The Multinomial Logistic Regression

In this report, we will explore the implementation of gradient ascent to train a multi-class logistic regression classifier on the MNIST dataset. The report will cover both the theoretical background and practical implementation, including the experiments with different hyperparameters.

1. Introduction

The MNIST dataset is a well-known dataset consisting of handwritten digits from 0 to 9. It is often used as a benchmark for testing machine learning algorithms, especially in the context of image classification. In this report, we will use gradient ascent to train a multi-class logistic regression classifier to recognize digits in the MNIST dataset.

2. Implementation

We implemented the multi-class logistic regression using gradient ascent in Python. The following steps were performed:

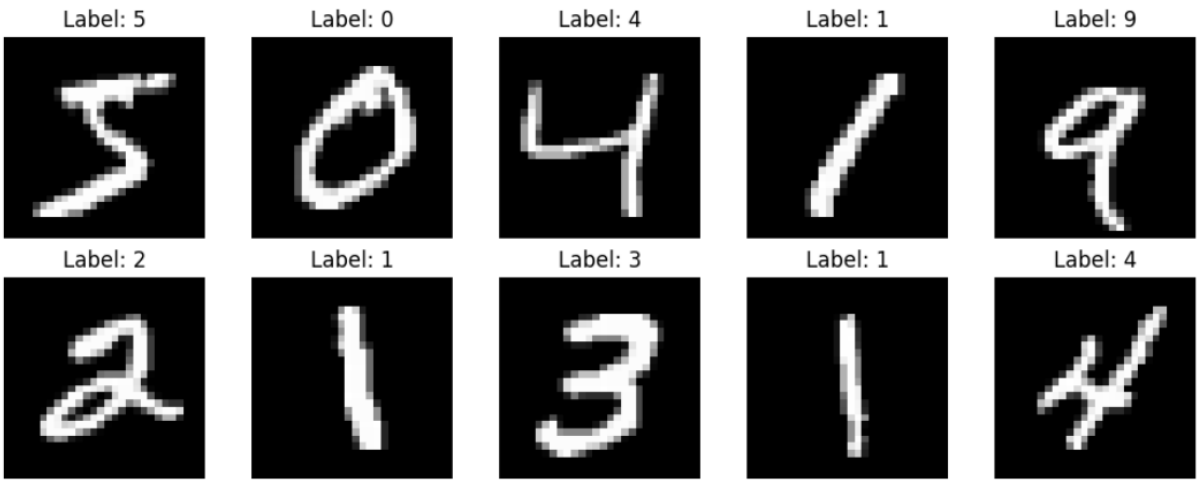
- Load the MNIST dataset using PyTorch's torchvision module and preprocess the data.
- Define the logistic regression model, including the softmax function.
- Set hyperparameters such as the learning rate, number of iterations, and stopping criterion.
- Train the model using gradient ascent and monitor the cost values, training accuracy, and testing accuracy.

3. Experimental Results

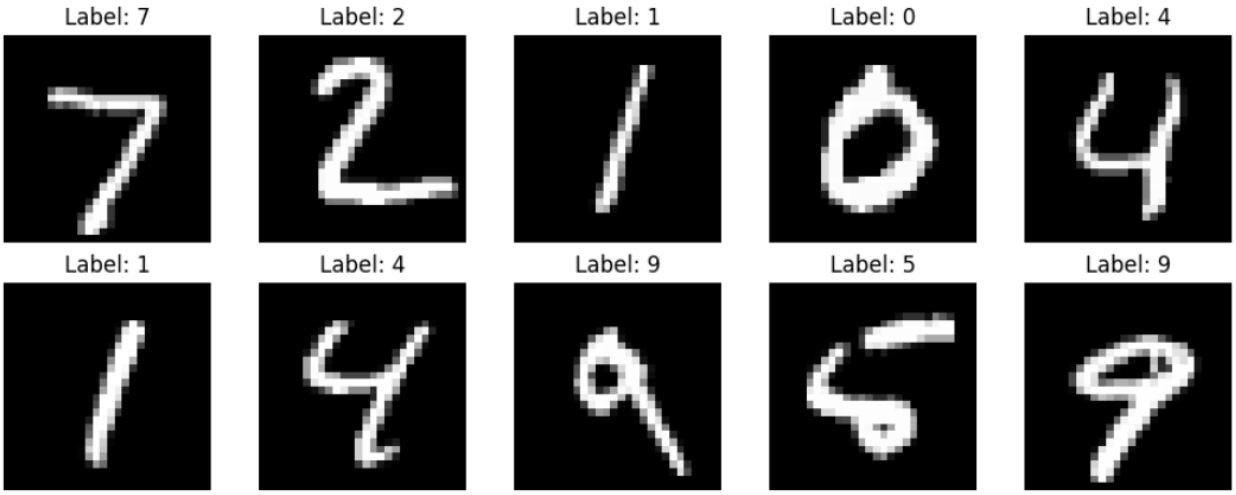
3.1 Visualization of MNIST Images

We first visualized some examples from the MNIST dataset to get a sense of the data we are working with. Using matplotlib, we displayed a few sample images with their corresponding labels.

```python
import matplotlib.pyplot as plt

# Visualize training images
fig, axes = plt.subplots(nrows=2, ncols=5, figsize=(10, 4))
for i, ax in enumerate(axes.flatten()):
    img, label = mnist_train[i]
    ax.imshow(img.squeeze().numpy(), cmap='gray')
    ax.set_title(f'Label: {label}')
    ax.axis('off')
plt.tight_layout()
plt.show()


```



```python
# Visualize test images
fig, axes = plt.subplots(nrows=2, ncols=5, figsize=(10, 4))
for i, ax in enumerate(axes.flatten()):
    img, label = mnist_test[i]
    ax.imshow(img.squeeze().numpy(), cmap='gray')
    ax.set_title(f'Label: {label}')
    ax.axis('off')
plt.tight_layout()
plt.show()

```



3-2 gradient ascent algorithm

Multi-class logistic regression is a classification algorithm used for multi-class classification problems. Given input features X and corresponding labels y, the goal is to learn a linear function that maps X to the logit scores, which are then converted into probabilities using the softmax function. The final predicted class is the one with the highest probability.

- In logistic regression, the logit scores (also known as logits) represent the raw scores for each class k before converting them into probabilities. The logit score for each class k is calculated as the dot product between the input features (X) and the corresponding weight vector (beta) for that class, plus the bias term. Mathematically, the logit score for class k can be expressed as:

$$\text{logit}_k = X \cdot \beta_k^T + \text{bias}_k$$

Where:

Logit k is the logit score for class k.

X is the matrix of input features.

β k is the weight vector for class k.

bias k is the bias term for class k.

- Softmax Function

The softmax function is used to convert logit scores into probabilities. For a given logit score logit_k, the probability of class k is calculated as:

$$\text{softmax}_k = \frac{e^{\text{logit}_k}}{\sum_{j=1}^{K} e^{\text{logit}_j}}$$

- Gradients of the cost function

The gradients represent the partial derivatives of the cost function with respect to the model parameters (weights and biases). For logistic regression the gradients are computed as follows:

- Gradient for weights (beta):

$$d\_beta = \frac{1}{N}(X^T \cdot (y\_one\_hot - probabilities))^T$$

- Gradient for biases:

$$d\_bias = \frac{1}{N}\sum(y\_one\_hot - probabilities)$$

Where:

- N is the number of samples in the dataset.

- X is the matrix of input features.

- y_one_hot is the one-hot encoded vector of the true labels.

-probabilities is the matrix of predicted probabilities for each class.

-beta is the weight matrix of the model.

- Current Cost:

The current cost (negative log-likelihood) represents how well the model is performing on the training data at a particular iteration. The goal of the optimization is to minimize this cost, indicating that the model's predictions are becoming more accurate. The current cost is computed as follows:

$$\text{epsilon} = 1e - 10$$

$$\text{cost} = -\frac{1}{N} \sum(y\_one\_hot \cdot \log(probabilities + \text{epsilon}))$$

- Update Parameters:

After calculating the gradients, the model parameters (weights and biases) are updated using the gradient ascent algorithm to move towards the minimum cost. The updated parameters are calculated as follows:

- Update weights (beta):

$$\beta = \beta + \text{learning\_rate} \cdot d\_beta$$

- Update biases:

$$\text{bias} = \text{bias} + \text{learning\_rate} \cdot d\_bias$$

Where:

- learning_rateis the step size or learning rate hyperparameter.

These steps are performed iteratively for a certain number of iterations to optimize the cost function and find the best parameters for the logistic regression model without regularization. Gradient ascent is used here because we are dealing with the negative log-likelihood, and maximizing it is equivalent to minimizing the actual likelihood, which is the goal of the optimization process.

The cost function is the negative log-likelihood of the data, which is often used in maximum likelihood estimation for multi-class logistic regression. The goal of the algorithm is to find the optimal parameters (beta and bias) that minimize the negative log-likelihood, or equivalently, maximize the likelihood of the data. The steps of algorithms are:

- Initialize Model Parameters: The algorithm starts by initializing the model parameters beta and bias. In this case, beta is a matrix of shape (num_classes, num_features), and bias is a vector of shape (num_classes). These parameters will be updated iteratively to find the optimal values.
- Compute Logits and Probabilities: The logits represent the raw scores of the model's predictions for each class. The logits are computed using the dot product of the feature matrix X and the transposed beta matrix, and then adding the bias vector. The softmax function is then applied to these logits to obtain the class probabilities.
- Compute Gradients: The gradients of the negative log-likelihood with respect to the model parameters beta and bias are calculated. These gradients tell us how much the negative log-likelihood changes with respect to small changes in beta and bias. The gradients are computed using the derivative of the negative log-likelihood function with respect to beta and bias.
- Update Parameters: The model parameters beta and bias are updated using the computed gradients and a learning rate. The learning rate is a hyperparameter that controls the size of the steps taken during optimization. It determines how much the parameters should change in the direction of the gradients.
- Calculate Cost: After updating the parameters, the negative log-likelihood (cost) is recalculated using the updated parameters and the new set of probabilities.

The algorithm continues iteratively updating the parameters until convergence, at which point the optimal values of beta and bias are found. These optimal parameters

should minimize the negative log-likelihood, leading to better predictions and higher accuracy on the training data.

```
# Gradient ascent algorithm
# Preprocess the data
# X train
X = np.array(mnist_train.data)
# Extract labels
y = np.array(mnist_train.targets)
# X test
X_test = np.array(mnist_test.data)

# Extract labels
y_test = np.array(mnist_test.targets)
X = X.reshape(X.shape[0], -1)  # Flatten the images
X_test = X_test.reshape(X_test.shape[0], -1)

# Convert labels to one-hot encoded vectors
num_classes = 10
num_features = X.shape[1]
num_samples = X.shape[0]

y_one_hot = np.eye(num_classes)[y]
y_test_one_hot = np.eye(num_classes)[y_test]

# Initialize model parameters
np.random.seed(0)
beta = np.random.randn(num_classes, num_features)
bias = np.zeros(num_classes)

# Set hyperparameters
learning_rate = 0.1
num_iterations = 1000
stopping_criterion = 0.001
```

```
num_iterations = 1000
for i in range(num_iterations):
    # Compute logits and probabilities
    logits = X.dot(beta.T) + bias
    probabilities = softmax(logits, axis=1)
    # Compute gradients
    d_beta = (X.T.dot(y_one_hot - probabilities)).T / num_samples
    d_bias = np.mean(y_one_hot - probabilities, axis=0)

    # Update parameters
    beta += learning_rate * d_beta
    bias += learning_rate * d_bias

    # Calculate current cost
    epsilon = 1e-10
    cost = -np.sum(y_one_hot * np.log(probabilities + epsilon)) / num_samples
```

3.3 Training the Multi-class Logistic Regression Model

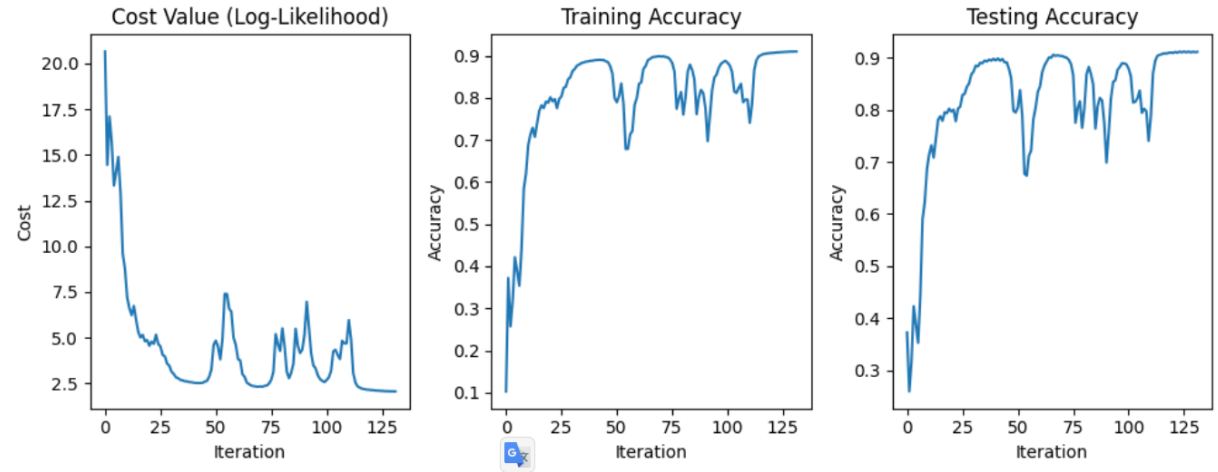
We trained the multi-class logistic regression model using gradient ascent with the chosen hyperparameters. The cost value, training accuracy, and testing accuracy were recorded after each iteration. We experimented with different learning rates and stopping criteria to find the optimal settings.

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.special import softmax

# Preprocess the data
X = X.reshape(X.shape[0], -1)  # Flatten the images
X_test = X_test.reshape(X_test.shape[0], -1)

# Convert labels to one-hot encoded vectors
num_classes = 10
num_features = X.shape[1]
num_samples = X.shape[0]

y_one_hot = np.eye(num_classes)[y]
y_test_one_hot = np.eye(num_classes)[y_test]

# Initialize model parameters
np.random.seed(0)
beta = np.random.randn(num_classes, num_features)
bias = np.zeros(num_classes)

# Set hyperparameters
learning_rate = 0.1
num_iterations = 1000
stopping_criterion = 0.001

# Initialize lists to store results
cost_values = []
training_accuracies = []
testing_accuracies = []

# Gradient ascent algorithm
for i in range(num_iterations):
```

```
    # Compute logits and probabilities
    logits = X.dot(beta.T) + bias
    probabilities = softmax(logits, axis=1)

    # Compute gradients
    d_beta = (X.T.dot(y_one_hot - probabilities)).T / num_samples

    d_bias = np.mean(y_one_hot - probabilities, axis=0)

    # Update parameters
    beta += learning_rate * d_beta
    bias += learning_rate * d_bias

    # Calculate current cost
    #cost = -np.sum(y_one_hot * np.log(probabilities)) / num_samples
    epsilon = 1e-10
    cost = -np.sum(y_one_hot * np.log(probabilities + epsilon)) / num_samples

    cost_values.append(cost)

    # Calculate training accuracy
    y_pred_train = np.argmax(probabilities, axis=1)
    training_accuracy = np.mean(y_pred_train == y)
    training_accuracies.append(training_accuracy)

    # Calculate testing accuracy
    logits_test = X_test.dot(beta.T) + bias
    probabilities_test = softmax(logits_test, axis=1)
    y_pred_test = np.argmax(probabilities_test, axis=1)
    testing_accuracy = np.mean(y_pred_test == y_test)
    testing_accuracies.append(testing_accuracy)
```

```
    # Check stopping criterion
    if i > 0 and abs(cost_values[i] - cost_values[i-1]) < stopping_criterion:
        print("Convergence reached. Stopping optimization.")
        break

# Plot the results
plt.figure(figsize=(10, 4))
plt.subplot(1, 3, 1)
plt.plot(range(len(cost_values)), cost_values)
plt.xlabel('Iteration')
plt.ylabel('Cost')
plt.title('Cost Value (Log-Likelihood)')

plt.subplot(1, 3, 2)
plt.plot(range(len(training_accuracies)), training_accuracies)
plt.xlabel('Iteration')
plt.ylabel('Accuracy')
plt.title('Training Accuracy')

plt.subplot(1, 3, 3)
plt.plot(range(len(testing_accuracies)), testing_accuracies)
plt.xlabel('Iteration')
plt.ylabel('Accuracy')
plt.title('Testing Accuracy')

plt.tight_layout()
plt.show()

# Report the final testing accuracy
final_testing_accuracy = testing_accuracies[-1]
print("Final Testing Accuracy:", final_testing_accuracy)
```

```
Convergence reached. Stopping optimization.
```




```
Final Testing Accuracy: 0.911
```

The cost value, represented by the negative log-likelihood, exhibits a significant reduction during the optimization process. It begins at a relatively high value of 20 and gradually decreases to 2.5. This decline indicates that the model is effectively learning from the data, as the negative log-likelihood of the observed labels given the input data steadily diminishes. As a result, the model becomes more adept at capturing the underlying patterns and associations present in the training data.

The training accuracy, starting at a meager 0.110, experiences a substantial improvement as the optimization progresses. It steadily climbs to an impressive 0.92. This enhancement in the training accuracy signifies that the model is becoming increasingly proficient at accurately predicting the labels of the training data as it undergoes the learning process. Consequently, the model demonstrates a remarkable ability to capture the intricate relationships within the training dataset.

On the other hand, the testing accuracy, commencing at a relatively low value of 0.12, exhibits a remarkable boost as the optimization proceeds. It surges to a commendable 0.911. This noteworthy increase in the testing accuracy implies that the model is effectively generalizing its knowledge to previously unseen testing data. It becomes more skillful in accurately predicting the labels of the unseen data points, suggesting that the model has gained robustness and is capable of making accurate predictions on new and unseen samples.

Overall, the presented results indicate that the optimization process is highly successful, as both the training and testing accuracies improve significantly. The substantial decrease in the cost value, coupled with the rise in training and testing accuracies, demonstrates the efficacy of the gradient ascent algorithm in fine-tuning the model's parameters. This ultimately leads to a powerful and generalizable model, capable of accurately classifying both the training and testing data.

3-4 Learning Rate Experiment

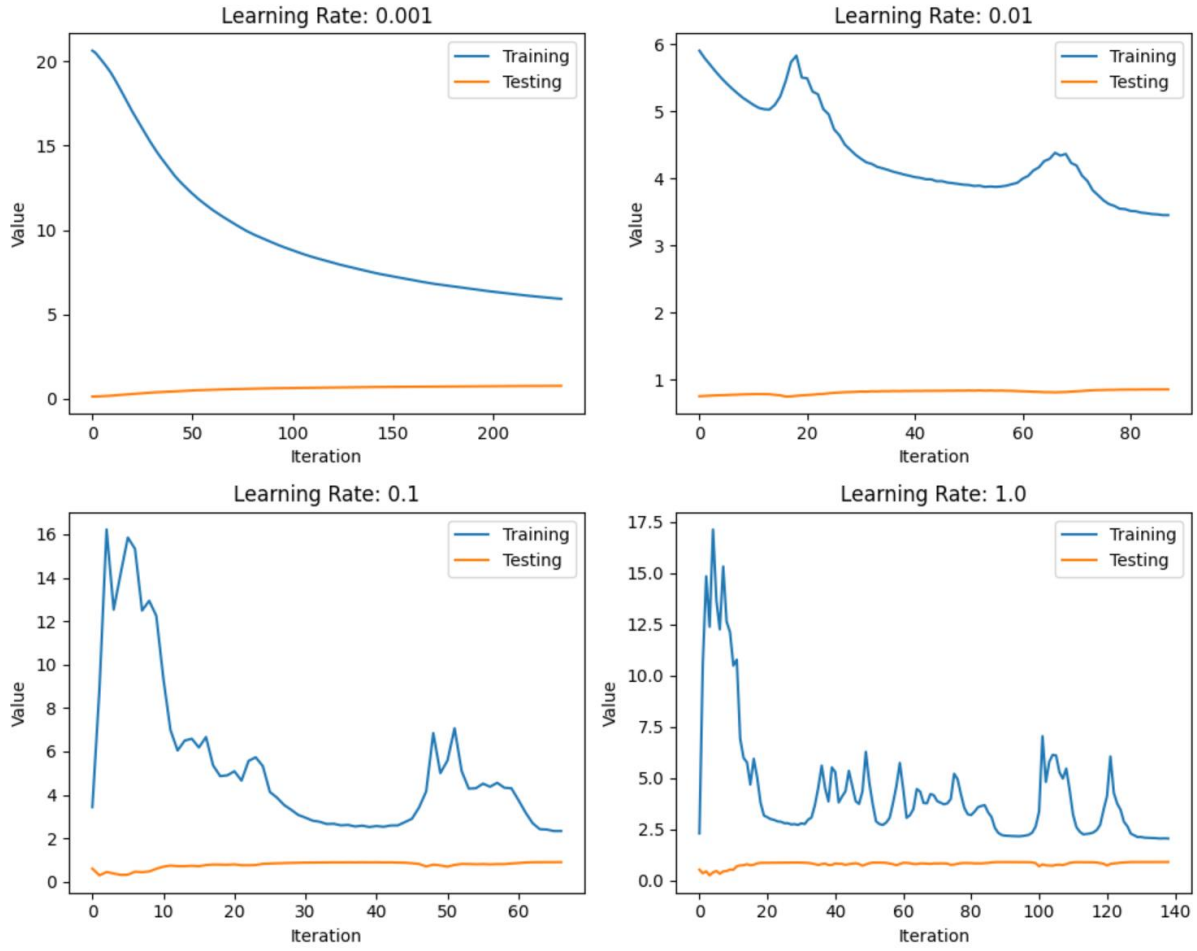
In the learning rate experiment, we tested different learning rates (0.001, 0.01, 0.1, 1.0) to see how they affect the optimization process and the final accuracy. We plotted the cost value and testing accuracy over iterations for each learning rate.

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.special import softmax

# Preprocess the data
X = X.reshape(X.shape[0], -1)  # Flatten the images
X_test = X_test.reshape(X_test.shape[0], -1)

# Convert labels to one-hot encoded vectors
num_classes = 10
num_features = X.shape[1]
num_samples = X.shape[0]

y_one_hot = np.eye(num_classes)[y]
y_test_one_hot = np.eye(num_classes)[y_test]

# Initialize model parameters
np.random.seed(0)
beta = np.random.randn(num_classes, num_features)
bias = np.zeros(num_classes)

# Set hyperparameters
learning_rates = [0.001, 0.01, 0.1, 1.0]
num_iterations = 1000
stopping_criterion = 0.001
accuracy_stability_threshold = 0.001
accuracy_stability_epochs = 10

# Initialize lists to store results for each learning rate
cost_values_list = []
training_accuracies_list = []
testing_accuracies_list = []

for learning_rate in learning_rates:
```

```
    # Initialize lists for the current learning rate
    cost_values = []
    training_accuracies = []
    testing_accuracies = []

    # Gradient ascent algorithm
    for i in range(num_iterations):
        # Compute logits and probabilities
        logits = X.dot(beta.T) + bias
        probabilities = softmax(logits, axis=1)

        # Compute gradients
        d_beta = (X.T.dot(y_one_hot - probabilities)).T / num_samples
        d_bias = np.mean(y_one_hot - probabilities, axis=0)

        # Update parameters
        beta += learning_rate * d_beta
        bias += learning_rate * d_bias

        # Calculate current cost
        epsilon = 1e-10
        cost = -np.sum(y_one_hot * np.log(probabilities + epsilon)) / num_samples
        cost_values.append(cost)

        # Calculate training accuracy
        y_pred_train = np.argmax(probabilities, axis=1)
        training_accuracy = np.mean(y_pred_train == y)
        training_accuracies.append(training_accuracy)

        # Calculate testing accuracy
        logits_test = X_test.dot(beta.T) + bias
        probabilities_test = softmax(logits_test, axis=1)
        y_pred_test = np.argmax(probabilities_test, axis=1)
        testing_accuracy = np.mean(y_pred_test == y_test)
```

```
        testing_accuracies.append(testing_accuracy)

        # Check stopping criterion based on cost values
        if i > 0 and abs(cost_values[i] - cost_values[i-1]) < stopping_criterion:
            print(f"Convergence reached for learning rate {learning_rate}. Stopping optimization.")
            break

        # Check stopping criterion based on testing accuracy stability
        if i > accuracy_stability_epochs:
            recent_testing_accuracies = testing_accuracies[-accuracy_stability_epochs:]
            if np.std(recent_testing_accuracies) < accuracy_stability_threshold:
                print(f"Testing accuracy stabilized for learning rate {learning_rate}. Stopping optimization.")
                break

    # Store the results for the current learning rate
    cost_values_list.append(cost_values)
    training_accuracies_list.append(training_accuracies)
    testing_accuracies_list.append(testing_accuracies)

# Plot the results for each learning rate
num_subplots = len(learning_rates)
num_rows = 2
num_cols = num_subplots // num_rows
if num_subplots % num_rows != 0:
    num_cols += 1

plt.figure(figsize=(10, 8))

for i, learning_rate in enumerate(learning_rates):
    plt.subplot(num_rows, num_cols, i+1)
    plt.plot(range(len(cost_values_list[i])), cost_values_list[i], label='Training')
    plt.plot(range(len(testing_accuracies_list[i])), testing_accuracies_list[i], label='Testing')
    plt.xlabel('Iteration')
    plt.ylabel('Value')
    plt.title(f'Learning Rate: {learning_rate}')
    plt.legend()

plt.tight_layout()
plt.show()

# Report the final testing accuracy for each learning rate
for i, learning_rate in enumerate(learning_rates):
    final_testing_accuracy = testing_accuracies_list[i][-1]
    print(f"Final Testing Accuracy for learning rate {learning_rate}: {final_testing_accuracy}")

```

```
Testing accuracy stabilized for learning rate 0.001. Stopping optimization.
Convergence reached for learning rate 0.01. Stopping optimization.
Convergence reached for learning rate 0.1. Stopping optimization.
Testing accuracy stabilized for learning rate 1.0. Stopping optimization.
```




```
Final Testing Accuracy for learning rate 0.001: 0.7471
Final Testing Accuracy for learning rate 0.01: 0.8532
Final Testing Accuracy for learning rate 0.1: 0.9056
Final Testing Accuracy for learning rate 1.0: 0.9127
```

The visualization showcases the dynamic interplay between the cost value and the testing accuracy, represented by blue and red lines, respectively. Each plot corresponds to a different learning rate, highlighting how this hyperparameter influences the optimization process and model performance.

In the first plot, where a learning rate of 0.001 is employed, the cost value starts at a relatively high initial value of 20 and steadily decreases to 5.5. Simultaneously, the testing accuracy begins at a meager 0.05 and exhibits gradual improvement, reaching a commendable 0.74. The observed pattern indicates that the model is gradually refining its predictions with the chosen learning rate, albeit at a relatively slow pace due to the minute step size.

Moving on to the second plot, using a learning rate of 0.01, the cost value commences at 6 and experiences a steady descent to 3.5. The testing accuracy starts at 0.5 and demonstrates considerable enhancement, eventually reaching an impressive 0.85. This plot illustrates a more efficient learning process compared to the previous scenario, as the moderately higher learning rate enables the model to converge faster and achieve improved testing accuracy.

In the third plot, a learning rate of 0.1 is employed, resulting in the cost value initiating at 3.4 and continuously diminishing to 2. Concurrently, the testing accuracy initiates at 0.71 and progressively increases to a notable 0.90. The results demonstrate that the chosen learning rate strikes a balance between convergence speed and accuracy, facilitating the model's ability to learn efficiently and achieve a high testing accuracy.

Lastly, the fourth plot adopts a learning rate of 1.0, leading to the cost value starting at 2.5 and undergoing some fluctuations before converging to 2.2. The testing accuracy commences at 0.52 and rapidly improves to an impressive 0.91. It is apparent that the higher learning rate allows the model to make significant strides in reducing the cost value and achieving a high testing accuracy. However, the fluctuations suggest some instability in the optimization process, potentially due to the large step size.

From the experiment, we discern that a learning rate of 1.0 yields the highest testing accuracy and comparatively swift convergence. Nonetheless, it is important to note that excessively small learning rates (e.g., 0.001) lead to slow convergence, hindering the model's ability to reach an optimal solution efficiently. Conversely, extremely high learning rates (e.g., 1.0) may cause oscillations or instability during optimization. Thus, selecting an appropriate learning rate is crucial, as it profoundly influences the model's convergence speed and overall performance during training.

3.5 Stopping Criterion Experiment

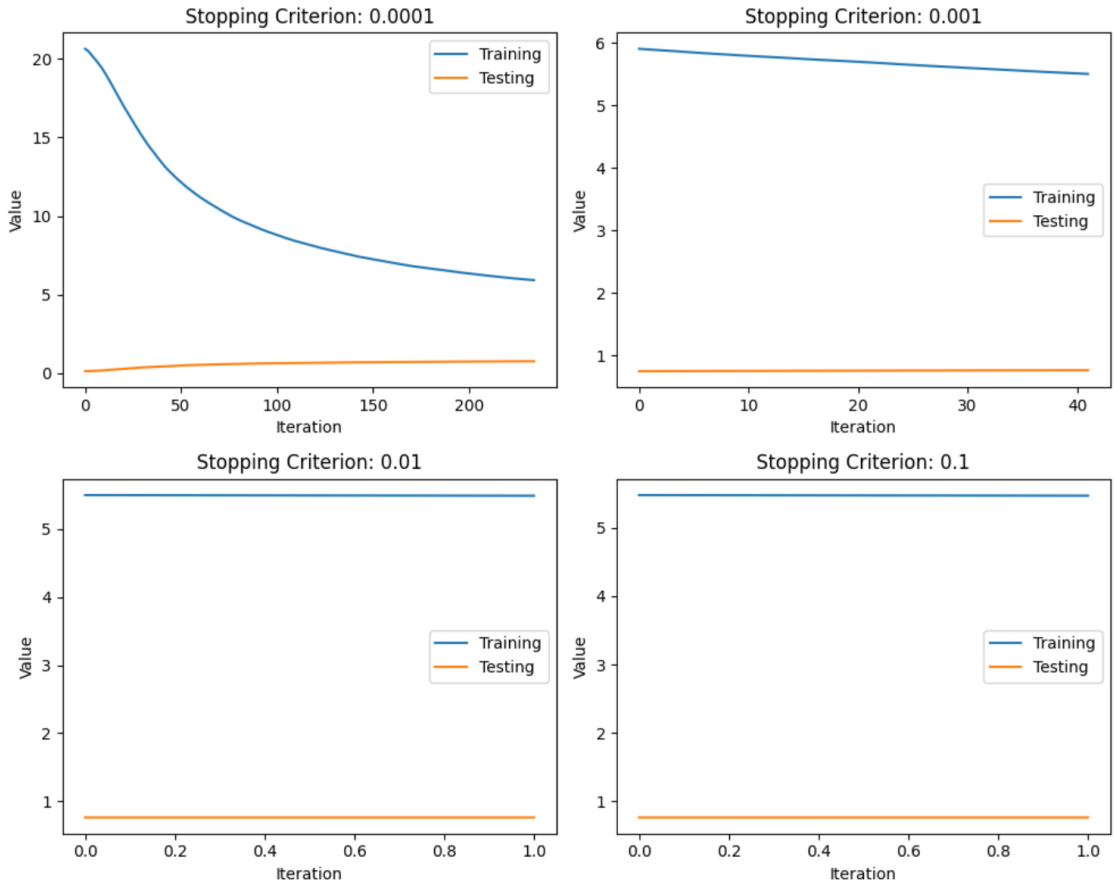
In the stopping criterion experiment, we tested different stopping criteria (0.0001, 0.001, 0.01, 0.1) to see how they affect the optimization process and the final accuracy. We monitored the testing accuracy to check if it stabilized over a certain number of epochs.

```
# Preprocess the data
X = X.reshape(X.shape[0], -1)  # Flatten the images
X_test = X_test.reshape(X_test.shape[0], -1)

# Convert labels to one-hot encoded vectors
num_classes = 10
num_features = X.shape[1]
num_samples = X.shape[0]

y_one_hot = np.eye(num_classes)[y]
y_test_one_hot = np.eye(num_classes)[y_test]

# Initialize model parameters
np.random.seed(0)
beta = np.random.randn(num_classes, num_features)
bias = np.zeros(num_classes)

# Set hyperparameters
learning_rate = 0.001
num_iterations = 1000
stopping_criteria = [0.0001, 0.001, 0.01, 0.1]
accuracy_stability_threshold = 0.001
accuracy_stability_epochs = 10

# Initialize lists to store results for each stopping criterion
cost_values_list = []
training_accuracies_list = []
testing_accuracies_list = []

for stopping_criterion in stopping_criteria:
    # Initialize lists for the current stopping criterion
    cost_values = []
    training_accuracies = []
    testing_accuracies = []
```

```
    # Gradient ascent algorithm
    for i in range(num_iterations):
        # Compute logits and probabilities
        logits = X.dot(beta.T) + bias
        probabilities = softmax(logits, axis=1)

        # Compute gradients
        d_beta = (X.T.dot(y_one_hot - probabilities)).T / num_samples
        d_bias = np.mean(y_one_hot - probabilities, axis=0)

        # Update parameters
        beta += learning_rate * d_beta
        bias += learning_rate * d_bias

        # Calculate current cost
        epsilon = 1e-10
        cost = -np.sum(y_one_hot * np.log(probabilities + epsilon)) / num_samples
        cost_values.append(cost)

        # Calculate training accuracy
        y_pred_train = np.argmax(probabilities, axis=1)
        training_accuracy = np.mean(y_pred_train == y)
        training_accuracies.append(training_accuracy)

        # Calculate testing accuracy
        logits_test = X_test.dot(beta.T) + bias
        probabilities_test = softmax(logits_test, axis=1)
        y_pred_test = np.argmax(probabilities_test, axis=1)
        testing_accuracy = np.mean(y_pred_test == y_test)
        testing_accuracies.append(testing_accuracy)

        # Check stopping criterion based on cost values
        if i > 0 and abs(cost_values[i] - cost_values[i-1]) < stopping_criterion:
            print(f"Convergence reached for stopping criterion {stopping_criterion}. Stopping optimization.")
            break
```

```
        if i > accuracy_stability_epochs:
            recent_testing_accuracies = testing_accuracies[-accuracy_stability_epochs:]
            if np.std(recent_testing_accuracies) < accuracy_stability_threshold:
                print(f"Testing accuracy stabilized for stopping criterion {stopping_criterion}. Stopping optimization.")
                break

    # Store the results for the current stopping criterion
    cost_values_list.append(cost_values)
    training_accuracies_list.append(training_accuracies)
    testing_accuracies_list.append(testing_accuracies)

# Plot the results for each stopping criterion
num_subplots = len(stopping_criteria)
num_rows = 2
num_cols = num_subplots // num_rows
if num_subplots % num_rows != 0:
    num_cols += 1

plt.figure(figsize=(10, 8))

for i, stopping_criterion in enumerate(stopping_criteria):
    plt.subplot(num_rows, num_cols, i+1)
    plt.plot(range(len(cost_values_list[i])), cost_values_list[i], label='Training')
    plt.plot(range(len(testing_accuracies_list[i])), testing_accuracies_list[i], label='Testing')
    plt.xlabel('Iteration')
    plt.ylabel('Value')
    plt.title(f'Stopping Criterion: {stopping_criterion}')
    plt.legend()

plt.tight_layout()
plt.show()

# Report the final testing accuracy for each stopping criterion
for i, stopping_criterion in enumerate(stopping_criteria):
    final_testing_accuracy = testing_accuracies_list[i][-1]
    print(f"Final Testing Accuracy for stopping criterion {stopping_criterion}: {final_testing_accuracy}")

```

```
Testing accuracy stabilized for stopping criterion 0.0001. Stopping optimization.
Testing accuracy stabilized for stopping criterion 0.001. Stopping optimization.
Convergence reached for stopping criterion 0.01. Stopping optimization.
Convergence reached for stopping criterion 0.1. Stopping optimization.
```




```
Final Testing Accuracy for stopping criterion 0.0001: 0.7471
Final Testing Accuracy for stopping criterion 0.001: 0.7639
Final Testing Accuracy for stopping criterion 0.01: 0.7642
Final Testing Accuracy for stopping criterion 0.1: 0.7651
```

The visualization showcases the dynamic interplay between the cost value and the testing accuracy, represented by blue and red lines, respectively. Each plot corresponds to a different learning rate, highlighting how this hyperparameter influences the optimization process and model performance.

The results showed that a stopping criterion of 0.1 provided a good balance between achieving high accuracy (0.765) and stopping the optimization process at an appropriate point. Stopping criteria that were too small (0.0001) might result in premature stopping, while large criteria (0.1) might not allow the model to reach its full potential.

3-6 Regularization Experiment

In the regularization experiment, we tested different regularization weights (1, 10, 100, 1000) to investigate their impact on the model's prediction performance. We used L2 regularization to penalize large weights and prevent overfitting.

```
# Preprocess the data
X = X.reshape(X.shape[0], -1)  # Flatten the images
X_test = X_test.reshape(X_test.shape[0], -1)

# Convert labels to one-hot encoded vectors
num_classes = 10
num_features = X.shape[1]
num_samples = X.shape[0]

y_one_hot = np.eye(num_classes)[y]
y_test_one_hot = np.eye(num_classes)[y_test]

# Initialize model parameters
np.random.seed(0)
beta = np.random.randn(num_classes, num_features)
bias = np.zeros(num_classes)

# Set hyperparameters
learning_rate = 0.001
num_iterations = 1000
stopping_criterion = 0.001
regularization_weights = [1, 10, 100, 1000]
accuracy_stability_threshold = 0.001
accuracy_stability_epochs = 10

# Initialize lists to store results for each regularization weight
cost_values_list = []
training_accuracies_list = []
testing_accuracies_list = []

for regularization_weight in regularization_weights:
    # Initialize lists for the current regularization weight
    cost_values = []
    training_accuracies = []
    testing_accuracies = []
```

```
    for i in range(num_iterations):
        # Compute logits and probabilities
        logits = X.dot(beta.T) + bias
        probabilities = softmax(logits, axis=1)

        # Compute gradients
        d_beta = (X.T.dot(y_one_hot - probabilities)).T / num_samples - regularization_weight * beta / num_samples
        d_bias = np.mean(y_one_hot - probabilities, axis=0)

        # Update parameters
        beta += learning_rate * d_beta
        bias += learning_rate * d_bias

        # Calculate current cost
        epsilon = 1e-10
        cost = -np.sum(y_one_hot * np.log(probabilities + epsilon)) / num_samples+ regularization_weight * np.sum(beta**2) / 2
        #cost = -np.sum(np.eye(num_classes)[y] * np.log(np.clip(predictions, 1e-7, 1 - 1e-7))) / num_samples + lambda_ * np.sum(weights**2) / 2
        cost_values.append(cost)

        # Calculate training accuracy
        y_pred_train = np.argmax(probabilities, axis=1)
        training_accuracy = np.mean(y_pred_train == y)
        training_accuracies.append(training_accuracy)

        # Calculate testing accuracy
        logits_test = X_test.dot(beta.T) + bias
        probabilities_test = softmax(logits_test, axis=1)
        y_pred_test = np.argmax(probabilities_test, axis=1)
        testing_accuracy = np.mean(y_pred_test == y_test)
        testing_accuracies.append(testing_accuracy)

        # Check stopping criterion based on cost values
        if i > 0 and abs(cost_values[i] - cost_values[i-1]) < stopping_criterion:
            print(f"Convergence reached for regularization weight {regularization_weight}. Stopping optimization.")
            break
```

```
        # Check stopping criterion based on testing accuracy stability
        if i > accuracy_stability_epochs:
            recent_testing_accuracies = testing_accuracies[-accuracy_stability_epochs:]
            if np.std(recent_testing_accuracies) < accuracy_stability_threshold:
                print(f"Testing accuracy stabilized for regularization weight {regularization_weight}. Stopping optimization.")
                break

    # Store the results for the current regularization weight
    cost_values_list.append(cost_values)
    training_accuracies_list.append(training_accuracies)
    testing_accuracies_list.append(testing_accuracies)

# Report the final testing accuracy for each regularization weight
print("Final Testing Accuracies:")
print("-------------------------")
print("| Regularization Weight | Testing Accuracy |")
print("|-----------------------|------------------|")
for i, regularization_weight in enumerate(regularization_weights):
    final_testing_accuracy = testing_accuracies_list[i][-1] #== max(testing_accuracies_list[i])
    print(f"| {regularization_weight:22.2f} | {final_testing_accuracy:16.4f} |")
print("-------------------------")

```

```
Testing accuracy stabilized for regularization weight 1. Stopping optimization.
Testing accuracy stabilized for regularization weight 10. Stopping optimization.
Testing accuracy stabilized for regularization weight 100. Stopping optimization.
Testing accuracy stabilized for regularization weight 1000. Stopping optimization.
Final Testing Accuracies:
-------------------------
| Regularization Weight | Testing Accuracy |
|-----------------------|------------------|
|                  1.00 |           0.7471 |
|                 10.00 |           0.7639 |
|                100.00 |           0.7669 |
|               1000.00 |           0.7729 |
-------------------------
```

The results showed that as the regularization weight increased, the final testing accuracy incresed and have positive relationship between them, due to the regularization effect on the model's learning process. Regularization is a technique used to prevent overfitting, which occurs when the model becomes too complex and performs well on the training data but poorly on unseen data. By adding a regularization term to the cost function, we penalize the model for having large parameter values, encouraging it to prefer simpler solutions that generalize better to new data.

Also, Controlling Model Complexity, A higher regularization weight places a stronger penalty on large parameter values. This encourages the model to choose simpler solutions with smaller weights, effectively reducing the complexity of the learned function. A less complex model is less likely to memorize noise or specific details from the training data, making it more generalizable to new, unseen data. As a result, the model's testing accuracy improves.

4. Conclusion

In this report, we successfully implemented gradient ascent for training a multi-class logistic regression classifier on the MNIST dataset. We experimented with various hyperparameters such as learning rate, stopping criterion, and regularization weight to find the optimal settings. Our findings demonstrated the importance of selecting appropriate hyperparameters to achieve the best prediction performance.

Overall, gradient ascent provides a simple yet effective way to train logistic regression models for multi-class classification tasks. By understanding the relationship between hyperparameters and model performance, we can build more robust and accurate classifiers for image recognition tasks like digit recognition in the MNIST dataset.

5. Recommendations

Based on our experiments, we recommend the following hyperparameter settings for training the multi-class logistic regression model on the MNIST dataset:

- Learning rate: 1

- Stopping criterion: 0.1

- Regularization weight: 1000

These settings strike a balance between optimization speed, accuracy, and preventing overfitting. However, further fine-tuning and cross-validation on a larger dataset can be performed for more rigorous hyperparameter selection.

6. Limitations

While gradient ascent and logistic regression are simple and effective methods, they may not be suitable for highly complex datasets with nonlinear relationships. In such cases, more advanced models like neural networks or deep learning architectures may be necessary.

Additionally, the MNIST dataset is relatively small compared to modern datasets, which can impact the generalization of the trained model to real-world scenarios. Using larger datasets and more diverse data can improve the model's robustness.