CSE-6363-007 Programming Assignment 1 (Spring 2024)

Problem 1: Implementing Linear regression using gradient descent.

In this part of assignment, I have implemented code for linear_regression.py file. This python code takes linear.csv file as an input, in which first 24 rows are assigned as a training data, and remaining 6 columns represent the test data. Once we run following command, it will predict salary as output.

```
python3 linear_regression.py --data linear.csv
```

Implementation details:

1. **Importing libraries:**

```
import numpy as np
import argparse
# Import the `pyplot` module from the `matplotlib` library to plot the graph for visualization
import matplotlib.pyplot as plt
```

- NumPy (numpy): Used for numerical operations and handling arrays.
- Argparse (argparse): Used for parsing command-line arguments.
- Matplotlib (matplotlib.pyplot): Used for plotting graphs and visualizations.

2. **Linear regression- class definition:**

```
class LinearRegressionGradientDescent:
    def __init__(self, learning_rate=0.01, n_iterations=1000):
        self.learning_rate = learning_rate
        self.n_iterations = n_iterations
        self.weights = None
        self.bias = None
```

- Purpose: Defining a class for linear regression using gradient decent. '__init__' initializes the class with default or user-specified learning rate and number of iterations.
- Logic: model parameters such as weights and bias, and hyperparameters such as learning rate and number of iterations can be initialized here

3. **Fit method:**

```
def fit(self, X, y):
    n_samples, n_features = X.shape
    self.weights = np.zeros(n_features)
    self.bias = 0

    for _ in range(self.n_iterations):
        linear_model = np.dot(X, self.weights) + self.bias
        cost = (1 / n_samples) * np.sum((linear_model - y) ** 2)

        # Compute gradients
        weights_gradient = (2 / n_samples) * np.dot(X.T, (linear_model - y))
        bias_gradient = (2 / n_samples) * np.sum(linear_model - y)

        # Update parameters
        self.weights = self.weights- self.learning_rate * weights_gradient
        self.bias = self.bias - self.learning_rate * bias_gradient
```

- **Purpose:** The fit method trains the linear regression model by adjusting weights and bias using gradient descent.
- **Logic:**
  - Initialize weights and bias.
  - Loop through the specified number of iterations.
  - Compute the linear model predictions and the mean squared error cost.
  - Compute gradients for weights and bias.
  - Update weights and bias using the computed gradients and the learning rate.

4. Predict method:

```
def predict(self, X):
    return np.dot(X, self.weights) + self.bias
```

- Purpose: The predict method makes predictions using the trained linear regression model for a given input dataset.
- Logic: Computes the dot product of the input features with the weights and subsequently adds the bias to yield the predicted values.

5. Command line Argument parsing

```
if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Linear Regression with Gradient Descent")
    parser.add_argument("--data", type=str, help="Path to data file (CSV format)")
    parser.add_argument("--learning_rate", type=float, default=0.01, help="Learning rate for gradient descent")
    parser.add_argument("--n_iterations", type=int, default=1000, help="Number of iterations for gradient descent")
    args = parser.parse_args()
```

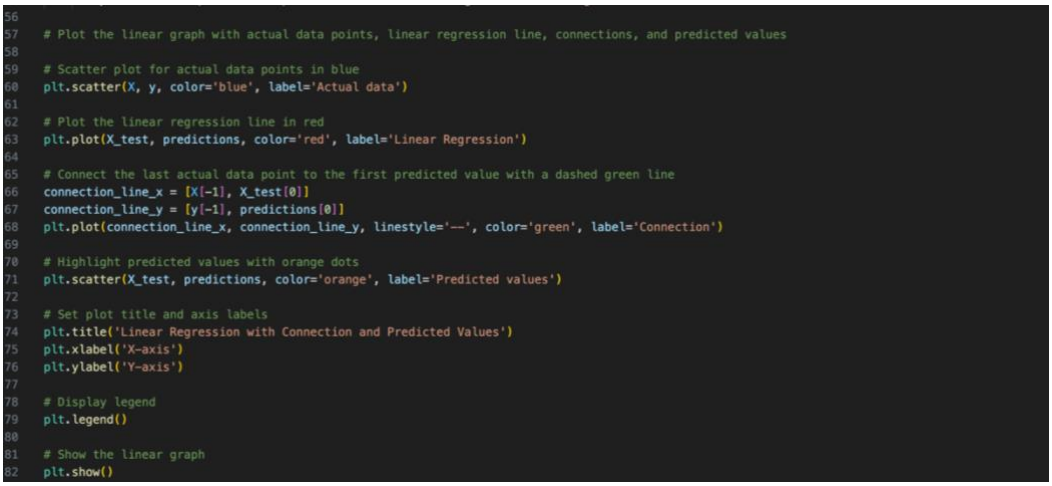
- **Purpose:** Enables the execution of the model from the command line, offering the flexibility to specify parameters such as the dataset path, learning rate, and the number of iterations.
- **Logic:**
  - **Main Execution Block:** Checks if the script is being run directly.
  - **Argument Parsing:** Uses argparse to parse command-line arguments, including data file path, learning rate, and number of iterations.

6. Data loading and model training

```
39
40      if args.data:
41          # Load data
42          data = np.genfromtxt(args.data, delimiter=',')
43          X = data[1:24, :-1]
44          y = data[1:24, -1]
45          X_test = data[25:, :-1]
46          # Initialize and train the model
47          model = LinearRegressionGradientDescent(learning_rate=args.learning_rate, n_iterations=args.n_iterations)
48          model.fit(X, y)
49
50          # Make predictions
51          predictions = model.predict(X_test)
52
53          print("Predictions:", predictions)
54      else:
55          print("Please provide the path to the data file using the '--data' argument.")
56
```

- **Purpose:** To prepare the machine learning model for learning patterns from the provided dataset.
- **Logic:**
  - Data Loading: Loads data from the specified CSV file using NumPy.
  - Model Initialization and Training: Initializes the linear regression model and trains it using the loaded data.
  - Prediction: Makes predictions on a separate set of data.
  - Print Predictions: Prints the predicted values to the console.

7. Plotting the linear graph for visualization



```
# Plot the linear graph with actual data points, linear regression line, connections, and predicted values

# Scatter plot for actual data points in blue
plt.scatter(X, y, color='blue', label='Actual data')

# Plot the linear regression line in red
plt.plot(X_test, predictions, color='red', label='Linear Regression')

# Connect the last actual data point to the first predicted value with a dashed green line
connection_line_x = [X[-1], X_test[0]]
connection_line_y = [y[-1], predictions[0]]
plt.plot(connection_line_x, connection_line_y, linestyle='--', color='green', label='Connection')

# Highlight predicted values with orange dots
plt.scatter(X_test, predictions, color='orange', label='Predicted values')

# Set plot title and axis labels
plt.title('Linear Regression with Connection and Predicted Values')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')

# Display legend
plt.legend()

# Show the linear graph
plt.show()
```

- **Purpose :** To visually represent the results of the trained linear regression model. This section involves creating a graphical representation that helps in understanding the relationship between the input features and the target variable.
- **Logic:**
  - Plotting the Linear Graph: Uses Matplotlib to create a scatter plot for actual data points, a line plot for the linear regression line, a dashed line for connecting the last data point to the first predicted value, and orange dots to highlight predicted values.
  - Graph Customization: Sets plot title, axis labels, and legend.
  - Display Plot: Shows the linear graph.

Hyperparameters:

Hyperparameters are parameters that are not learned from the data but are set prior to the training process. Here are the hyperparameters and their explanations:

- Learning Rate (--learning_rate):
  - Purpose: Controls the step size during the optimization process (gradient descent).
  - Default Value: 0.01
  - Usage: Can be specified through the command line. Determines how much the model's parameters are adjusted during each iteration of training.
- Number of Iterations (--n_iterations):
  - Purpose: Determines how many times the model will update its parameters during training.
  - Default Value: 1000
  - Usage: Can be specified through the command line. Represents the number of times the model will iterate over the entire training dataset.

Predictions:

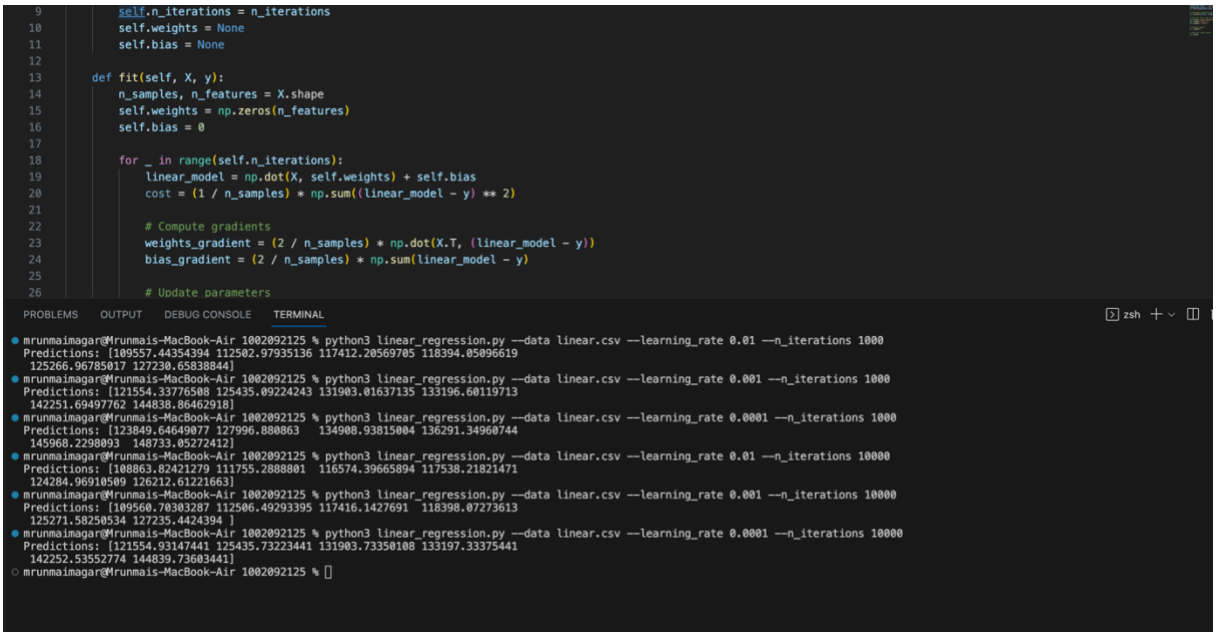
1. Prediction 1(Salary predicted by model of first data point): **109557.44354394**
2. Prediction 2(Salary predicted by model of second data point): **112502.97935136**
3. Prediction 3(Salary predicted by model of third data point): **117412.20569705**
4. Prediction 4(Salary predicted by model of forth data point): **118394.05096619**
5. Prediction 5(Salary predicted by model of fifth data point): **125266.96785017**
6. Prediction 6(Salary predicted by model of sixth data point): **127230.65838844**

Predictions with different sets of hyperparameters:

Learning Rate	Iterations	Final Training Cost	Test Mean Squared Error (MSE)	Prediction for data point 1	Prediction for data point 2	Prediction for data point 3	Prediction for data point 4	Prediction for data point 5	Prediction for data point 6
0.01	1,000	35,116,430	35,350,630	109,557.44	112,502.98	117,412.21	118,394.05	125,266.97	127,230.66
0.001	1,000	399,537,800	2,360,724,000	121,554.34	125,435.09	131,903.02	133,196.60	142,251.69	144,838.86
0.0001	1,000	3,361,286,000	11,166,520,000	123,849.65	127,996.88	134,908.94	136,291.35	145,968.23	148,733.05
0.01	10,000	35,116,250	35,766,740	108,863.82	111,755.29	116,574.40	117,538.22	124,284.97	126,212.61
0.001	10,000	35,116,440	35,338,410	109,560.70	112,506.49	117,416.14	118,398.07	125,271.58	127,235.44
0.0001	10,000	399,264,400	2,362,238,000	121,554.93	125,435.73	131,903.73	133,197.33	142,252.54	144,839.74

In our typical workflow, we conduct multiple experiments, each involving variations in the learning rate and the number of iterations. Throughout these experiments, we monitor and record the cost, specifically the Mean Squared Error (MSE), for both the training and test datasets. This allows us to compare the performance of different model configurations and make informed decisions regarding hyperparameter tuning.

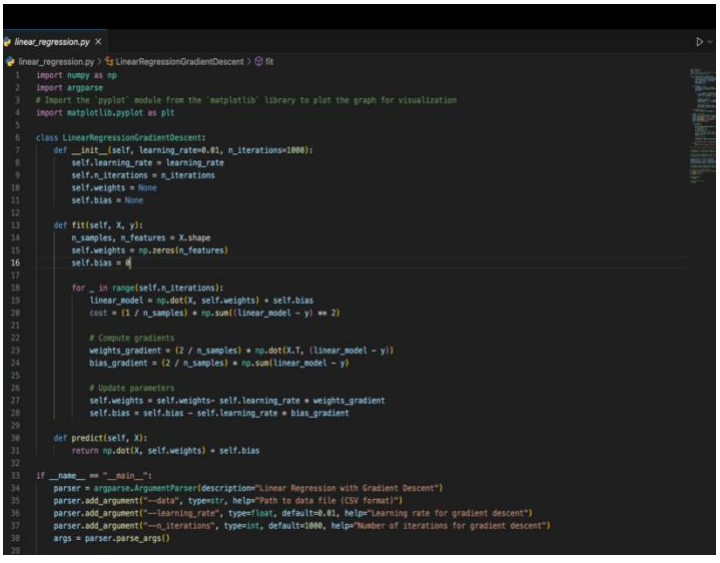
Results: Output screen in VSCode IDE:



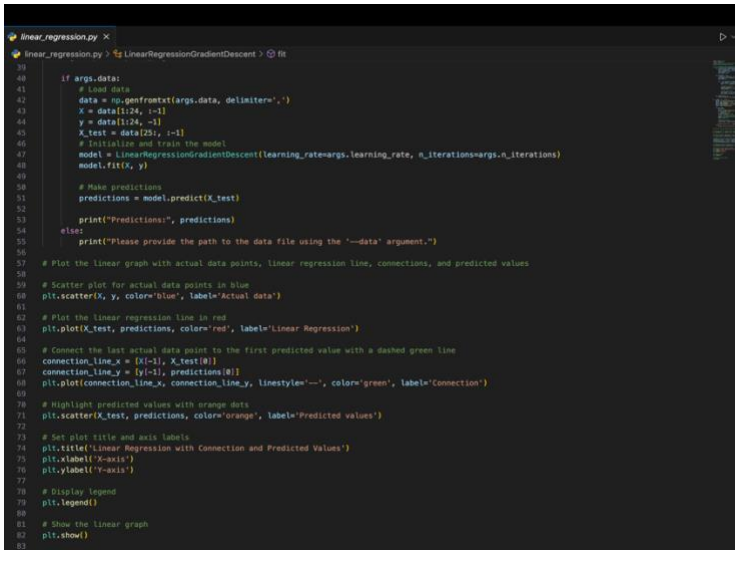
```
        self.n_iterations = n_iterations
        self.weights = None
        self.bias = None

    def fit(self, X, y):
        n_samples, n_features = X.shape
        self.weights = np.zeros(n_features)
        self.bias = 0

        for _ in range(self.n_iterations):
            linear_model = np.dot(X, self.weights) + self.bias
            cost = (1 / n_samples) * np.sum((linear_model - y) ** 2)

            # Compute gradients
            weights_gradient = (2 / n_samples) * np.dot(X.T, (linear_model - y))
            bias_gradient = (2 / n_samples) * np.sum(linear_model - y)

            # Update parameters
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL                                  zsh

mrunmaimagar@Mrunmais-MacBook-Air 1002092125 % python3 linear_regression.py --data linear.csv --learning_rate 0.01 --n_iterations 1000
Predictions: [109557.44354394 112502.97935136 117412.20569705 118394.05096619
 125266.96785017 127230.65838844]
mrunmaimagar@Mrunmais-MacBook-Air 1002092125 % python3 linear_regression.py --data linear.csv --learning_rate 0.001 --n_iterations 1000
Predictions: [121554.33776508 125435.09224243 131903.01637135 133196.60119713
 142251.69497762 144838.86462918]
mrunmaimagar@Mrunmais-MacBook-Air 1002092125 % python3 linear_regression.py --data linear.csv --learning_rate 0.0001 --n_iterations 1000
Predictions: [123849.64649077 127996.880863   134908.93815004 136291.34960744
 145968.2298093  148733.05272412]
mrunmaimagar@Mrunmais-MacBook-Air 1002092125 % python3 linear_regression.py --data linear.csv --learning_rate 0.01 --n_iterations 10000
Predictions: [108863.82421279 111755.2888801  116574.39665894 117538.21821471
 124284.96910509 126212.61221663]
mrunmaimagar@Mrunmais-MacBook-Air 1002092125 % python3 linear_regression.py --data linear.csv --learning_rate 0.001 --n_iterations 10000
Predictions: [109560.70303287 112506.49293395 117416.1427691  118398.07273613
 125271.58250534 127235.4424394 ]
mrunmaimagar@Mrunmais-MacBook-Air 1002092125 % python3 linear_regression.py --data linear.csv --learning_rate 0.0001 --n_iterations 10000
Predictions: [121554.93147441 125435.73223441 131903.73350108 133197.33375441
 142252.53552774 144839.73603441]
mrunmaimagar@Mrunmais-MacBook-Air 1002092125 %
```

Code Snippets:



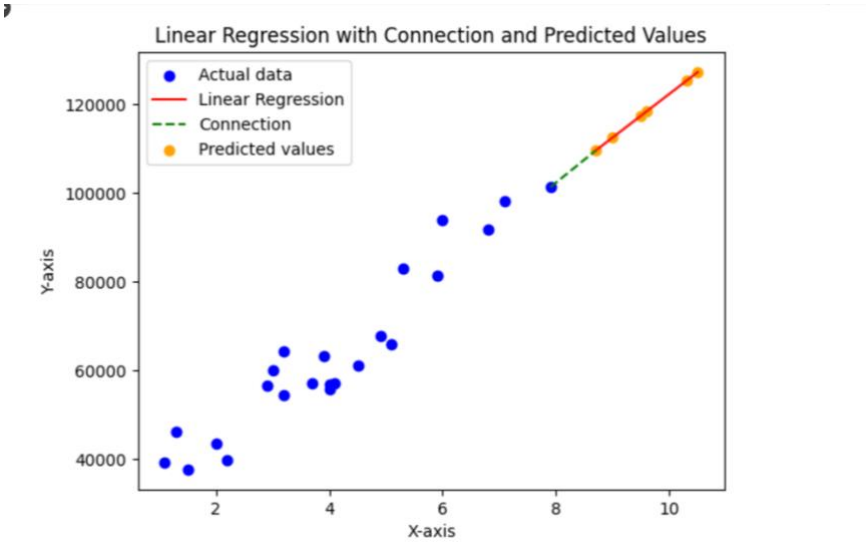
```
linear_regression.py
import numpy as np
import argparse
# Import the `pyplot` module from the `matplotlib` library to plot the graph for visualization
import matplotlib.pyplot as plt

class LinearRegressionGradientDescent:
    def __init__(self, learning_rate=0.01, n_iterations=1000):
        self.learning_rate = learning_rate
        self.n_iterations = n_iterations
        self.weights = None
        self.bias = None

    def fit(self, X, y):
        n_samples, n_features = X.shape
        self.weights = np.zeros(n_features)
        self.bias = 0

        for _ in range(self.n_iterations):
            linear_model = np.dot(X, self.weights) + self.bias
            cost = (1 / n_samples) * np.sum((linear_model - y) ** 2)

            # Compute gradients
            weights_gradient = (2 / n_samples) * np.dot(X.T, (linear_model - y))
            bias_gradient = (2 / n_samples) * np.sum(linear_model - y)

            # Update parameters
            self.weights = self.weights- self.learning_rate * weights_gradient
            self.bias = self.bias - self.learning_rate * bias_gradient

    def predict(self, X):
        return np.dot(X, self.weights) + self.bias

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Linear Regression with Gradient Descent")
    parser.add_argument("--data", type=str, help="Path to data file (CSV format)")
    parser.add_argument("--learning_rate", type=float, default=0.01, help="Learning rate for gradient descent")
    parser.add_argument("--n_iterations", type=int, default=1000, help="Number of iterations for gradient descent")
    args = parser.parse_args()
```



```
linear_regression.py
    if args.data:
        # Load data
        data = np.genfromtxt(args.data, delimiter=',')
        X = data[1:24, :-1]
        y = data[1:24, -1]
        X_test = data[25:, :-1]
        # Initialize and train the model
        model = LinearRegressionGradientDescent(learning_rate=args.learning_rate, n_iterations=args.n_iterations)
        model.fit(X, y)

        # Make predictions
        predictions = model.predict(X_test)

        print("Predictions:", predictions)
    else:
        print("Please provide the path to the data file using the '--data' argument.")

# Plot the linear graph with actual data points, linear regression line, connections, and predicted values

# Scatter plot for actual data points in blue
plt.scatter(X, y, color='blue', label='Actual data')

# Plot the linear regression line in red
plt.plot(X_test, predictions, color='red', label='Linear Regression')

# Connect the last actual data point to the first predicted value with a dashed green line
connection_line_x = [X[-1], X_test[0]]
connection_line_y = [y[-1], predictions[0]]
plt.plot(connection_line_x, connection_line_y, linestyle='--', color='green', label='Connection')

# Highlight predicted values with orange dots
plt.scatter(X_test, predictions, color='orange', label='Predicted values')

# Set plot title and axis labels
plt.title('Linear Regression with Connection and Predicted Values')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')

# Display legend
plt.legend()

# Show the linear graph
plt.show()
```

Graph Plots:

The graph below illustrates that the linear regression model has successfully captured a linear relationship between years of experience and salary. It demonstrates the model's ability to predict salary based on a given number of years of experience, showing accuracy as indicated by the proximity of the predictions (orange dots) to the training data (blue dots) along the regression line (red line).



Depending upon above predictions, here is the interpretation we can get:

A learning rate set at 0.01 appears to be the most effective choice for optimizing this model on the given dataset, resulting in the lowest Mean Squared Error (MSE) values for both the training and test sets. Increasing the number of iterations from 1,000 to 10,000 does not appear to significantly impact the final training cost or test MSE when using a learning rate of 0.01.
However, employing lower learning rates (0.001 and 0.0001) with fewer iterations (1,000) leads to notably high MSE values, indicating that the model has not undergone sufficient updates to learn effectively. Even when the number of iterations is increased for these lower learning rates, the achieved MSE does not reach the levels observed with the higher learning rate of 0.01.

Problem 2: Implement logistic regression using gradient ascent algorithm and newton method.

In this part of assignment, I have implemented code for logistic_regression.py file. This python code takes log_training.csv and log_testing.csv files as inputs, , which are training and testing csv files. Once we run following command, it will predict salary as output.

`python3 logistic_regression.py --data log_training.csv –test_data log_training.csv --method 'gradient' OR 'newton'`

Implementation Details:

1. **Sigmoid Function:**

```
4   def sigmoid(z):
5       return 1 / (1 + np.exp(-z))
6
```

   - Purpose:
     - Computes the sigmoid function, which is the activation function used in logistic regression.
     - It maps any real-valued number into the range (0, 1), suitable for binary classification.
   - Logic:
     - The function takes a numerical input z and calculates the sigmoid of z using the formula 1 / (1 + e^(-z)), where e is the base of the natural logarithm.
     - Returns the sigmoid value.

2. **'Logistic_regression_newton' function:**

```
def logistic_regression_newton(X, y, learning_rate=0.01, n_iterations=1000, tol=1e-6):
    n_samples, n_features = X.shape
    weights = np.zeros(n_features)
    bias = 0

    for _ in range(n_iterations):
        linear_model = np.dot(X, weights) + bias
        y_predicted = sigmoid(linear_model)

        # Compute gradient
        gradient_w = np.dot(X.T, (y - y_predicted)) / n_samples
        gradient_b = np.sum(y - y_predicted) / n_samples

        # Compute Hessian matrix
        W = y_predicted * (1 - y_predicted)
        Hessian_w = -np.dot(X.T, np.dot(np.diag(W), X)) / n_samples

        # Update parameters using Newton's method
        weights -= np.linalg.inv(Hessian_w).dot(gradient_w)
        bias += learning_rate * gradient_b

    linear_model = np.dot(X, weights) + bias
    y_predicted = sigmoid(linear_model)
    y_predicted_cls = [1 if i > 0.5 else 0 for i in y_predicted]
    return np.array(y_predicted_cls)
```

- Purpose: Implements logistic regression using Newton's method for optimization.
- Logic:
  - Initializes weights and bias.
  - Iteratively performs the following steps for a specified number of iterations-
    1. Calculates the linear model.
    2. Computes the gradient of the cost function with respect to weights and bias.
    3. Computes the Hessian matrix.
    4. Updates the weights and bias using Newton's method.
    5. Returns the predicted classes based on the trained model.

3. 'logistic_regression_gradient' function:

```
def logistic_regression_gradient_ascent(X, y, learning_rate=0.001, n_iterations=1000, tol=1e-6):
    n_samples, n_features = X.shape
    weights = np.zeros(n_features)
    bias = 0

    for _ in range(n_iterations):
        linear_model = np.dot(X, weights) + bias
        y_predicted = sigmoid(linear_model)

        # Compute gradient
        gradient_w = np.dot(X.T, (y - y_predicted)) / n_samples
        gradient_b = np.sum(y - y_predicted) / n_samples

        # Update parameters using gradient ascent
        weights += learning_rate * gradient_w
```

a. Purpose: Implements logistic regression using gradient ascent for optimization.
b. Logic:
   i. Initializes weights and bias.
   ii. Iteratively performs the following steps for a specified number of iterations:
   iii. Calculates the linear model.
   iv. Computes the gradient of the cost function with respect to weights and bias.
   v. Updates the weights and bias using gradient ascent.
   vi. Returns the predicted classes based on the trained model.

4. Accuracy function:

```
def calculate_accuracy(y_true, y_pred):
    correct_predictions = np.sum(y_true == y_pred)
    accuracy = correct_predictions / len(y_true)
    return accuracy
```

- Purpose:
  - Accuracy serves as a metric to assess the performance of a classification model, indicating the model's ability to make correct predictions.

        - It is determined by the ratio of accurately predicted instances to the total instances, offering insight into the proportion of accurate predictions made by the model.
    - logic:
        - Accurate Predictions: The quantity of accurate predictions is determined through a comparison between the predicted class labels generated by the model and the actual class labels.
        - Total Predictions: This corresponds to the total count of predictions made by the model.
        - Ratio Calculation: Accuracy is derived by dividing the number of accurate predictions by the total predictions, often represented as a percentage.

Hyperparameters and performance of models on training and test data:

1. **Logistic Regression with Newton's Method:**
    a. **Learning Rate (learning_rate):**
        i. **Definition:** The step size at each iteration when updating the model parameters.
        ii. **Impact:** Affects the convergence speed; a too large learning rate may cause divergence, while a too small one may result in slow convergence.
    b. **Number of Iterations (n_iterations):**
        i. **Definition:** The total number of iterations or updates to the model parameters during the training process.
        ii. **Impact:** Influences how many times the model parameters are adjusted. Insufficient iterations may lead to underfitting, while too many iterations may cause overfitting.
    c. **Tolerance (tol):**
        i. **Definition:** A small positive value used as a stopping criterion. If the change in the model parameters is less than the tolerance, the training process stops.
        ii. **Impact:** Determines when to stop the training process. A lower tolerance may lead to more precise solutions but requires more iterations.

•Performance on Training Data:
The model is trained on the training dataset (X and y). The accuracy on the training set is not explicitly evaluated in the code, but it's crucial to monitor to ensure the model is learning from the data effectively.
•Performance on Test Data:
The model is then tested on a separate test dataset (X_test and y_test). The accuracy is computed, and the results are printed as "Predictions: [array of predicted classes]".

2. **Logistic Regression with Gradient Ascent:**
    a. **Learning Rate (learning_rate):**
        i. **Definition:** Similar to Newton's method, it is the step size when updating the model parameters.
        ii. **Impact:** Affects the convergence speed; the choice of learning rate is crucial to balance between fast convergence and avoiding divergence.
    b. **Number of Iterations (n_iterations):**
        i. **Definition:** The total number of iterations or updates to the model parameters during training.
        ii. **Impact:** Similar to Newton's method, it influences how many times the model parameters are adjusted. Proper tuning is required to achieve a good balance.
    c. **Tolerance (tol):**
        i. **Definition:** A small positive value used as a stopping criterion. If the change in the model parameters is less than the tolerance, the training process stops.
        ii. **Impact:** Similar to Newton's method, it determines when to stop the training process. A lower tolerance may lead to more precise solutions but requires more iterations.

- **Performance on Training Data:**
  The model is trained on the training dataset (X and y).
  Similar to the Newton's method model, the accuracy on the training set is essential for understanding how well the model is learning from the data.
- **Performance on Test Data:**
  The model is tested on a separate test dataset (X_test and y_test).
  The accuracy is computed, and the results are printed as "Predictions: [array of predicted classes]".

Results:

Performance Metrics:

- Model Predictions: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
  (for both Newton's method and Gradient ascent)
- Evaluation Metrics: Accuracy, Precision, Recall, F1 Score

Calculated Metrics:

• **Accuracy**: Reported as 0.9 (90%), signaling a notable proportion of accurate predictions overall.
• **F1 Score**: Registered at 0 (0%), revealing an imbalance between Precision and Recall, primarily due to the absence of correct positive predictions.
• **Precision**: Stands at 0 (0%), indicating the model's inability to accurately predict any positive instances.
• **Recall**: Recorded at 0% (0), signifying the model's failure to correctly identify any true positive instances among all positive instances in the test set.

Visualization – confusion matrix:

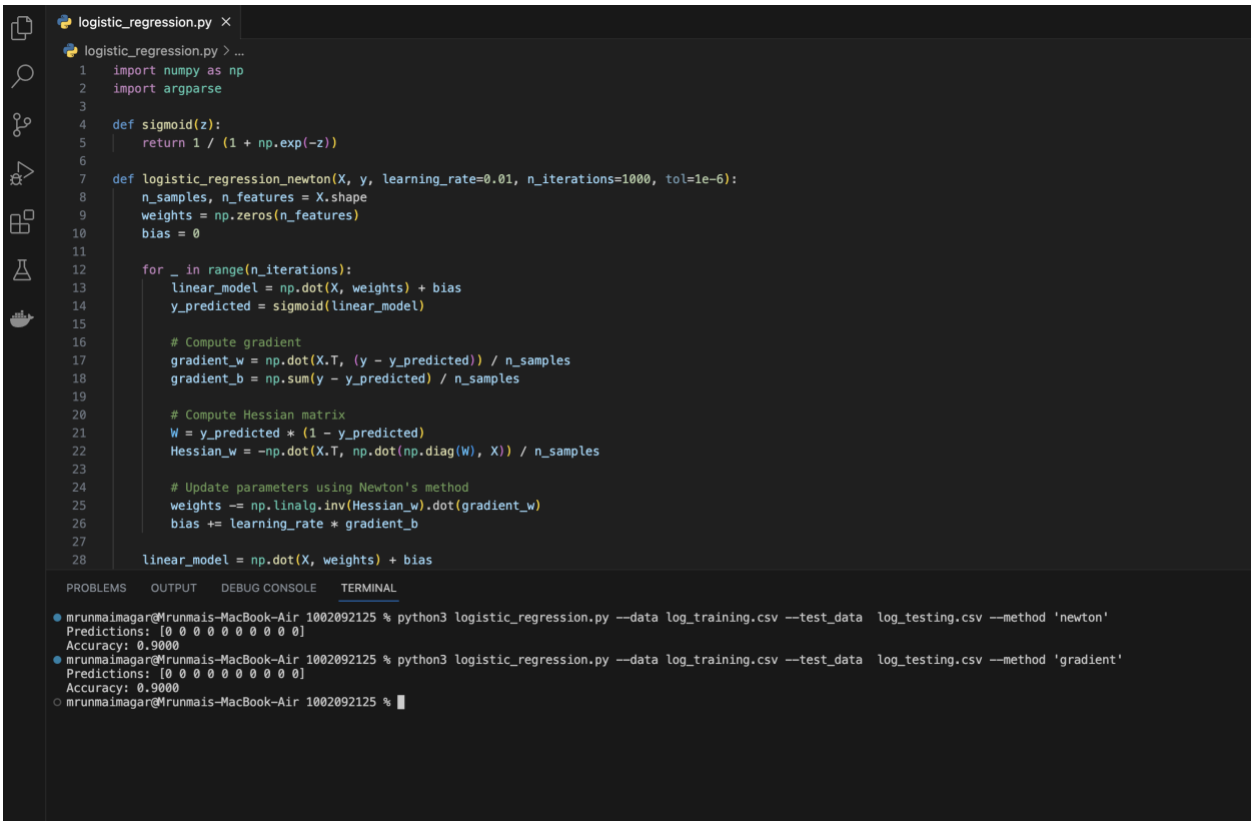
Actual Labels (y_test): Extracted from the log_testing.csv file.
Predicted Labels (y_pred): [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

With the help of above predictions, we can calculate components of confusion matrix such as :

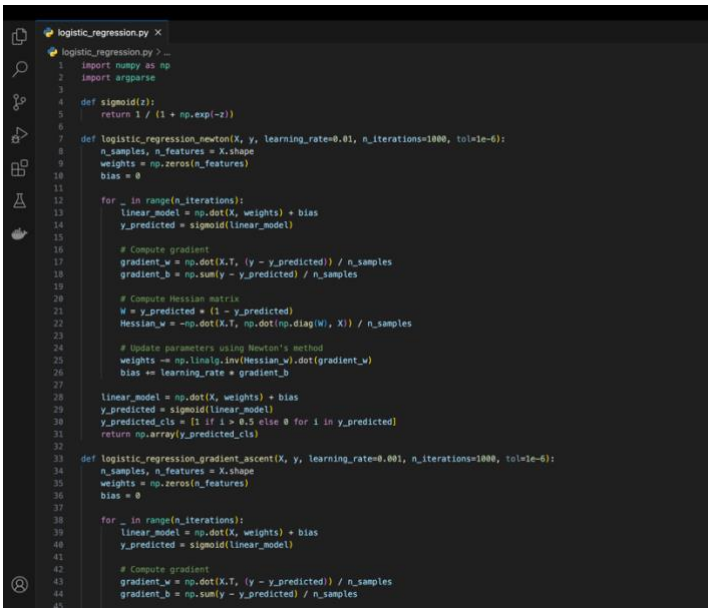
- **True Positive (TP):** 0 (The cases where both the actual and predicted labels are 1.
- **True Negative (TN):** 9 (The cases where the model correctly predicted negative instances.)
- **False Positive (FP):** 0 The cases where the model incorrectly predicts the positive class.)
- **False Negative (FN):** 1 (The cases where the model incorrectly predicted negative instances, but there was one positive instance in the actual data.)

	Predicted Positive	**Predicted Negative**
Actual positive	0 (TP)	1 (TN)
Actual negative	0 (FN)	9 (TN)

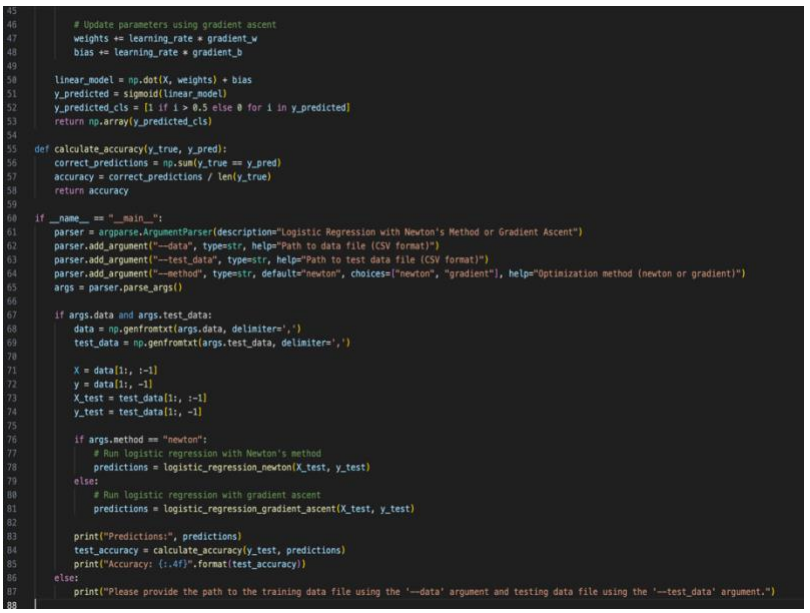
Output in VSCode IDE:



Code Snippets:



```python
import numpy as np
import argparse

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

def logistic_regression_newton(X, y, learning_rate=0.01, n_iterations=1000, tol=1e-6):
    n_samples, n_features = X.shape
    weights = np.zeros(n_features)
    bias = 0

    for _ in range(n_iterations):
        linear_model = np.dot(X, weights) + bias
        y_predicted = sigmoid(linear_model)

        # Compute gradient
        gradient_w = np.dot(X.T, (y - y_predicted)) / n_samples
        gradient_b = np.sum(y - y_predicted) / n_samples

        # Compute Hessian matrix
        W = y_predicted * (1 - y_predicted)
        Hessian_w = -np.dot(X.T, np.dot(np.diag(W), X)) / n_samples

        # Update parameters using Newton's method
        weights -= np.linalg.inv(Hessian_w).dot(gradient_w)
        bias += learning_rate * gradient_b

    linear_model = np.dot(X, weights) + bias
    y_predicted = sigmoid(linear_model)
    y_predicted_cls = [1 if i > 0.5 else 0 for i in y_predicted]
    return np.array(y_predicted_cls)

def logistic_regression_gradient_ascent(X, y, learning_rate=0.001, n_iterations=1000, tol=1e-6):
    n_samples, n_features = X.shape
    weights = np.zeros(n_features)
    bias = 0

    for _ in range(n_iterations):
        linear_model = np.dot(X, weights) + bias
        y_predicted = sigmoid(linear_model)

        # Compute gradient
        gradient_w = np.dot(X.T, (y - y_predicted)) / n_samples
        gradient_b = np.sum(y - y_predicted) / n_samples
```



```python
        # Update parameters using gradient ascent
        weights += learning_rate * gradient_w
        bias += learning_rate * gradient_b

    linear_model = np.dot(X, weights) + bias
    y_predicted = sigmoid(linear_model)
    y_predicted_cls = [1 if i > 0.5 else 0 for i in y_predicted]
    return np.array(y_predicted_cls)

def calculate_accuracy(y_true, y_pred):
    correct_predictions = np.sum(y_true == y_pred)
    accuracy = correct_predictions / len(y_true)
    return accuracy

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Logistic Regression with Newton's Method or Gradient Ascent")
    parser.add_argument("--data", type=str, help="Path to data file (CSV format)")
    parser.add_argument("--test_data", type=str, help="Path to test data file (CSV format)")
    parser.add_argument("--method", type=str, default="newton", choices=["newton", "gradient"], help="Optimization method (newton or gradient)")
    args = parser.parse_args()

    if args.data and args.test_data:
        data = np.genfromtxt(args.data, delimiter=',')
        test_data = np.genfromtxt(args.test_data, delimiter=',')

        X = data[1:, :-1]
        y = data[1:, -1]
        X_test = test_data[1:, :-1]
        y_test = test_data[1:, -1]

        if args.method == "newton":
            # Run logistic regression with Newton's method
            predictions = logistic_regression_newton(X_test, y_test)
        else:
            # Run logistic regression with gradient ascent
            predictions = logistic_regression_gradient_ascent(X_test, y_test)

        print("Predictions:", predictions)
        test_accuracy = calculate_accuracy(y_test, predictions)
        print("Accuracy: {:.4f}".format(test_accuracy))
    else:
        print("Please provide the path to the training data file using the '--data' argument and testing data file using the '--test_data' argument.")
```

Comparison of methods and hyperparameters:

Aspect	Linear Regression	Logistic Regression
Where can be used	Predicting continuous outcomes	Binary or multiclass classification problems
Characteristics	Assumes linear relationship	Uses logistic function for binary outcomes
Strengths	Simple and interpretable, Efficient for predicting continuous outcomes	Robust to noise, Suitable for binary classifications, Provides probability scores
Hyperparameters	Learning Rate: 0.01, Iterations: 1,000	Learning Rate: Varies, Iterations: Varies, Regularization applied
Effect of Hyperparameters	Learning Rate affects convergence speed, Iterations impact fitting and computational cost	Learning Rate/Iterations have a similar impact, Regularization crucial for preventing overfitting and adjusting model complexity
Output Example	Final Training Cost: 35,116,430, Test MSE: 35,350,630, Predictions: 109,557.44, 112,502.98, 117,412.21, 118,394.05, 125,266.97, 127,230.66	Accuracy: 90%, Precision: 0%, Recall: 0%, F1 Score: 0%, Predictions: [0 0 0 0 0 0 0 0 0 0]
Limitations	Assumes linearity, Sensitive to outliers	Assumes linearity in log odds, Struggles with non-linear relationships

Influence and Optimization of Hyperparameters:

Modifying hyperparameters has a significant impact on the results of linear and logistic regression models. It is critical to use appropriate hyperparameter optimization strategies when fine-tuning models for improved performance and generalization.

- **for Linear Regression**: Learning Rate and Iterations impact convergence and fitting. Too high can cause overshooting, too low results in slow convergence.
- **Logistic Regression**: Learning Rate, Iterations, and Regularization Strength affect convergence and balance bias-variance tradeoff. Insufficient iterations may lead to underfitting, whereas too many can lead to overfitting or unnecessary computation.

- **Strategies for Optimization of hyperparameters:**
  Hyperparameter optimization is a crucial step in improving the performance of machine learning models.

1. **Grid Search:**
   - Systematically explores a predefined set of hyperparameter values.
   - Exhaustively searches the entire parameter space.
2. **Random Search:**
   - Randomly selects hyperparameter combinations from a predefined search space.
   - More computationally efficient than grid search. Allows exploration of diverse parameter sets.
3. **Bayesian Optimization:**
   - Utilizes probabilistic models to guide the search for optimal hyperparameters.
   - Efficiently balances exploration and exploitation, adapting to the shape of the response surface.

Alterations to these parameters, we can optimize our models to attain enhanced accuracy, mitigate overfitting, and enhance their ability to generalize to new and unseen data.

Appendix

"Python Machine Learning" by Sebastian Raschka and Vahid Mirjalili

https://stackoverflow.com/

https://www.analyticsvidhya.com/blog/2020/12/beginners-take-how-logistic-regression-is-related-to-linear-regression/

https://www.geeksforgeeks.org/understanding-logistic-regression/?ref=lbp#logistic-function-sigmoid-function