Department of Electronic and Telecommunication Engineering
University of Moratuwa

EN 3150: Pattern Recognition

Assignment 02: Learning from data and related challenges and classification.

Clarance L.G.S. - 200094R

September 29, 2023

1. Logistic regression weight update process

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
# Generate synthetic data
np.random.seed(0)
centers = [[-5, 0], [0, 1.5]]
X, y = make_blobs(n_samples=1000, centers=centers, random_state=40)
transformation = [[0.4, 0.2], [-0.4, 1.2]]
X = np.dot(X, transformation)
# Add a bias term to the feature matrix
X = np.c_[np.ones((X.shape[0], 1)), X]
# Initialize coefficients
W = np.zeros(X.shape[1])
# Define the logistic sigmoid function
def sigmoid(z):
    return 1 / (1 + np.exp(-z))
# Define the logistic loss ( binary cross - entropy ) function
def log_loss(y_true, y_pred):
    epsilon = 1e-15
    y_pred = np.clip(y_pred, epsilon, 1-epsilon) # Clip to avoid log (0)
    return -(y_true * np.log(y_pred) + (1 - y_true) * np.log(1 - y_pred))
```

- This code sets up the data, initializes logistic regression coefficients, defines necessary functions like 'sigmoid', log_loss' and prepares parameters for training.
- Now we perform gradient descent based weight update for the data.
- Here we use binary cross entropy as a loss function.
- Batch Gradient descent weight update is given below,

$$w_{(t+1)} \leftarrow w_{(t)} - \alpha \frac{1}{N} \left(sigm\left(w_{(t)}^{T} X\right) - y\right) X$$

```
# Gradient descent and Newton method parameters
learning_rate = 0.1
iterations = 10
loss_history_G = []
# Performing Gradient Descent
for i in range(iterations):
    # Compute the predicted probabilities
    y_pred = sigmoid(np.dot(X, W))

    # Compute the gradient of the loss function
    gradient = np.dot(X.T, (y_pred - y)) / len(y)
    # Update the weights using the gradient and learning rate
    W -= learning_rate * gradient
    # Calculate the binary cross-entropy loss and append it to the history
    loss = np.mean(log_loss(y, y_pred))
    loss_history_G.append(loss)
```

- To plot the loss with respect to number of iterations the following code is used.

```
# Plot the loss with respect to the number of iterations
iteration_numbers = np.arange(1, iterations + 1)
plt.plot(iteration_numbers, np.reshape(loss_history_G, (iterations, 1)), marker='o')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.title('Loss vs. Iterations')
plt.grid(True)
plt.show()
```

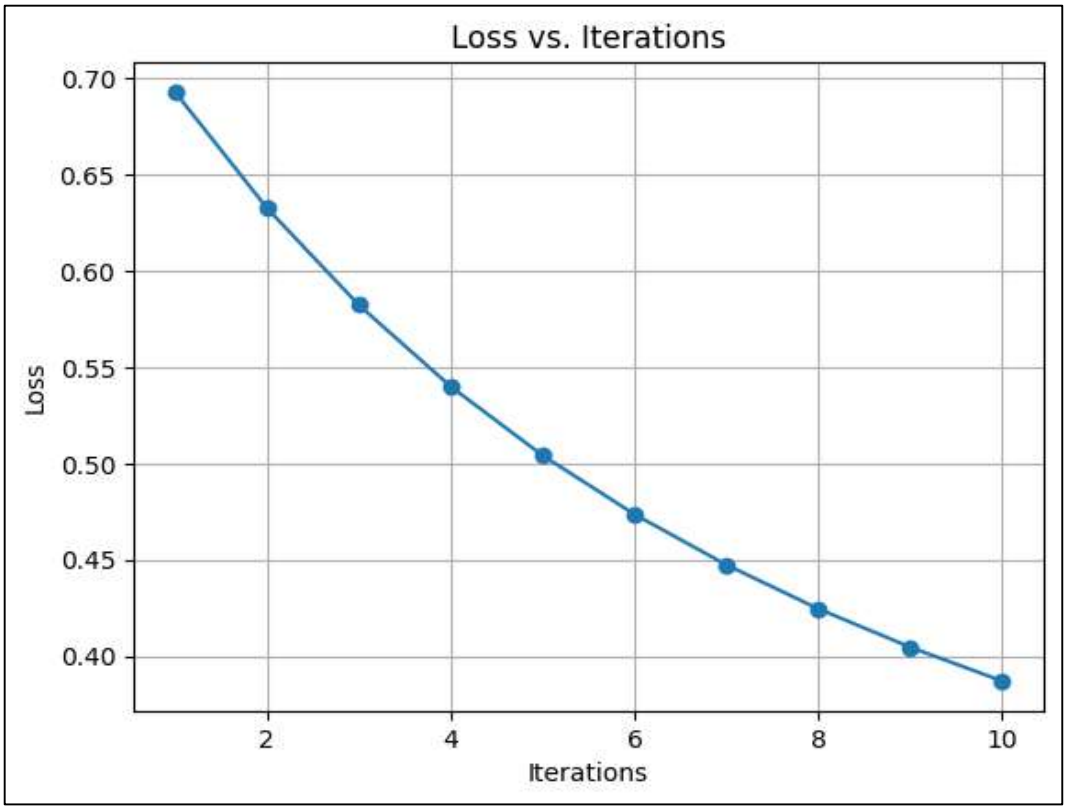

Figure 1: Loss w.r.t number of iterations for Gradient descent

- Now we perform Newton's method weight update for the data.
- Here we use binary cross entropy as a loss function.
- Batch Newton's method weight update is given below,

$$w_{(t+1)} \leftarrow w_{(t)} - \left(\frac{1}{N}X^T SX\right)^{-1} \left(\frac{1}{N}\left(sigma\left(w_{(t)}^T X\right) - y\right)X\right)$$

```
# Gradient descent and Newton method parameters
learning_rate = 0.1
iterations = 10
loss_history_N = []
# Performing Newton's Method
for i in range(iterations):
    # Compute the predicted probabilities
    y_pred = sigmoid(np.dot(X, W))
    # Calculate the diagonal matrix S
```

```
    S = np.diag(y_pred * (1 - y_pred))
    # Compute the gradient of the loss function
    gradient = np.dot(X.T, (y_pred - y)) / len(y)
    # Compute the Hessian matrix
    H = np.dot(np.dot(X.T, S), X) / len(y)
    # Update the weights using Newton's method
    W -= np.dot(np.linalg.inv(H), gradient)
    # Calculate the binary cross-entropy loss and append it to the history
    loss = np.mean(log_loss(y, y_pred))
    loss_history_N.append(loss)
```

- To plot the loss with respect to number of iterations the following code is used.

```
# Plot the loss with respect to the number of iterations
iteration_numbers = np.arange(1, iterations + 1)
plt.plot(iteration_numbers, np.reshape(loss_history_N, (iterations, 1)), marker='o', color='red')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.title('Loss vs. Iterations (Newton\'s Method)')
plt.grid(True)
plt.show()
```

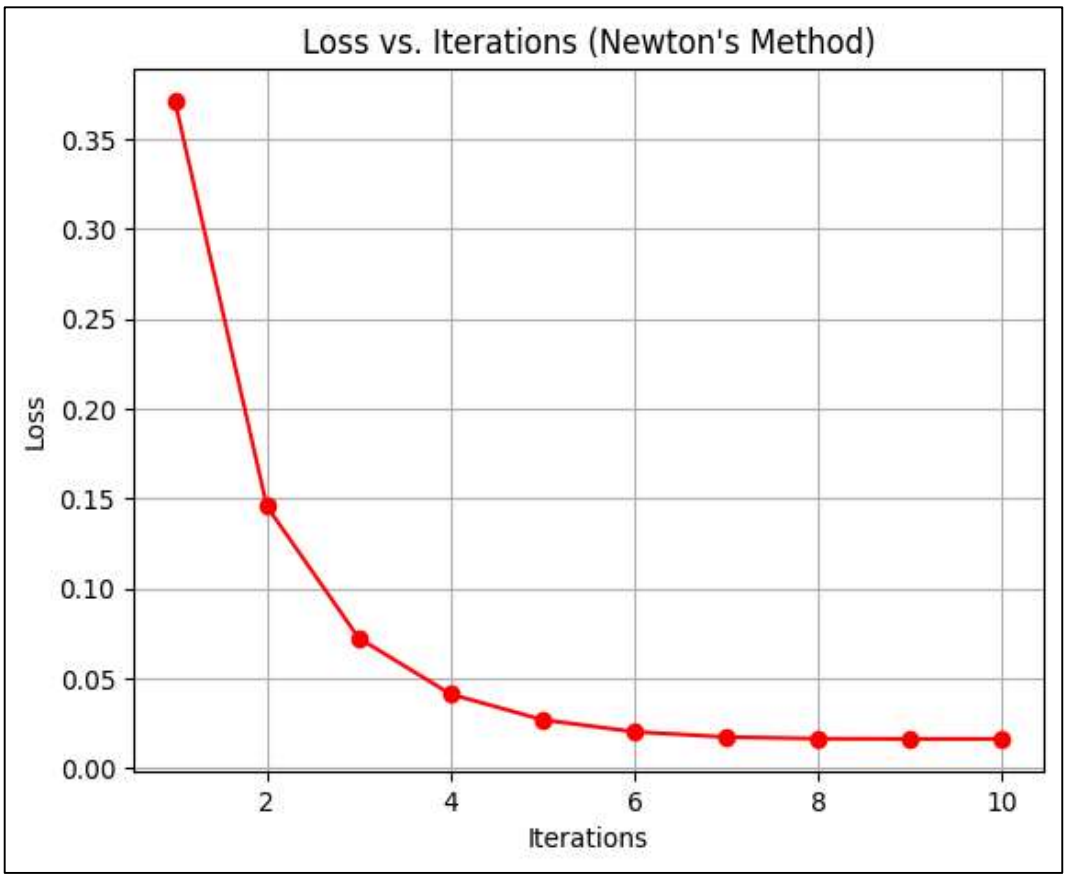

Figure 2: Loss w.r.t no of iterations for Newton's method

- To plot the loss with respect to number of iterations for both Gradient descent and Newton method's in a single plot the following code is used.

```
# Plot the loss with respect to the number of iterations for both methods
iteration_numbers = np.arange(1, iterations + 1)
plt.plot(iteration_numbers, np.reshape(loss_history_G, (iterations, 1)), marker='o', color='blue',
label='Gradient Descent')
plt.plot(iteration_numbers, np.reshape(loss_history_N, (iterations, 1)), marker='o', color='red',
label='Newton\'s Method')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.title('Loss vs. Iterations (Gradient Descent vs. Newton\'s Method)')
plt.grid(True)
plt.legend()
plt.show()
```

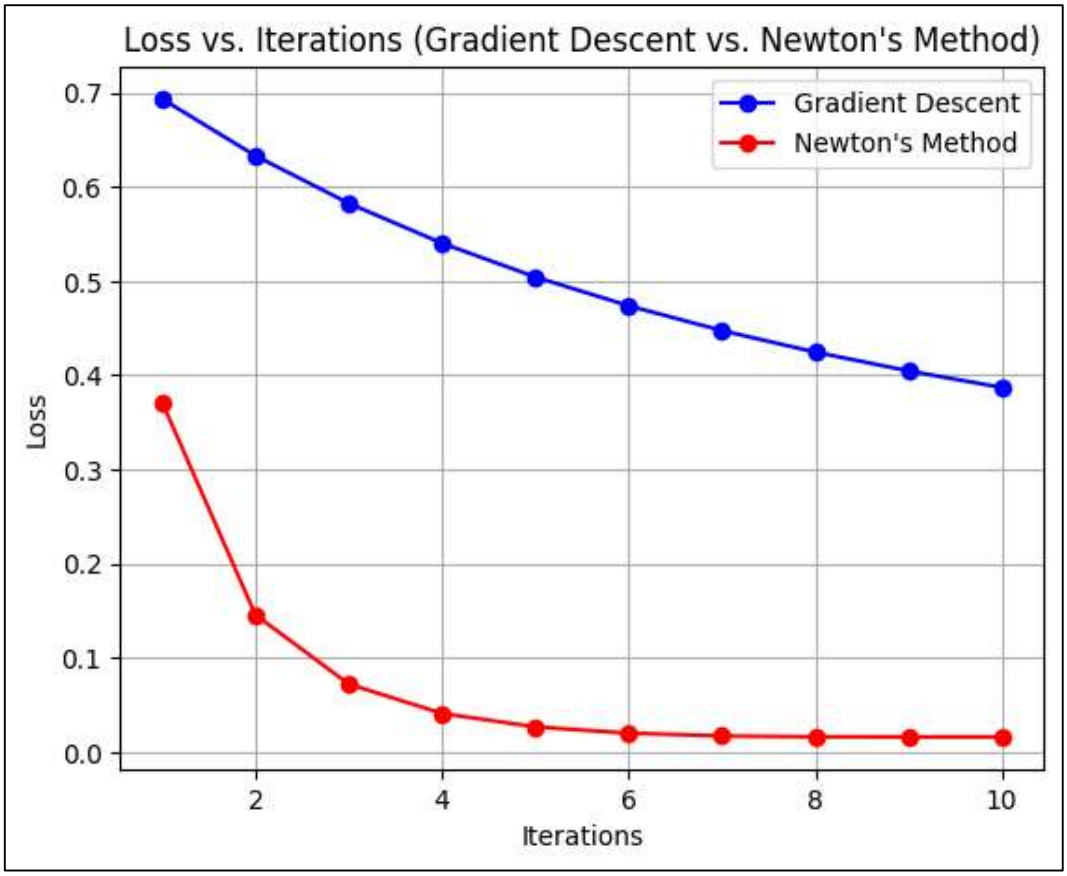

Figure 3: Loss w.r.t iterations for both

- As we can see from the plot that the loss of the gradient descent method is quite larger than Newton's method at the end of 10^{th} iteration.
- Newton's method gives the faster convergence time than gradient descent with learning rate is 0.01 for this given data.
- If we increase the no of iterations for gradient descent, we can reduce the loss further.

2.Perform grid search for hyper-parameter tuning

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_openml
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score
from sklearn.utils import check_random_state
# data loading
train_samples = 500
X, y = fetch_openml('mnist_784', version=1, return_X_y=True, as_frame=False)
random_state = check_random_state(0)
permutation = random_state.permutation(X.shape[0])
X = X[permutation]
y = y[permutation]
X = X.reshape((X.shape[0], -1))
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=train_samples, test_size=100)
```

- Here the code loads the MNIST dataset, performs some data preprocessing, and splits it into training and test sets.
- The purpose of 'X = X[permutation]' and 'y=y[permutation]'
    - It serves the purpose of shuffling the data and labels in the MNIST dataset based on a random permutation.
    - Here,
        - ✓ X is the feature matrix, which represents the images in the MNIST dataset. Each row of X corresponds to an image, and each column represents a pixel in that image.
        - ✓ y contains the labels for the images, indicating the digit (0 to 9) represented by each image.
        - ✓ permutation is a random permutation of indices that determines the new order of the data. It effectively shuffles the rows of X and y randomly.
    - We randomize the order of data instances (images) and their corresponding labels, to prevent any order-related biases during model training and evaluation.

```
# Create a pipeline with scaling and Lasso logistic regression
lasso_logistic_pipeline = Pipeline([
   ('scaler', StandardScaler()), # Standardize feature
   ('lasso_logistic', LogisticRegression(penalty='l1', solver='liblinear', multi_class='auto'))])
# Define the parameter grid for hyperparameter tuning
param_grid = {'lasso_logistic__C': np.logspace(-2, 2, 9)}
```

- Here we use lasso logistic regression for image classification and create a pipeline that includes the scaling and the Lasso logistic regression estimator.
- Also, we define the parameter grid for hyperparameter tuning.

```
# Create GridSearchCV to find the optimal value of C
grid_search = GridSearchCV(lasso_logistic_pipeline, param_grid, cv=5, n_jobs=-1)
grid_search.fit(X_train, y_train)
# Get the best value of C
best_C = grid_search.best_params_['lasso_logistic__C']
print('Best C:', best_C)
```

- Here we perform the grid search over the range (np.logspace(-2, -2, 9) and we find the optimal value of hyperparameter C.
  - **Best C: 0.31622776601683794**
- To plot the classification accuracy with respect to hyperparameter C the following code is used.

```
# Extract the hyperparameter values and corresponding mean test scores
C_values = np.logspace(-2, 2, 9)
mean_test_scores = grid_search.cv_results_['mean_test_score']
# Plot the mean test score against C
plt.semilogx(C_values, mean_test_scores, marker='o')
plt.xlabel('C')
plt.ylabel('Accuracy')
plt.title('Mean Test Score vs. C')
plt.grid(True)
plt.show()
```

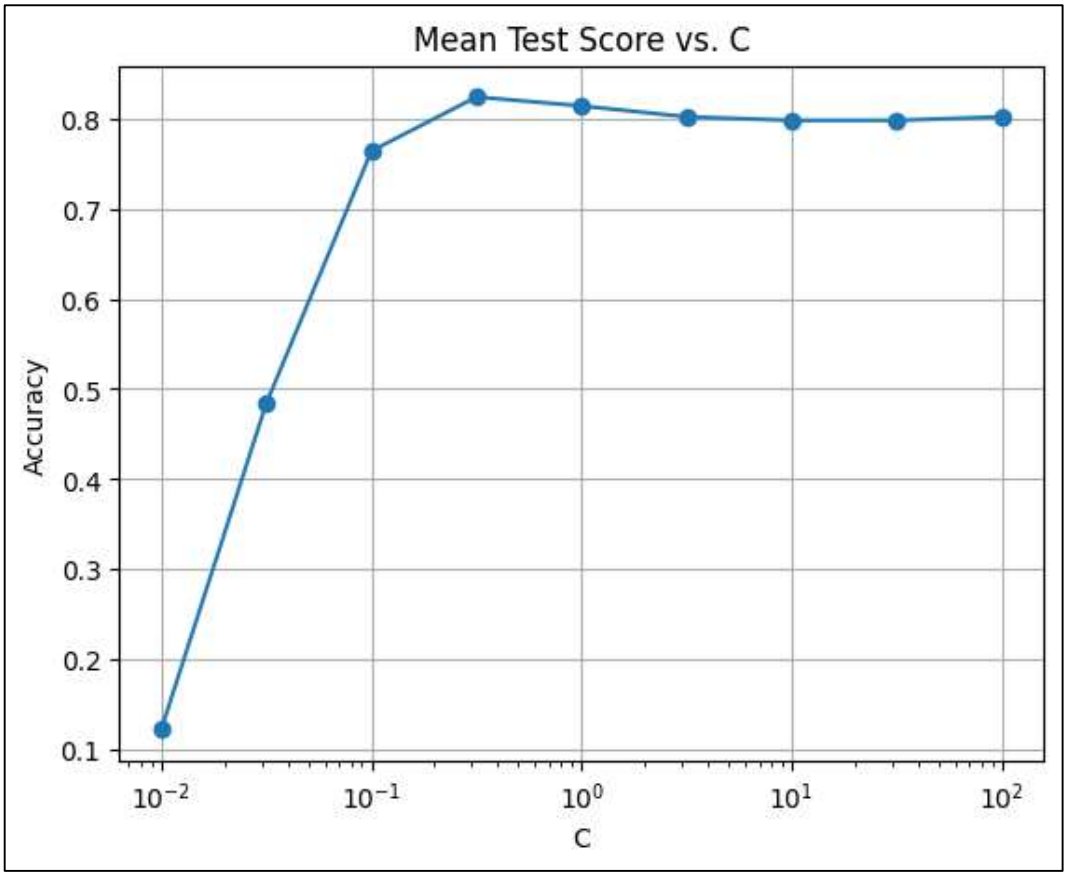

Figure 4: Classification accuracy w.r.t Hyperparameter C

- The plot shows how the model's accuracy changes as 'C' varies.
- From the plot we can clearly see that the optimal 'C' value that maximizes accuracy is around 0.3.

```
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score, recall_score,
f1_score
# Make predictions on the test set
y_pred = grid_search.predict(X_test)
# Calculate the accuracy
accuracy = accuracy_score(y_test, y_pred)
# Calculate confusion matrix
confusion_matrix = confusion_matrix(y_test, y_pred)
# Calculate precision
precision = precision_score(y_test, y_pred, average='macro')
# Calculate recall
recall = recall_score(y_test, y_pred, average='macro')
# Calculate F1 score
f1 = f1_score(y_test, y_pred, average='macro')
# Print the metrics
print('Accuracy:', accuracy)
print('Confusion Matrix:\n', confusion_matrix)
print('Precision:', precision)
print('Recall:', recall)
print('F1 Score:', f1)
```

- The above code is used to calculate confusion matrix, precision, recall, and F1-score.
  - Accuracy: **0.85**
  - Confusion Matrix:
    [[6 0 0 0 0 1 0 0 1 0]
    [0 13 0 0 0 0 0 0 0 0]
    [0 0 6 0 0 0 0 0 0 0]
    [0 0 0 8 0 0 0 0 0 0]
    [0 1 0 0 8 0 0 0 0 2]
    [0 0 0 0 1 3 0 0 0 0]
    [0 0 0 0 1 0 8 0 0 0]
    [0 0 0 0 1 0 0 11 0 2]
    [0 0 0 0 1 0 0 0 11 0]
    [0 1 0 1 0 1 0 1 0 11]]
  - Precision: **0.8588888888888888**
  - Recall: **0.85518759018759**
  - F1 Score: **0.8526539913624311**

- To get the confusion matrix we use alternative method too.

```
# Plot the confusion matrix
import scikitplot as skplt
skplt.metrics.plot_confusion_matrix(y_test, y_pred, normalize=False)
plt.title('Confusion Matrix')
plt.show()
```

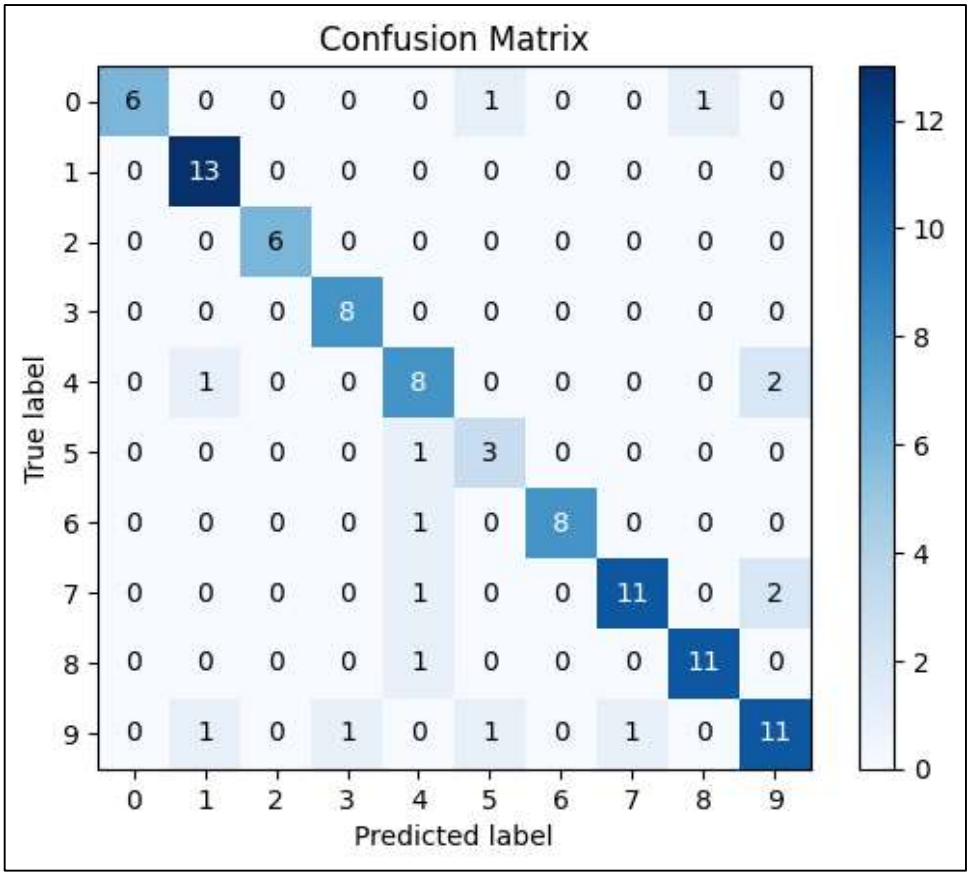

Figure 5: Confusion Matrix

- Accuracy: The model achieved an accuracy of 0.85, which indicates that it correctly classified approximately 85% of the test samples. This suggests that the model is reasonably effective in distinguishing between the different digits in the MNIST dataset.
- Confusion Matrix: The confusion matrix provides detailed information about the model's performance for each digit class. Key observations:
    - Diagonal elements (from top-left to bottom-right) represent the number of correctly classified samples for each digit.
    - Off-diagonal elements represent misclassifications.
- Precision: The precision is approximately 0.859. Precision measures the ratio of correctly predicted positive instances to the total predicted positive instances. In the context of multi-class classification, we calculate the precision for each class independently and then average them. It indicates the model's ability to avoid false positives.
- Recall: The recall is approximately 0.855. Recall measures the ratio of correctly predicted positive instances to the total actual positive instances. Like precision, we calculate the recall for each class independently and then average them. It indicates the model's ability to capture all relevant instances of each class.
- F1 Score: The F1 score is approximately 0.853. The F1 score is the harmonic mean of precision and recall. It provides a balanced measure of a model's performance, considering both false positives and false negatives.

3.Logistic regression

$$x_1 = No\ of\ hours\ studied$$

$$x_2 = GPA$$

$$y = student\ received\ A^+$$

W_0	-6
W_1	0.05
W_2	1

Regression model:

$$\Pr(y) = sigmoid(w_0 + w_1x_1 + w_2x_2)$$
$$= \frac{1}{1 + e^{-(w_0+w_1x_1+w_2x_2)}}$$

- $x_1 = 40$
  $x_2 = 3.5$
  $\Pr(y) = \frac{1}{1 + e^{-(-6+0.05*40+\ *3.5)}} = \mathbf{37.5\%}$

- $\Pr(y) = 0.5$
  $x_2 = 3.5$
  $0.5 = \frac{1}{1 + e^{-(-6+0.05*x_1+3.5)}}$

  $x_1 = \mathbf{50\ \boldsymbol{hrs}}$

References

- https://scikit-learn.org/stable/auto_examples/compose/plot_digits_pipe.html
- https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

GitHub Repository

- https://github.com/ClaranceLGS/EN3150-Pattern-Recognition/tree/main/Assignment%202