
#     print(f"Final Results: β = {b}")
#     print(f"Final Results: c = {c}")
#     print(f"Final Mean Square Loss: {ell}")

#     #predictions = predict(x_train,b,c)
#     plt.plot(train_loss_trajectory, label='Train Loss')
#    # plt.plot(x_train, predictions, color='red', label='Predictions')
```

c.) Cross Entropy Loss and Newton's Method

In [211…

```
df1 = pd.read_csv('logistic.csv')
df1.head()
```

Out[211…

	Y	X1	X2	X3	X4	X5	X6	X7	X8	X9	...	X21	X22	X23	X24	X25	X26	
0	M	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	0.2419	...	25.38	17.33	184.60	2019.0	0.1622	0.6656	0.7
1	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	0.1812	...	24.99	23.41	158.80	1956.0	0.1238	0.1866	0.2
2	M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	0.2069	...	23.57	25.53	152.50	1709.0	0.1444	0.4245	0.4
3	M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	0.2597	...	14.91	26.50	98.87	567.7	0.2098	0.8663	0.6
4	M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	0.1809	...	22.54	16.67	152.20	1575.0	0.1374	0.2050	0.4

5 rows × 31 columns

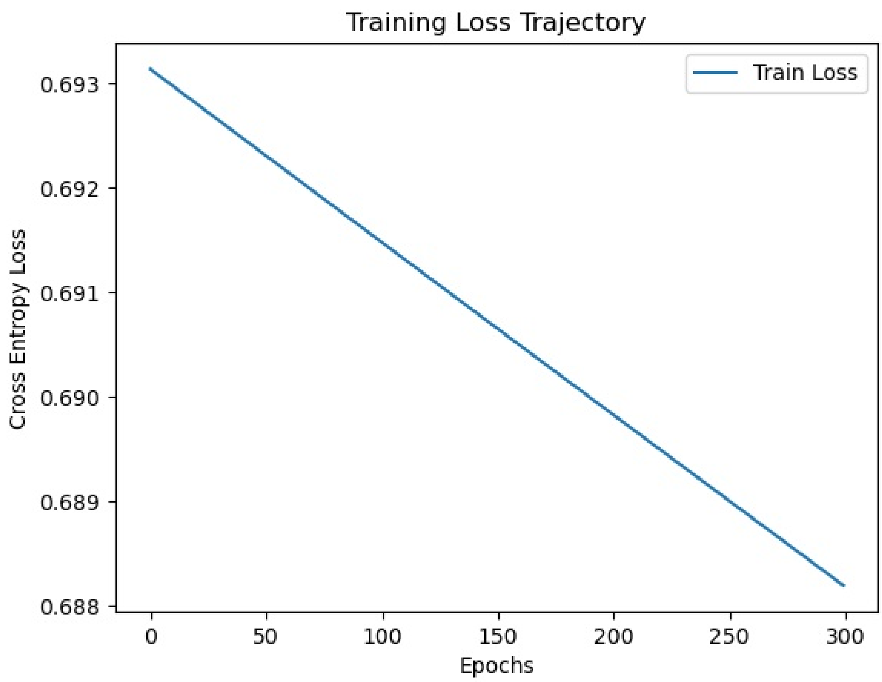
In [260…

```
#One hot encoding
df1['Y'] = df1['Y'].map({'M':'0', 'B':'1'})
df1['Y'].astype(float)

train_data, test_data = train_test_split(df1)

Xtrain = train_data.drop('Y', axis=1)
Ytrain = train_data['Y'].astype(float)
Xtrainn = (Xtrain - Xtrain.mean())/Xtrain.std()


Xtests = test_data.drop('Y', axis=1)
Ytests = test_data['Y'].astype(float)
Xtestss = (Xtests - Xtests.mean())/Xtests.std()
```

In [270…

```
class Loss:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def mean_square_loss(self, theta):
        predictions = np.dot(self.x, theta)
        return np.mean((self.y - predictions) ** 2)

    def mean_square_gradient(self, theta):
        predictions = np.dot(self.x, theta)
        gradient = -2 * np.dot(self.x.T, (self.y - predictions)) / len(self.y)
        return gradient

    def cross_entropy_loss(self, theta):
        predictions = self.sigmoid(np.dot(self.x, theta))
        return -np.mean(self.y * np.log(predictions) + (1 - self.y) * np.log(1 - predictions))

    def cross_entropy_gradient(self, theta):
        predictions = self.sigmoid(np.dot(self.x, theta))
        gradient = np.dot(self.x.T, (predictions - self.y)) / len(self.y)
        return gradient


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


class Optimization:
```

```
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.loss_obj = Loss(x, y)

    def mean_square_hessian(self):
        return 2 * np.dot(self.x.T, self.x) / len(self.x)

    def cross_entropy_hessian(self,theta):
        p = self.sigmoid(np.dot(self.x, theta))
        W = np.diag((p*(1-p)))

        hess = np.dot((np.dot(self.x.T,W)),self.x)

        return hess

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


    def newtons_method(self, theta, lr=0.01, epochs=50, loss_trajectory=None):

        for epoch in range(epochs):
            # Compute gradient and Hessian for Mean Square Loss
            gradient = self.loss_obj.cross_entropy_gradient(theta)
            hessian = self.cross_entropy_hessian(theta)

            inv_hessian = np.linalg.inv(hessian)
            theta -= lr * np.dot(inv_hessian, gradient)

            # Calculate loss for monitoring
            loss = self.loss_obj.cross_entropy_loss(theta)
            loss_trajectory.append(loss)
            #print(f"Epoch {epoch + 1}, Mean Square Loss: {loss}")

        return theta


class LinearRegression:
    def __init__(self):
        self.theta = None  # Model parameters (coefficients)
        self.train_loss_trajectory = []

    def fit(self, x, y, lr=0.01, epochs=200):
        self.theta = np.zeros(x.shape[1])  # Initialize coefficients (theta)

        # Use the Optimization class to minimize Mean Square Loss
        optimizer = Optimization(x, y)
        self.theta = optimizer.newtons_method(self.theta, lr=lr, epochs=epochs, loss_trajectory=self.train_loss_
        return self.theta

    def predict(self, x):
        return np.dot(x, self.theta)

    def predict2(self, x):
        z = np.dot(x, self.theta)
        return 1 / (1 + np.exp(-z))


if __name__ == "__main__":

    model2 = LinearRegression()
    model2.fit(Xtrainn, Ytrain, lr=0.01, epochs=300)

    # Make predictions on test set
    #predictions = model2.predict2(Xtests)

    # Plot training loss trajectory
    plt.plot(model2.train_loss_trajectory, label='Train Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Cross Entropy Loss')
    plt.legend()
    plt.title("Training Loss Trajectory")
    plt.show()
```



In [276…

```
    # Making predictions
predictions = model2.predict2(Xtests)

    # Plotting the testing loss
plt.plot(model3.train_loss_trajectory, label='Test Loss')
plt.xlabel('Epochs')
plt.ylabel('Cross Entropy Loss')
plt.legend()
plt.title("Test Loss Trajectory")
plt.show()
```

In [ ]:

In [283…

```
from sklearn.metrics import classification_report

threshold = 0.5
binary_predictions = (predictions >= threshold).astype(int)
print("Logistic Regression Classification Report:")
print(classification_report(Ytests, binary_predictions))
```

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

         0.0       0.00      0.00      0.00        26
         1.0       0.77      1.00      0.87        88

    accuracy                           0.77       114
   macro avg       0.39      0.50      0.44       114
weighted avg       0.60      0.77      0.67       114
```

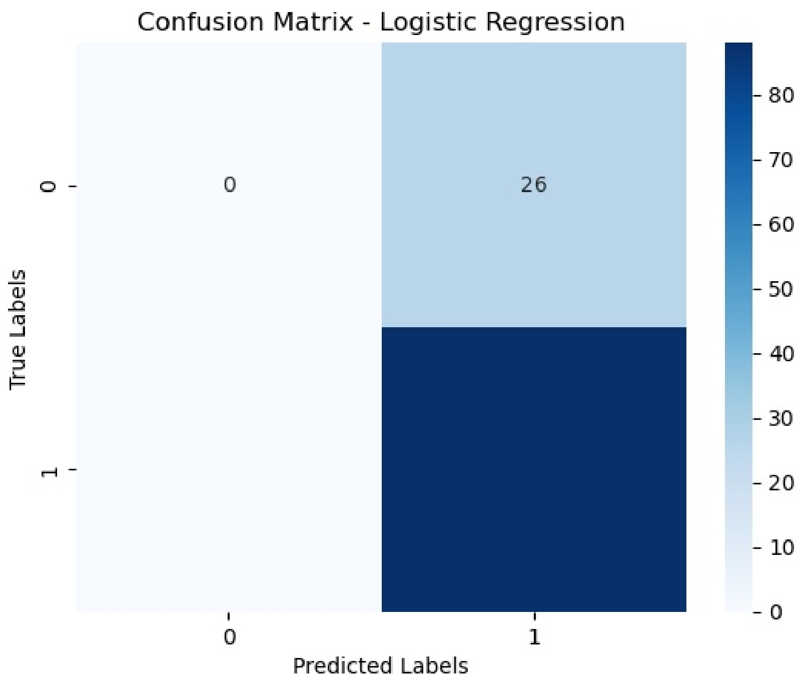
```
C:\Users\tegbe\anaconda3\Lib\site-packages\sklearn\metrics\_classification.py:1344: UndefinedMetricWarning: Prec
ision and F-score are ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division`
parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
C:\Users\tegbe\anaconda3\Lib\site-packages\sklearn\metrics\_classification.py:1344: UndefinedMetricWarning: Prec
ision and F-score are ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division`
parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
C:\Users\tegbe\anaconda3\Lib\site-packages\sklearn\metrics\_classification.py:1344: UndefinedMetricWarning: Prec
ision and F-score are ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division`
parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

Suppose model A and model B both have the same accuracy, but model B has a higher F-score. Which model would be more suited?

Given that both models have the same accuracy but Model B has a higher F-score, Model B would be more suited for scenarios where positive class identification is crucial. It indicates that Model B has better predictive performance in terms of both precision and recall compared to Model A, making it the preferable choice in many practical applications, especially in contexts where false negatives are costly.

In [284…

```python
cm_lr = confusion_matrix(Ytests, binary_predictions)

sns.heatmap(cm_lr, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('Confusion Matrix - Logistic Regression')
plt.show()
```



In [ ]:

In [145…

```python
# def log_likelihood(x, y, beta):
#     z = np.dot(x, beta)
#     log = np.sum( y*z - np.log(1 + np.exp(z)) )
#     return log

# def gradient_ascent(X, h, y):
#     return np.dot(X.T, y - h)

# def logistic_function(X, beta):
#     z = x_train.dot(beta)
#     return 1 / (1 + np.exp(-z))

# def sigmoid(x):
#     return 1/(1+np.exp(-x))
```

```
# def newton(beta0, y, X, lr):

#     p = np.array(sigmoid(X.dot(beta0[:,0]))).T
#     W = np.diag((p*(1-p)))


#     hess = np.dot((np.dot(X.T,W)),X)
#     grad = (np.transpose(X)).dot(y-p)


#     s =lr*(np.dot(np.linalg.inv(hess), grad))
#     beta = beta0 + s

#     return beta

# logloss = lambda y,ypred: np.mean((y*np.log(ypred)+(1-y)*np.log(1-ypred)))
# cost = lambda y,ypred: np.mean((y - ypred)**2)
```

In [ ]:

In [ ]:

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js