6CS012 - Artificial Intelligence and Machine Learning. Image Classification Using Softmax Regression.

Prepared By: Siman Giri {Module Leader - 6CS012}

Feb 28, 2025

Worksheet - 2.

1 Instructions

This worksheet contains programming exercises based on the material discussed from the slides.This is a graded exercise and are to be completed on your own and is compulsory to submit.
Please answer the questions below using python in the Jupyter Notebook and follow the guidelines below:

- This worksheet must be completed individually.
- All the solutions must be written in Jupyter Notebook.

Learning Objectives:

- Set up the computing environment using Jupyter Notebook or Google Colab (Recommended).
- Understand the key steps in building a machine learning classification model.
- Review and apply essential Python libraries: **NumPy**, **Matplotlib**, and **Scikit-Learn**.
- Implement and evaluate **Softmax Regression** for handwritten digit classification using the **MNIST dataset**.

2 Introduction.

The following implementation follows the Empirical Risk Minimization (ERM) Framework. We will first develop utility functions to load, display, and visualize the image dataset. Next, we will implement helper functions necessary for training and evaluating a Softmax Regression model, aligning with the key components of the ERM framework.

1. Problem Statement and Dataset:

In this **Worksheet** you are expected to build a **Multinomial Logistic Regression aka Softmax Regression** to classify the {english} handwritten digits. The dataset used for this project is very famous MNIST dataset. This task shall serve as an refresher to anchor the notations and how mathematical expressions are mapped to python.
Key Words: Classification, Logistic Regression, MNIST Digits

2. MNIST Handwritten English Digit:

The MNIST dataset consists of handwritten digit images and is divided into 60,000 examples for the training set and 10,000 examples for testing. All the digital images have been size-normalized and centered in fixed size image of $\mathbf{28 \times 28}$ pixels.Each pixel of the image is represents by a value between **0** and **255** where:

1. **0** : represents black.
2. **255** : represents white.
3. anything in between is different shade of gray.

Since machine learning models cannot directly process raw image data, humans must extract relevant features and store them in a structured format, such as a CSV file.
For this task, we will extract pixel values from the image dataset and save them in a CSV file. A pre-processed CSV file has been provided for you.
Here are some examples of MNIST digits:

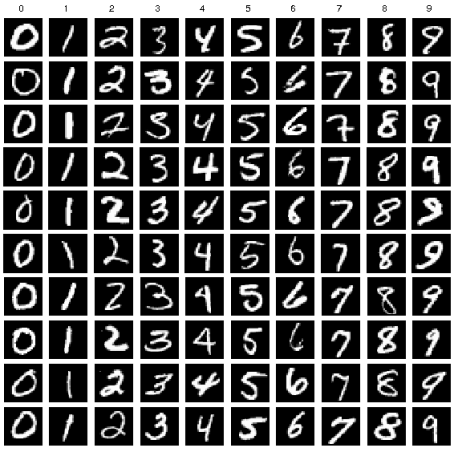


Figure 1: MNIST Dataset: Collection of Handwritten Digits.

3 Exercise - Building a Softmax Regression for MNIST Digit Classification:

We will begin by implementing a series of helper functions aligned with the Empirical Risk Minimization (ERM) framework. This includes defining the decision function, formulating the loss and cost functions, and implementing an optimization algorithm—specifically, gradient descent—to train the model effectively.

3.1 Decision Function:

In Softmax Regression, the decision function consists of two key components:

1. **Softmax Function:** Computes the probability distribution over classes for a given input.
2. **Prediction Function:** Assigns a class label based on the highest probability following the decision rule.

About a Softmax Function:

The softmax function is an extension of the sigmoid function for multivariate inputs, mapping a vector of real numbers to a probability distribution over K classes. Formally, the softmax function $\sigma : \mathbb{R}^K \to \mathbb{R}^K$ is defined as:

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}, \quad \text{for } i = 1, 2, \ldots, K,$$

where $z = [z_1, z_2, \ldots, z_K]^\top$ is the input vector. The function adheres to the following properties:

- **Boundedness**: Each output value lies in the interval $(0, 1)$, ensuring that the outputs represent valid probabilities.
- **Sum-to-one**: The outputs sum to 1, making them a proper probability distribution:

$$\sum_{i=1}^{K} \sigma(z)_i = 1.$$

- **Exponentially scaled inputs**: The exponential term e^{z_i} magnifies differences in the input values, accentuating the relative importance of larger inputs.
- **Monotonicity**: The function is monotonic in the sense that increasing z_i increases $\sigma(z)_i$ while decreasing the probabilities for other z_j $(j \neq i)$.
- **Smoothness**: The softmax function is continuously differentiable, making it suitable for optimization in machine learning models.

Application:

The softmax function is particularly useful in multi-class classification problems, where it is often used as the activation function in the output layer of neural networks. By converting logits (raw scores) into probabilities, it allows for a probabilistic interpretation of the predictions.

Prediction Function and Example:

For a three-class classification problem, let $z = [z_1, z_2, z_3]$. The softmax probabilities are:

$$\sigma(z)_1 = \frac{e^{z_1}}{e^{z_1} + e^{z_2} + e^{z_3}}, \quad \sigma(z)_2 = \frac{e^{z_2}}{e^{z_1} + e^{z_2} + e^{z_3}}, \quad \sigma(z)_3 = \frac{e^{z_3}}{e^{z_1} + e^{z_2} + e^{z_3}}.$$

These probabilities are used to assign class labels by selecting the class with the highest probability. The softmax function's ability to normalize outputs makes it indispensable in many machine learning tasks, particularly in deep learning frameworks.

3.1.1 Your Task:

Implement the **Softmax and Prediction functions** using either of the following approaches:

- **Build from scratch:** Develop the functions independently based on the mathematical formulation.
- **Use the provided functional template:** Complete the given function structure available in the starter code.

Regardless of the approach you choose, your implementation must pass the provided test case without any modifications to the test case itself.

" Do not Modify the Test Case."

1. Starter Code - Softmax Function:

Softmax Function.

```
import numpy as np
def softmax(z):
    """
    Compute the softmax probabilities for a given input matrix.
    Parameters:
    z (numpy.ndarray): Logits (raw scores) of shape (m, n), where
                    - m is the number of samples.
                    - n is the number of classes.
    Returns:
    numpy.ndarray: Softmax probability matrix of shape (m, n), where
                each row sums to 1 and represents the probability
                distribution over classes.
    Notes:
    - The input to softmax is typically computed as: z = XW + b.
    - Uses numerical stabilization by subtracting the max value per row.
    """
    # Your Code Here.
    return
```

Softmax Test Function.

```
# This test case checks that each row in the resulting softmax probabilities sums to 1, which is the
    fundamental property of softmax.
# Example test case
```

```
z_test = np.array([[2.0, 1.0, 0.1], [1.0, 1.0, 1.0]])
softmax_output = softmax(z_test)

# Verify if the sum of probabilities for each row is 1 using assert
row_sums = np.sum(softmax_output, axis=1)

# Assert that the sum of each row is 1
assert np.allclose(row_sums, 1), f"Test failed: Row sums are {row_sums}"

print("Softmax function passed the test case!")
```

2. Starter Code - Prediction Function:

Prediction Function.

```
def predict_softmax(X, W, b):
    """
    Predict the class labels for a set of samples using the trained softmax model.
    Parameters:
    X (numpy.ndarray): Feature matrix of shape (n, d), where n is the number of samples and d is the
        number of features.
    W (numpy.ndarray): Weight matrix of shape (d, c), where c is the number of classes.
    b (numpy.ndarray): Bias vector of shape (c,).
    Returns:
    numpy.ndarray: Predicted class labels of shape (n,), where each value is the index of the
        predicted class.
    """
    predicted_classes = # Your Code Here
    return predicted_classes
```

Prediction Test Case.

```
# The test function ensures that the predicted class labels have the same number of elements as the
    input samples, verifying that the model produces a valid output shape.
# Define test case
X_test = np.array([[0.2, 0.8], [0.5, 0.5], [0.9, 0.1]]) # Feature matrix (3 samples, 2 features)
W_test = np.array([[0.4, 0.2, 0.1], [0.3, 0.7, 0.5]]) # Weights (2 features, 3 classes)
b_test = np.array([0.1, 0.2, 0.3]) # Bias (3 classes)
# Expected Output:
# The function should return an array with class labels (0, 1, or 2)
y_pred_test = predict_softmax(X_test, W_test, b_test)
# Validate output shape
assert y_pred_test.shape == (3,), f"Test failed: Expected shape (3,), got {y_pred_test.shape}"
# Print the predicted labels
print("Predicted class labels:", y_pred_test)
```

3.2 Implementation of Loss and Cost Function:

1. About a Categorical Cross - Entropy Loss for Softmax Regression:

The Categorical Cross-Entropy Loss, often referred to as cross-entropy loss, is a generalization of the log-loss function used in multi-class classification problems. For a dataset with n samples, K classes,

true labels $y_i \in \{0,1\}^K$ (one-hot encoded), and predicted probabilities $\hat{\mathbf{y}}_i \in [0,1]^K$, the loss is defined as:

$$\text{Categorical Cross-Entropy Loss} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K} y_{ik}\log(\hat{y}_{ik}),$$

where:

- y_{ik} is 1 if the true class of sample i is k, and 0 otherwise.
- $\hat{y}_{ik}$ is the predicted probability for sample i belonging to class k.

Key Characteristics:

- Penalizes incorrect class predictions by focusing on the probability assigned to the true class.
- Encourages predicted probabilities $\hat{y}_{ik}$ close to the true label probabilities y_{ik}.
- Suitable for multi-class classification tasks where each sample belongs to exactly one class.

Where does the Categorical Cross-Entropy Loss come from? {Optional}

The following derivation of categorical cross-entropy loss is based on the Maximum Likelihood Estimation (MLE) framework, providing a theoretical foundation for its formulation.

Maximum Likelihood Estimation (MLE) Interpretation of Categorical Cross Entropy Loss:

1. The Probabilistic Model

For multi-class classification, softmax regression predicts the probability that a given input $\mathbf{x}$ belongs to class k:

$$P(y = k \mid \mathbf{x}, \mathbf{W}) = \hat{y}_{ik} = \frac{e^{\mathbf{w}_k^\top \mathbf{x}_i}}{\sum_{j=1}^{K} e^{\mathbf{w}_j^\top \mathbf{x}_i}},$$

where $\mathbf{W}$ is the parameter matrix and $k \in \{1, 2, \ldots, K\}$ are the classes.

2. Multinomial Distribution and PMF

The model assumes the labels follow a multinomial distribution. For each sample i, the probability of observing the one-hot label $\mathbf{y}_i$ is:

$$P(\mathbf{y}_i \mid \mathbf{x}_i, \mathbf{W}) = \prod_{k=1}^{K} \hat{y}_{ik}^{y_{ik}},$$

where $\hat{y}_{ik}$ is the predicted probability for class k, and y_{ik} is 1 if $\mathbf{y}_i$ belongs to class k, 0 otherwise.

3. Likelihood Function

Given n independent samples $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{n}$, the likelihood is the joint probability of the observed data:

$$\mathcal{L}(\mathbf{W}) = \prod_{i=1}^{n}\prod_{k=1}^{K} \hat{y}_{ik}^{y_{ik}}.$$

4. Log-Likelihood Function

Taking the natural logarithm of the likelihood:

$$\log \mathcal{L}(\mathbf{W}) = \sum_{i=1}^{n} \sum_{k=1}^{K} y_{ik} \log(\hat{y}_{ik}).$$

5. Negative Log-Likelihood (NLL)

To simplify optimization, we minimize the negative log-likelihood:

$$\text{NLL}(\mathbf{W}) = -\sum_{i=1}^{n} \sum_{k=1}^{K} y_{ik} \log(\hat{y}_{ik}).$$

6. Categorical Cross-Entropy Loss

To normalize the loss across all samples:

$$\text{Cross-Entropy Loss} = -\frac{1}{n} \sum_{i=1}^{n} \sum_{k=1}^{K} y_{ik} \log(\hat{y}_{ik}).$$

Key Notes:

- The cross-entropy loss arises naturally from MLE, as minimizing the negative log-likelihood corresponds to finding parameters $\mathbf{W}$ that maximize the probability of the observed data.
- It is based on the multinomial distribution, which generalizes the Bernoulli distribution to multiple classes.
- The loss encourages the predicted probabilities $\hat{y}_{ik}$ to align closely with the true labels y_{ik}.

3.2.1 Your Task:

Implement the **Loss and Cost functions** using either of the following approaches:

- **Build from scratch:** Develop the functions independently based on the mathematical formulation.
- **Use the provided functional template:** Complete the given function structure available in the starter code.

Regardless of the approach you choose, your implementation must pass the provided test case without any modifications to the test case itself.

```
" Do not Modify the Test Case."
```

1. Implementation of Loss Function:

Categorical Cross Entropy Loss:

```
def loss_softmax(y_pred, y):
    """
    Compute the cross-entropy loss for a single sample.
    Parameters:
    y_pred (numpy.ndarray): Predicted probabilities of shape (c,) for a single sample,
                            where c is the number of classes.
    y (numpy.ndarray): True labels (one-hot encoded) of shape (c,), where c is the number of classes.
    Returns:
    float: Cross-entropy loss for the given sample.
    """
    loss = # Your Code Here
    return loss
```

Test case for Loss Function:

```
import numpy as np
# This test case Compares loss for correct vs. incorrect predictions.
    # Expects low loss for correct predictions.
    # Expects high loss for incorrect predictions.
# Define correct predictions (low loss scenario)
y_true_correct = np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]]) # True one-hot labels
y_pred_correct = np.array([[0.9, 0.05, 0.05],
                           [0.1, 0.85, 0.05],
                           [0.05, 0.1, 0.85]]) # High confidence in the correct class

# Define incorrect predictions (high loss scenario)
y_pred_incorrect = np.array([[0.05, 0.05, 0.9], # Highly confident in the wrong class
                             [0.1, 0.05, 0.85],
                             [0.85, 0.1, 0.05]])

# Compute loss for both cases
loss_correct = loss_softmax(y_pred_correct, y_true_correct)
loss_incorrect = loss_softmax(y_pred_incorrect, y_true_correct)

# Validate that incorrect predictions lead to a higher loss
assert loss_correct < loss_incorrect, f"Test failed: Expected loss_correct < loss_incorrect, but got
    {loss_correct:.4f} >= {loss_incorrect:.4f}"

# Print results
print(f"Cross-Entropy Loss (Correct Predictions): {loss_correct:.4f}")
print(f"Cross-Entropy Loss (Incorrect Predictions): {loss_incorrect:.4f}")
```

2. Implementation of Cost Function:

Cost Function:

```
def cost_softmax(X, y, W, b):
    """
    Compute the average softmax regression cost (cross-entropy loss) over all samples.
    Parameters:
    X (numpy.ndarray): Feature matrix of shape (n, d), where n is the number of samples and d is the
        number of features.
    y (numpy.ndarray): True labels (one-hot encoded) of shape (n, c), where n is the number of
        samples and c is the number of classes.
```

```
    W (numpy.ndarray): Weight matrix of shape (d, c).
    b (numpy.ndarray): Bias vector of shape (c,).
    Returns:
    float: Average softmax cost (cross-entropy loss) over all samples.
    """
    total_loss = # Your Code Here
    # Return average loss
    return total_loss / n
```

Test case for Cost Function.

```
# The test case assures that the cost for the incorrect prediction should be higher than for the
    correct prediction, confirming that the cost function behaves as expected.
import numpy as np
# Example 1: Correct Prediction (Closer predictions)
X_correct = np.array([[1.0, 0.0], [0.0, 1.0]]) # Feature matrix for correct predictions
y_correct = np.array([[1, 0], [0, 1]]) # True labels (one-hot encoded, matching predictions)
W_correct = np.array([[5.0, -2.0], [-3.0, 5.0]]) # Weights for correct prediction
b_correct = np.array([0.1, 0.1]) # Bias for correct prediction
# Example 2: Incorrect Prediction (Far off predictions)
X_incorrect = np.array([[0.1, 0.9], [0.8, 0.2]]) # Feature matrix for incorrect predictions
y_incorrect = np.array([[1, 0], [0, 1]]) # True labels (one-hot encoded, incorrect predictions)
W_incorrect = np.array([[0.1, 2.0], [1.5, 0.3]]) # Weights for incorrect prediction
b_incorrect = np.array([0.5, 0.6]) # Bias for incorrect prediction
# Compute cost for correct predictions
cost_correct = cost_softmax(X_correct, y_correct, W_correct, b_correct)
# Compute cost for incorrect predictions
cost_incorrect = cost_softmax(X_incorrect, y_incorrect, W_incorrect, b_incorrect)
# Check if the cost for incorrect predictions is greater than for correct predictions
assert cost_incorrect > cost_correct, f"Test failed: Incorrect cost {cost_incorrect} is not greater
    than correct cost {cost_correct}"
# Print the costs for verification
print("Cost for correct prediction:", cost_correct)
print("Cost for incorrect prediction:", cost_incorrect)
print("Test passed!")
```

3.3 Implementation of Gradient Descent Algorithm:

Steps in Gradient Descent for Softmax Regression:

1. Initialize Parameters: Choose initial values for the parameters (e.g., weights $\mathbf{w}_c$ and bias b_c for each class c). These are often initialized randomly or with zeros.

2. Compute the Gradients: Calculate the gradient of the cost function (cross-entropy loss) with respect to each parameter. The gradient is a vector of partial derivatives, pointing in the direction of the steepest increase of the cost function.

The Gradients are:

The gradients of the multi-class cross-entropy loss are:

$$\frac{\partial \text{Categorical Cross Entropy}}{\partial \mathbf{w}_k} = -\frac{1}{n}\sum_{i=1}^{n}\left(y_{i,k} - \hat{y}_{i,k}\right)\mathbf{x}_i,$$

$$\frac{\partial \text{Categorical Cross Entropy}}{\partial b_k} = -\frac{1}{n}\sum_{i=1}^{n}\left(y_{i,k} - \hat{y}_{i,k}\right).$$

3. `Update the Gradients`: Update the parameters by moving in the direction of the negative gradient, with a step size proportional to the gradient (controlled by the learning rate α).

Update the Parameters:

The weights $\mathbf{w}_c$ and bias b_c for each class c are updated as:

$$\mathbf{w}_c \leftarrow \mathbf{w}_c - \alpha\frac{\partial \texttt{Categorical Loss}}{\partial \mathbf{w}_c}, \quad b_c \leftarrow b_c - \alpha\frac{\partial \texttt{Categorical Loss}}{\partial b_c}.$$

4. `Repeat`: Repeat steps 2 and 3 until convergence, i.e., when the change in the cost function becomes very small, or a maximum number of iterations is reached.

5. `Convergence Criteria`: We terminate the algorithm when one of the following conditions is met:

   - The cost function or empirical risk $\mathcal{L}(\mathbf{w}, \mathbf{b})$ does not change significantly (below a threshold) between iterations.
   - A predefined number of iterations is reached.

Gradient Descent Algorithm:

Algorithm 1 Gradient Descent for Softmax Regression

1: **Input:** Dataset $\mathcal{D}_n = \{(\mathbf{x}_i, y_i) \mid i = 1, 2, \ldots, n\}$, learning rate $\alpha > 0$, number of iterations T, number of classes C

2: **Initialize:** Parameters $\mathbf{w}_c \in \mathbb{R}^d$ and $b_c \in \mathbb{R}$ for each class $c = 1, 2, \ldots, C$

3: **for** t = 1 to T **do**

4: Compute predictions using softmax:

$$\hat{y}_{i,c} = \frac{e^{\mathbf{w}_c^\top \mathbf{x}_i + b_c}}{\sum_{j=1}^{C} e^{\mathbf{w}_j^\top \mathbf{x}_i + b_j}} \quad \forall i \in \{1, 2, \ldots, n\},\ c \in \{1, 2, \ldots, C\}$$

5: Compute gradients:

$$\frac{\partial \text{Log Loss}}{\partial \mathbf{w}_c} = -\frac{1}{n} \sum_{i=1}^{n} (y_{i,c} - \hat{y}_{i,c})\, \mathbf{x}_i$$

$$\frac{\partial \text{Log Loss}}{\partial b_c} = -\frac{1}{n} \sum_{i=1}^{n} (y_{i,c} - \hat{y}_{i,c})$$

6: Update parameters:

$$\mathbf{w}_c \leftarrow \mathbf{w}_c - \alpha \frac{\partial \text{Log Loss}}{\partial \mathbf{w}_c}$$

$$b_c \leftarrow b_c - \alpha \frac{\partial \text{Log Loss}}{\partial b_c}$$

7: **end for**

8: **Output:** Optimal parameters $\mathbf{w}_c^*$ and b_c^* for each class c =0

3.3.1 Your Task:

Implement the **Compute Gradients and Gradient Descent functions** using either of the following approaches:

- **Build from scratch:** Develop the functions independently based on the mathematical formulation.
- **Use the provided functional template:** Complete the given function structure available in the starter code.

Regardless of the approach you choose, your implementation must pass the provided test case without any modifications to the test case itself.

```
" Do not Modify the Test Case."
```

Implementation of Compute Gradients:

Computing Gradients against W and b

```
def compute_gradient_softmax(X, y, W, b):
    """
    Compute the gradients of the cost function with respect to weights and biases.
    Parameters:
    X (numpy.ndarray): Feature matrix of shape (n, d).
    y (numpy.ndarray): True labels (one-hot encoded) of shape (n, c).
    W (numpy.ndarray): Weight matrix of shape (d, c).
    b (numpy.ndarray): Bias vector of shape (c,).
    Returns:
    tuple: Gradients with respect to weights (d, c) and biases (c,).
    """
    grad_W = # Your Code Here # Gradient with respect to weights
    grad_b = #Your Code Here # Gradient with respect to biases
    return grad_W, grad_b
```

Test case for compute_gradient_softmax Function

```
import numpy as np
# Define a simple feature matrix and true labels
X_test = np.array([[0.2, 0.8], [0.5, 0.5], [0.9, 0.1]]) # Feature matrix (3 samples, 2 features)
y_test = np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]]) # True labels (one-hot encoded, 3 classes)
# Define weight matrix and bias vector
W_test = np.array([[0.4, 0.2, 0.1], [0.3, 0.7, 0.5]]) # Weights (2 features, 3 classes)
b_test = np.array([0.1, 0.2, 0.3]) # Bias (3 classes)
# Compute the gradients using the function
grad_W, grad_b = compute_gradient_softmax(X_test, y_test, W_test, b_test)
# Manually compute the predicted probabilities (using softmax function)
z_test = np.dot(X_test, W_test) + b_test
y_pred_test = softmax(z_test)
# Compute the manually computed gradients
grad_W_manual = np.dot(X_test.T, (y_pred_test - y_test)) / X_test.shape[0]
grad_b_manual = np.sum(y_pred_test - y_test, axis=0) / X_test.shape[0]
# Assert that the gradients computed by the function match the manually computed gradients
assert np.allclose(grad_W, grad_W_manual), f"Test failed: Gradients w.r.t. W are not equal.\
    nExpected: {grad_W_manual}\nGot: {grad_W}"
assert np.allclose(grad_b, grad_b_manual), f"Test failed: Gradients w.r.t. b are not equal.\
    nExpected: {grad_b_manual}\nGot: {grad_b}"
# Print the gradients for verification
print("Gradient w.r.t. W:", grad_W)
print("Gradient w.r.t. b:", grad_b)
print("Test passed!")
```

Implementations of Gradient Descent Algorithm:

Gradient Descent Algorithm

```
def gradient_descent_softmax(X, y, W, b, alpha, n_iter, show_cost=False):
    """
    Perform gradient descent to optimize the weights and biases.
    Parameters:
    X (numpy.ndarray): Feature matrix of shape (n, d).
    y (numpy.ndarray): True labels (one-hot encoded) of shape (n, c).
    W (numpy.ndarray): Weight matrix of shape (d, c).
    b (numpy.ndarray): Bias vector of shape (c,).
    alpha (float): Learning rate.
    n_iter (int): Number of iterations.
    show_cost (bool): Whether to display the cost at intervals.
    Returns:
    tuple: Optimized weights, biases, and cost history.
    """
    cost_history = []
    for i in range(n_iter):
        # Compute gradients
        grad_W, grad_b = compute_gradient_softmax(X, y, W, b)
        # Your Code Here
        """
        """"
    return W, b, cost_history
```

3.4 Preparing the Dataset:

Since machine learning models cannot directly process raw image data, humans must extract relevant features and store them in a structured format, such as a CSV file.
For this task, we will extract pixel values from the image dataset and save them in a CSV file. A pre-processed CSV file has been provided for you.
Question - 1:
Is extracting pixel values sufficient for effective feature extraction? Why or why not?

3.4.1 Your Task:

Implement the **load_and_prepare_mnist; and plot_sample_images** using either of the following approaches:

- **Build from scratch:** Develop the functions independently based on the mathematical formulation.
- **Use the provided functional template:** Complete the given function structure available in the starter code.

Whichever appraoch you pick you are expected to perform following:

- ✓ Loads MNIST CSV file into a Pandas DataFrame.
- ✓ Extracts labels (y) and image pixel values (X).
- ✓ Normalizes pixel values (optional, but helps convergence).
- ✓ Splits data into training and test sets (default: 80% train, 20% test).
- ✓ Plots one example image per digit (0-9) for visualization.

1. Starter - Code - load_and_prepare_mnist:

Load and Prepare Mnist Dataset:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
def load_and_prepare_mnist(csv_file, test_size=0.2, random_state=42):
    """
    Reads the MNIST CSV file, splits data into train/test sets, and plots one image per class.
    Arguments:
    csv_file (str) : Path to the CSV file containing MNIST data.
    test_size (float) : Proportion of the data to use as the test set (default: 0.2).
    random_state (int) : Random seed for reproducibility (default: 42).
    Returns:
    X_train, X_test, y_train, y_test : Split dataset.
    """
    # Load dataset
    df = pd.read_csv(csv_file)
    # Separate labels and features
    y = df.iloc[:, 0].values # First column is the label
```

```
    X = df.iloc[:, 1:].values # Remaining columns are pixel values
    # Normalize pixel values (optional but recommended)
    X = X / 255.0 # Scale values between 0 and 1
    # Split data into train and test sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size, random_state=
        random_state)
    # Plot one sample image per class
    plot_sample_images(X, y)
    return X_train, X_test, y_train, y_test
```

2. Starter - Code - plot_sample_images:

Plot Sample Image:

```
    def plot_sample_images(X, y):
    """
    Plots one sample image for each digit class (0-9).
    Arguments:
    X (np.ndarray): Feature matrix containing pixel values.
    y (np.ndarray): Labels corresponding to images.
    """
    plt.figure(figsize=(10, 4))
    unique_classes = np.unique(y) # Get unique class labels
    for i, digit in enumerate(unique_classes):
        index = np.where(y == digit)[0][0] # Find first occurrence of the class
        image = X[index].reshape(28, 28) # Reshape 1D array to 28x28
        plt.subplot(2, 5, i + 1)
        plt.imshow(image, cmap='gray')
        plt.title(f"Digit: {digit}")
        plt.axis('off')
    plt.tight_layout()
    plt.show()
```

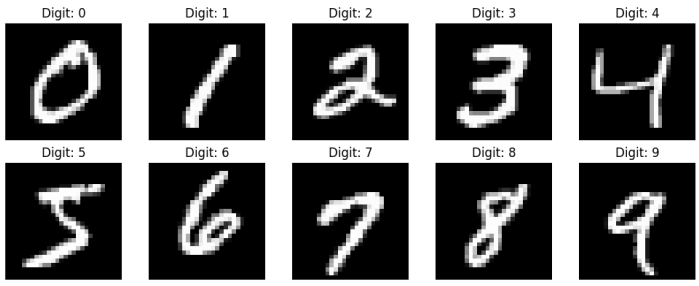


Figure 2: Sample Output

3.5 Training of the Model:

Next, we will integrate all the helper functions we've developed so far to train a softmax regression model for classifying MNIST images. Before we begin, let's run the following assert statements to ensure there are no shape mismatch errors

Shape Check before Training.

```
# Assert that X and y have matching lengths
assert len(X_train) == len(y_train), f"Error: X and y have different lengths! X={len(X_train)}, y={
    len(y_train)}"
print("Move forward: Dimension of Feture Matrix X and label vector y matched.")
```

If you have followed the provided code structure, the following code should work. However, if you have implemented your own solutions and all test cases have passed, you are encouraged to rewrite the code according to your design. Additionally, you are expected to generate a plot of Loss vs. Iteration, as shown below:

Training of the Softmax Regression Model.

```
from sklearn.preprocessing import OneHotEncoder
# Check if y_train is one-hot encoded
if len(y_train.shape) == 1:
    encoder = OneHotEncoder(sparse_output=False) # Use sparse_output=False for newer versions of
        sklearn
    y_train = encoder.fit_transform(y_train.reshape(-1, 1)) # One-hot encode labels
    y_test = encoder.transform(y_test.reshape(-1, 1)) # One-hot encode test labels
# Now y_train is one-hot encoded, and we can proceed to use it
d = X_train.shape[1] # Number of features (columns in X_train)
c = y_train.shape[1] # Number of classes (columns in y_train after one-hot encoding)
# Initialize weights with small random values and biases with zeros
W = np.random.randn(d, c) * 0.01 # Small random weights initialized
b = np.zeros(c) # Bias initialized to 0
# Set hyperparameters for gradient descent
alpha = 0.1 # Learning rate
n_iter = 1000 # Number of iterations to run gradient descent
# Train the model using gradient descent
W_opt, b_opt, cost_history = gradient_descent_softmax(X_train, y_train, W, b, alpha, n_iter,
    show_cost=True)
# Plot the cost history to visualize the convergence
plt.plot(cost_history)
plt.title('Cost Function vs. Iterations')
plt.xlabel('Iterations')
plt.ylabel('Cost')
plt.grid(True)
plt.show()
```

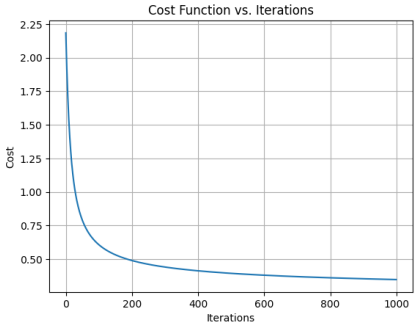


Figure 3: Expected Cost vs. Iterations Plot.

3.6 Evaluating Your Model Performance:

For evaluating the performance of your model, we will use a confusion matrix and compute precision and recall. The necessary code is provided below:

```
    import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, precision_score, recall_score, f1_score
# Evaluation Function
def evaluate_classification(y_true, y_pred):
    """
    Evaluate classification performance using confusion matrix, precision, recall, and F1-score.
    Parameters:
    y_true (numpy.ndarray): True labels
    y_pred (numpy.ndarray): Predicted labels
    Returns:
    tuple: Confusion matrix, precision, recall, F1 score
    """
    # Compute confusion matrix
    cm = confusion_matrix(y_true, y_pred)
    # Compute precision, recall, and F1-score
    precision = precision_score(y_true, y_pred, average='weighted')
    recall = recall_score(y_true, y_pred, average='weighted')
    f1 = f1_score(y_true, y_pred, average='weighted')
    return cm, precision, recall, f1
```

Putting it all together:

How well your model did?

```
# Predict on the test set
y_pred_test = predict_softmax(X_test, W_opt, b_opt)
# Evaluate accuracy
y_test_labels = np.argmax(y_test, axis=1) # True labels in numeric form
# Evaluate the model
cm, precision, recall, f1 = evaluate_classification(y_test_labels, y_pred_test)
# Print the evaluation metrics
print("\nConfusion Matrix:")
print(cm)
print(f"Precision: {precision:.2f}")
print(f"Recall: {recall:.2f}")
print(f"F1-Score: {f1:.2f}")
# Visualizing the Confusion Matrix
fig, ax = plt.subplots(figsize=(12, 12))
cax = ax.imshow(cm, cmap='Blues') # Use a color map for better visualization
# Dynamic number of classes
num_classes = cm.shape[0]
ax.set_xticks(range(num_classes))
ax.set_yticks(range(num_classes))
ax.set_xticklabels([f'Predicted {i}' for i in range(num_classes)])
ax.set_yticklabels([f'Actual {i}' for i in range(num_classes)])
# Add labels to each cell in the confusion matrix
for i in range(cm.shape[0]):
    for j in range(cm.shape[1]):
        ax.text(j, i, cm[i, j], ha='center', va='center', color='white' if cm[i, j] > np.max(cm) / 2
            else 'black')
```

```
# Add grid lines and axis labels
ax.grid(False)
plt.title('Confusion Matrix', fontsize=14)
plt.xlabel('Predicted Label', fontsize=12)
plt.ylabel('Actual Label', fontsize=12)
# Adjust layout
plt.tight_layout()
plt.colorbar(cax)
plt.show()
```

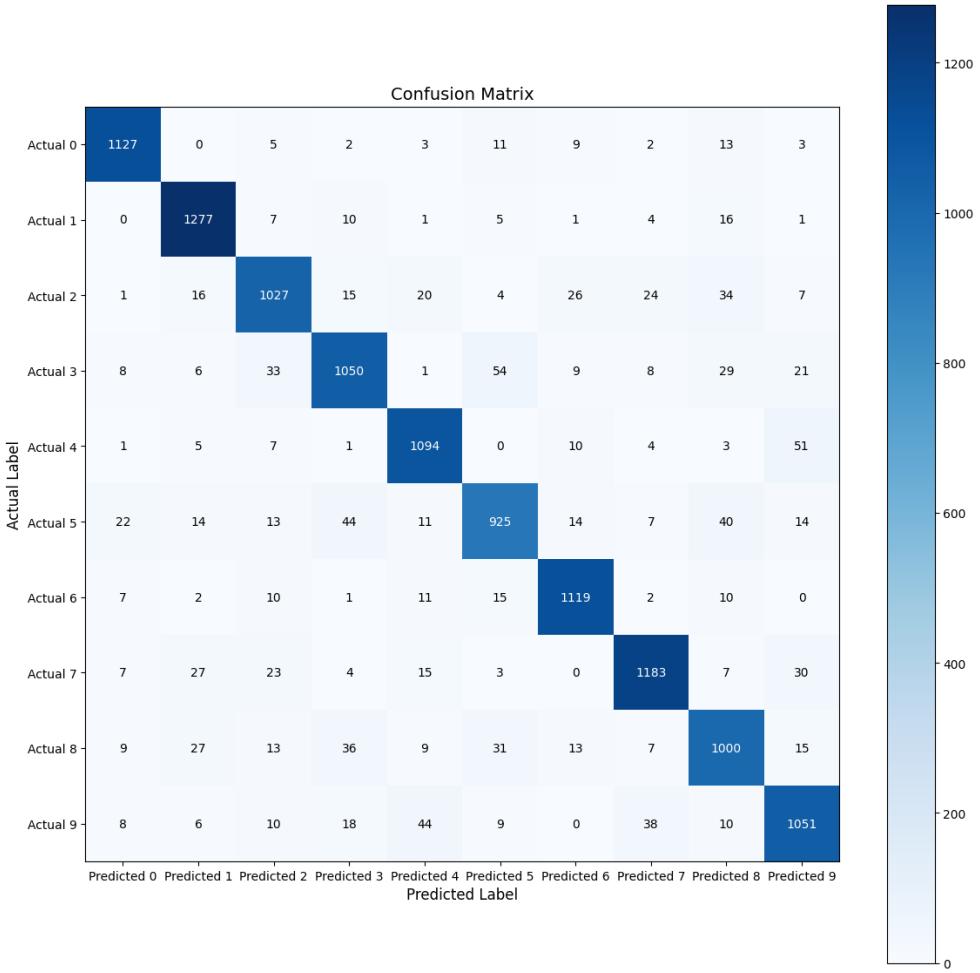


Figure 4: Expected Output - Confusion Matrix

4 Exercise - Linear Separability and Logistic Regression.

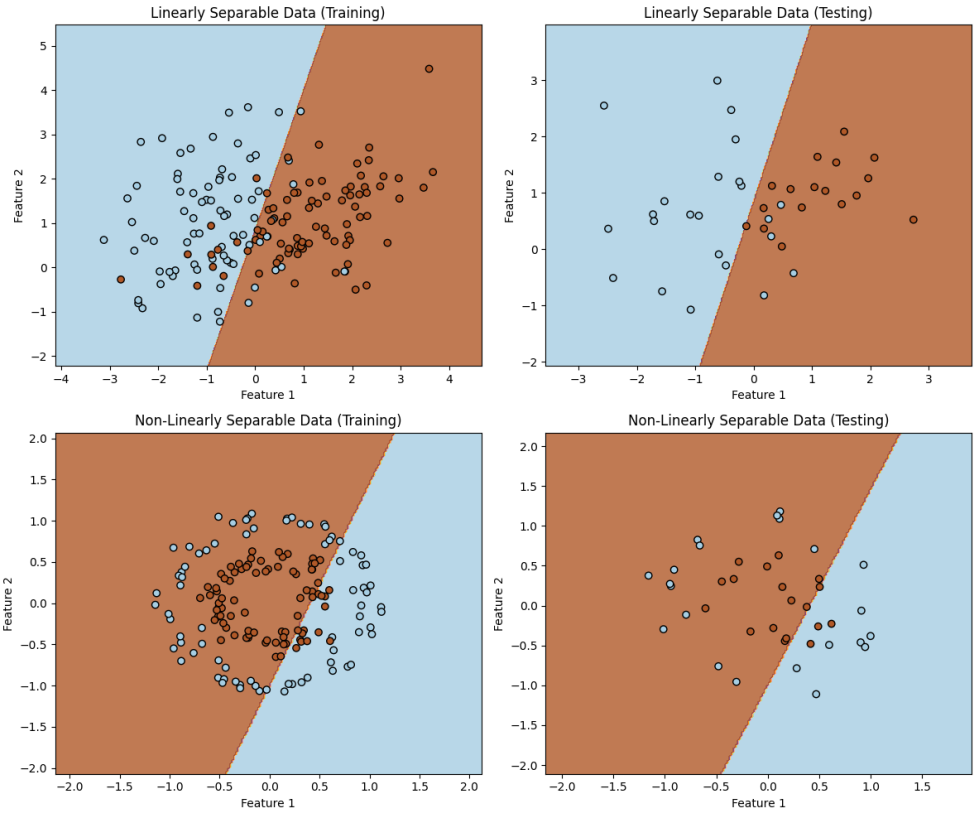
Re-implement the following code exactly as provided.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification, make_circles
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
# Set random seed for reproducibility
np.random.seed(42)
# Generate linearly separable dataset
X_linear_separable, y_linear_separable = make_classification(n_samples=200, n_features=2,
    n_informative=2,
                                                            n_redundant=0, n_clusters_per_class=1,
                                                                random_state=42)

# Split the data into training and testing sets
X_train_linear, X_test_linear, y_train_linear, y_test_linear = train_test_split(
    X_linear_separable, y_linear_separable, test_size=0.2, random_state=42
)
# Train logistic regression model on linearly separable data
logistic_model_linear_separable = LogisticRegression()
logistic_model_linear_separable.fit(X_train_linear, y_train_linear)
# Generate non-linearly separable dataset (circles)
X_non_linear_separable, y_non_linear_separable = make_circles(n_samples=200, noise=0.1, factor=0.5,
    random_state=42)
# Split the data into training and testing sets
X_train_non_linear, X_test_non_linear, y_train_non_linear, y_test_non_linear = train_test_split(
    X_non_linear_separable, y_non_linear_separable, test_size=0.2, random_state=42
)
# Train logistic regression model on non-linearly separable data
logistic_model_non_linear_separable = LogisticRegression()
logistic_model_non_linear_separable.fit(X_train_non_linear, y_train_non_linear)
# Plot decision boundaries for linearly and non-linearly separable data
def plot_decision_boundary(ax, model, X, y, title):
    h = .02 # step size in the mesh
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    ax.contourf(xx, yy, Z, alpha=0.8, cmap=plt.cm.Paired)
    ax.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k', cmap=plt.cm.Paired)
    ax.set_title(title)
    ax.set_xlabel('Feature 1')
    ax.set_ylabel('Feature 2')
# Create subplots
fig, axes = plt.subplots(2, 2, figsize=(12, 10))
# Plot decision boundary for linearly separable data (Training)
plot_decision_boundary(axes[0, 0], logistic_model_linear_separable, X_train_linear, y_train_linear,
    'Linearly Separable Data (Training)')
# Plot decision boundary for linearly separable data (Testing)
plot_decision_boundary(axes[0, 1], logistic_model_linear_separable, X_test_linear, y_test_linear, '
    Linearly Separable Data (Testing)')
# Plot decision boundary for non-linearly separable data (Training)
```

```
plot_decision_boundary(axes[1, 0], logistic_model_non_linear_separable, X_train_non_linear,
    y_train_non_linear, 'Non-Linearly Separable Data (Training)')
# Plot decision boundary for non-linearly separable data (Testing)
plot_decision_boundary(axes[1, 1], logistic_model_non_linear_separable, X_test_non_linear,
    y_test_non_linear, 'Non-Linearly Separable Data (Testing)')
plt.tight_layout()
# Save the plots as PNG files
plt.savefig('decision_boundaries.png')
plt.show()
```

If done correctly, you should generate the following plots: Once you have obtained the aforementioned



plots, answer the following questions:

- **Question - 2:** Provide an interpretation of the output based on your understanding.
- **Question - 3:** Describe any challenges you faced while implementing the code above.

Good Luck.