Department of Electronic & Telecommunication Engineering
University of Moratuwa

EN3150 - PATTERN RECOGNITION

LEARNING FROM DATA AND RELATED CHALLENGES AND CLASSIFICATION

MANIMOHAN T. *200377M*

This report is submitted as Assignment - 02 of module EN3150
2nd October 2023

ABSTRACT

This report describe about the Assignment - 2 EN3150 - PATTERN RECOGNITION module, which conducted by Dr. Sampath Perera

In this assignment, we learned about logistic regression, a powerful machine-learning technique. We first generated synthetic data and used it to practice gradient descent and Newton's method for updating model weights. We also explored hyperparameter tuning and evaluated the performance of a logistic regression model on real-world data from the MNIST dataset. We learned how to measure the model's accuracy and how to analyze its confusion matrix, precision, recall, and F1-score to understand how well it works. Finally, we applied logistic regression to predict a student's chances of getting an A+ based on their study hours and GPA. This assignment helped us gain practical insights into logistic regression and its real-world applications.

1 LOGISTIC REGRESSION WEIGHT UPDATE PROCESS

1.1 Use the code given in listing 1 to generate data.

Listing 1.1 — Generate data

```python
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))

# Gradient descent and Newton method parameters
learning_rate = 0.1
iterations = 10
loss_history = []
```

The provided code is used for data generation, configuration, coefficient initialization, and defining essential functions required for the Logistic Regression weight update process. These functions include Sigmoid and log loss calculations.

1.2 Initializing weights as zeros, perform gradient descent-based weight update for the given data. Here, use binary cross-entropy as a loss function. Further, use a learning rate as $\alpha = 0.1$ and the number of iterations as $t = 10$. Batch Gradient Descent weight update is given below:

$$w_{(t+1)} \leftarrow w_{(t)} - \alpha \cdot \frac{1}{N}\left(1_N^T \text{diag}\left(\text{sigmoid}(w_{(t)}^T x_i) - y_i\right) X\right)^T$$

Here, X is the data matrix of dimensions $N \times (D+1)$.Here N is the total number of data samples, and D is the number of features. Now, X is given by:

$$X = \begin{bmatrix} 1 & x_{1,1} & x_{2,1} & \dots & x_{D,1} \\ 1 & x_{1,2} & x_{2,2} & \dots & x_{D,2} \\ \vdots & \vdots & \ddots & \vdots & \\ 1 & x_{1,i} & x_{2,i} & \dots & x_{D,i} \\ \vdots & \vdots & \dots & \vdots & \\ 1 & x_{1,N} & x_{2,N} & \dots & x_{D,N} \end{bmatrix} = \begin{bmatrix} x_1^T \\ x_2^T \\ \vdots \\ x_i^T \\ \vdots \\ x_N^T \end{bmatrix}.$$

Listing 1.2 — Batch Gradient Descent weight update

```python
# Perform batch gradient descent
for epoch in range(iterations):
    # Calculate predictions using the current weights
    predictions = sigmoid(np.dot(X, W))

    # Compute the gradient of the loss function
    gradient = np.dot(X.T, (predictions - y)) / len(y)

    # Update the weights using the gradient
    W -= learning_rate * gradient

    # Calculate the loss and store it in the history
    loss = np.mean(log_loss(y, predictions))
    loss_history.append(loss)
```

1.3 Plot the loss with respect to number of iterations.

Listing 1.3 — Loss with respect to number of iterations

```python

# Plot the loss history
import matplotlib.pyplot as plt
plt.plot(range(1, iterations + 1), np.reshape(loss_history,
(iterations, 1)), marker='o')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.title('Loss vs. Iterations')
plt.show()
```

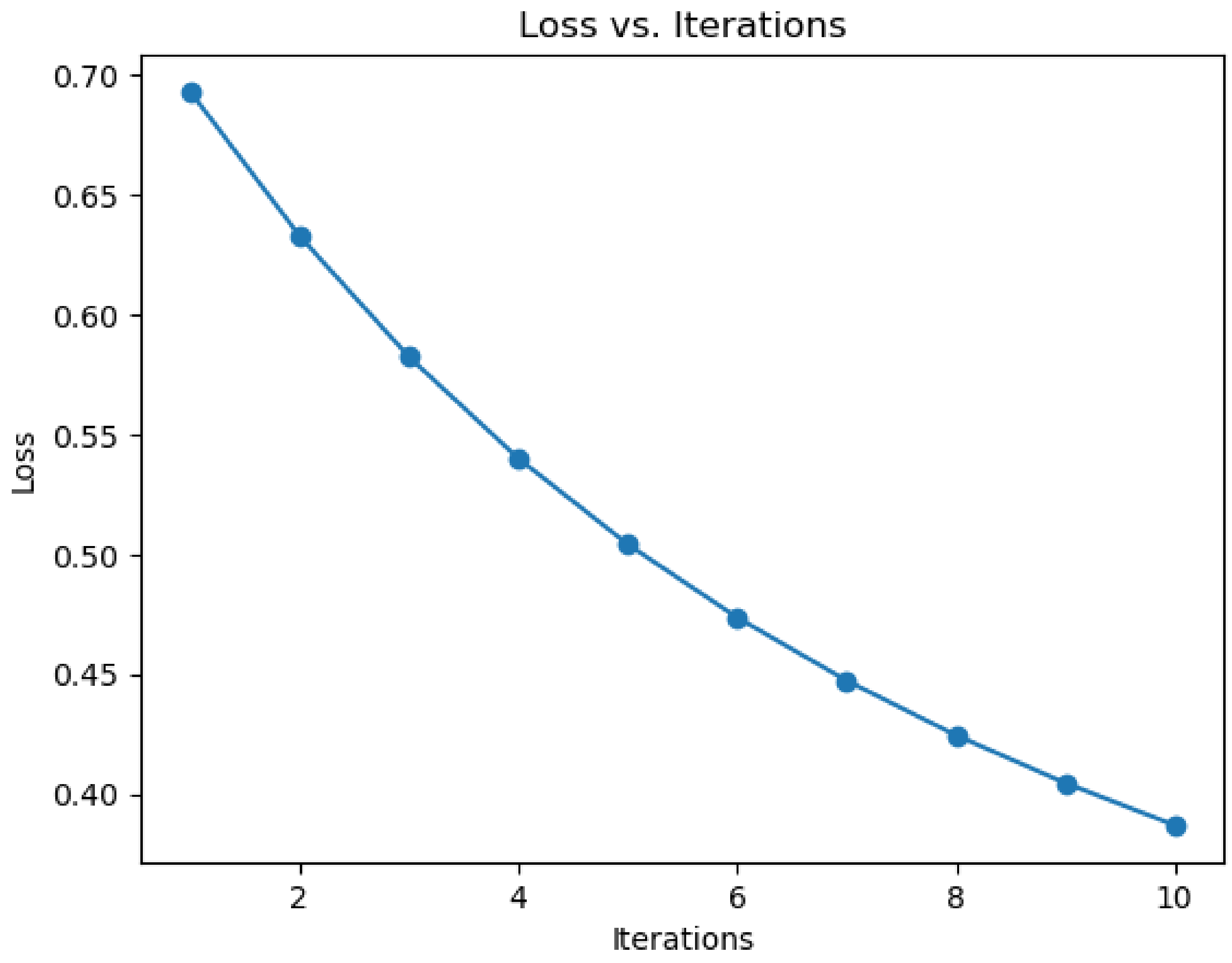


Figure 1 — Loss with respect to number of iterations

1.4 Initializing weights as zeros, perform Newton's method weight update for the given data. Here, use binary cross-entropy as a loss function. Further, set the number of iterations as $t = 10$. Batch Newton's method weight update is given below:

$$w_{(t+1)} \leftarrow w_{(t)} - \left(\frac{1}{N} X^T S X\right)^{-1} \left(\frac{1}{N} \left(1_N^T \text{diag}\left(\text{sigmoid}(w_{(t)}^T x_i) - y_i\right) X\right)^T\right)$$

Where:

$$S = \text{diag}(s_1, s_2, \ldots, s_N),$$
$$s_i = \left(\text{sigmoid}(w_{(t)}^T x_i) - y_i\right)\left(1 - \text{sigmoid}(w_{(t)}^T x_i) - y_i\right).$$

Listing 1.4 — Batch Newton's method weight update

```
# Newton's method parameters
W_newton = np.zeros(X.shape[1])
learning_rate = 0.1
iterations = 10
loss_history_n = []

# Perform Newton's method
for _ in range(iterations):
    # Calculate predicted probabilities
    y_pred = sigmoid(np.dot(X, W_newton))

    # Calculate the diagonal matrix S
    S = np.diag(y_pred * (1 - y_pred))

    # Calculate the gradient
    gradient = np.dot(X.T, y_pred - y) / len(y)

    # Calculate the Hessian matrix
    Hessian = np.dot(np.dot(X.T, S), X) / len(y)

    # Update the weights using Newton's method formula
    W_newton -= np.dot(np.linalg.inv(Hessian), gradient)

    # Calculate and store the loss
    loss = np.mean(log_loss(y, y_pred))
    loss_history_n.append(loss)
```

1.5 Plot the loss with respect to number of iterations.

Listing 1.5 — Loss with respect to number of iterations(Newton's Method)

```
# Plot the loss history for Newton's method
plt.plot(range(1, iterations + 1), 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.show()
```

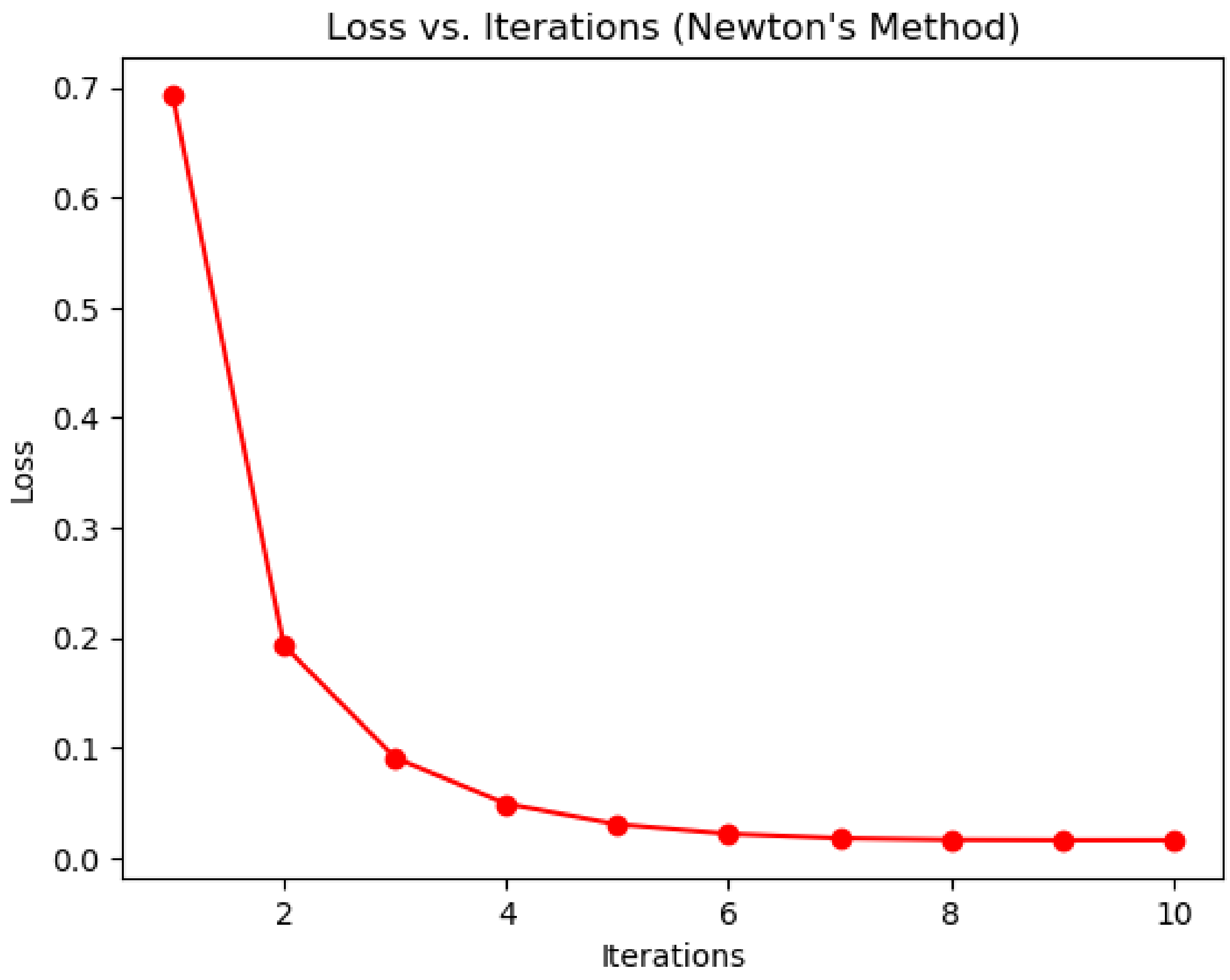


Figure 2 — Loss with respect to number of iterations(Newton's Method)

1.6 Plot the loss with respect to number of iterations for both Gradient descent and Newton method's in a single plot. Comment on your results

We will consider the comparison:

Listing 1.6 — Loss with respect to number of iterations for both Gradient descent and Newton method's in a single plot

```python
import matplotlib.pyplot as plt

# Create a figure with a specified size
plt.figure(figsize=(12, 6))

# Plot the loss history for Gradient Descent
plt.plot(range(1, iterations + 1), np.reshape(loss_history,
(iterations, 1)), label='Gradient␣Descent', marker='o')

# Plot the loss history for Newton's Method
plt.plot(range(1, iterations + 1), np.reshape(loss_history_n,
(iterations, 1)), label="Newton's␣Method", marker='x')
```

```
# Add labels, legend, and title
plt.xlabel('Number␣of␣Iterations')
plt.ylabel('Loss')
plt.legend()
plt.title('Loss␣vs.␣Number␣of␣Iterations')

# Show the plot
plt.show()
```

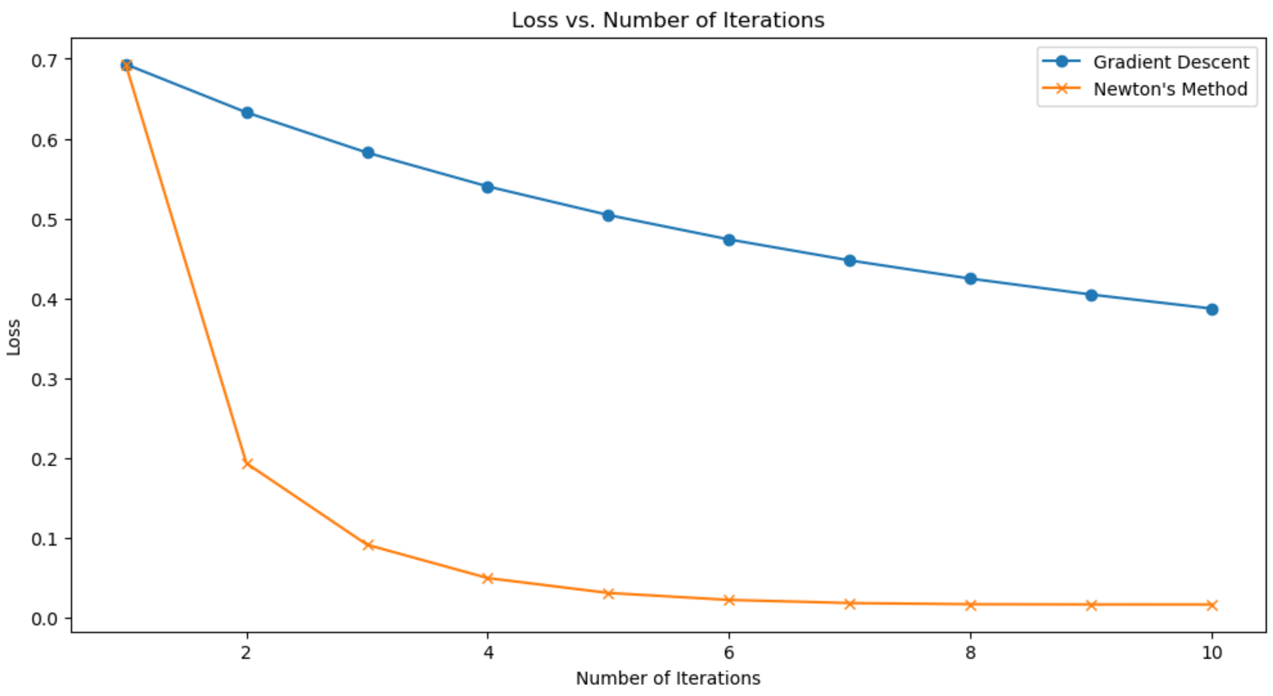


Figure 3 — Loss with respect to number of iterations for both Gradient descent and Newton method's in a single plot

The plot above clearly illustrates the contrasting convergence rates between the mentioned methodologies.

* Newton's Method demonstrates an exponential decay pattern in its loss function, while the Gradient Descent method exhibits a nearly linear decay.
* Newton's Method achieves faster convergence with respect to the number of iterations. After just 3 iterations, its loss value drops below 0.1. In contrast, batch gradient descent does not reach that level even after 10 iterations.
* The Gradient Descent method requires a greater number of iterations to achieve a reduction in loss.

Overall, the plot highlights the superiority of Newton's Method in terms of convergence speed and the ability to reduce the loss rapidly, especially compared to batch gradient descent.

2.1 Use the code given in listing 2 to load data..

Listing 2.1 — Load data

```
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)
```

2.2 Explain the purpose of ”X = X[permutation]” and ” y = y[permutation]”

In the provided code, the rows of both X and y are rearranged based on the permutation array. The purpose of these actions can be summarized as follows:

* The ”permutation” array consists of indices that represent a random reshuffling of the dataset.
* $X = X[permutation]$ is used to shuffle the rows of the feature matrix X, aligning them with the random order specified by the indices in the permutation array. This step ensures that the data is arranged in a random order, which is crucial for training and testing machine learning models, helping to mitigate potential biases.
* Similarly, $y = y[permutation]$ shuffles the corresponding target labels y in the same random order as X, ensuring that the labels remain correctly associated with the respective data points.

In summary, the purpose of these operations is to prevent any biases stemming from the original order of data instances by introducing randomness through the permutation array. This randomization is particularly important in machine learning to promote fair and unbiased model training and evaluation.

2.3 Use lasso logistic regression for image classification as "LogisticRegression(penalty='l1', solver='liblinear', multi_class='auto')". Next, create a pipeline that includes the scaling, the Lasso logistic regression estimator, and a parameter grid for hyperparameter tuning (C value).[Hint refer url]

Listing 2.2 — Use lasso logistic regression

```python
import numpy as np
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

# Create a pipeline with scaling, Lasso logistic regression,
and hyperparameter tuning
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('lasso_logistic', LogisticRegression(penalty='l1', solver='liblinear',
    multi_class='auto'))])
```

2.4 Use GridSearchCV to perform a grid search over the range (e.g., np.logspace(-2, 2, 9)) of to find optimal value of hyperparameter C [Hint refer url]

Listing 2.3 — Find optimal value of hyperparameter C

```python
# Define the hyperparameter grid for C values
param_grid = {
    'lasso_logistic__C': np.logspace(-2, 2, 9)}
# Perform grid search with cross-validation
grid_search = GridSearchCV(pipeline, param_grid, cv=5, scoring='accuracy')
# Fit the grid search to your training data
grid_search.fit(X_train, y_train)

# Get the best hyperparameters
best_params = grid_search.best_params_['lasso_logistic__C']
print(best_params)
```

```
0.31622776601683794
```

In the given process, a grid search is conducted over the range of hyperparameters, specifically using the values obtained from np.logspace(-2, -2, 9). After performing the search, the optimal value for the hyperparameter C is determined to be approximately C = 0.31622776601683794.

2.5 Plot the classification accuracy with respect to hyperparameter C. Comment on your results

Listing 2.4 — Plot the classification accuracy with respect to hyperparameter C

```python
import matplotlib.pyplot as plt

# Get the cross-validation results
results = grid_search.cv_results_
mean_test_scores = results['mean_test_score']

# Create a figure with a specified size
plt.figure(figsize=(10, 6))

# Plot accuracy vs. C with a logarithmic scale on the x-axis
plt.semilogx(param_grid['lasso_logistic__C'], mean_test_scores,
marker='o')
plt.title('Classification␣Accuracy␣vs.␣Hyperparameter␣C')
plt.xlabel('C␣(Inverse␣of␣Regularization␣Strength)')
plt.ylabel('Mean␣Test␣Accuracy')
plt.grid(True)
# Show the plot
plt.show()
```

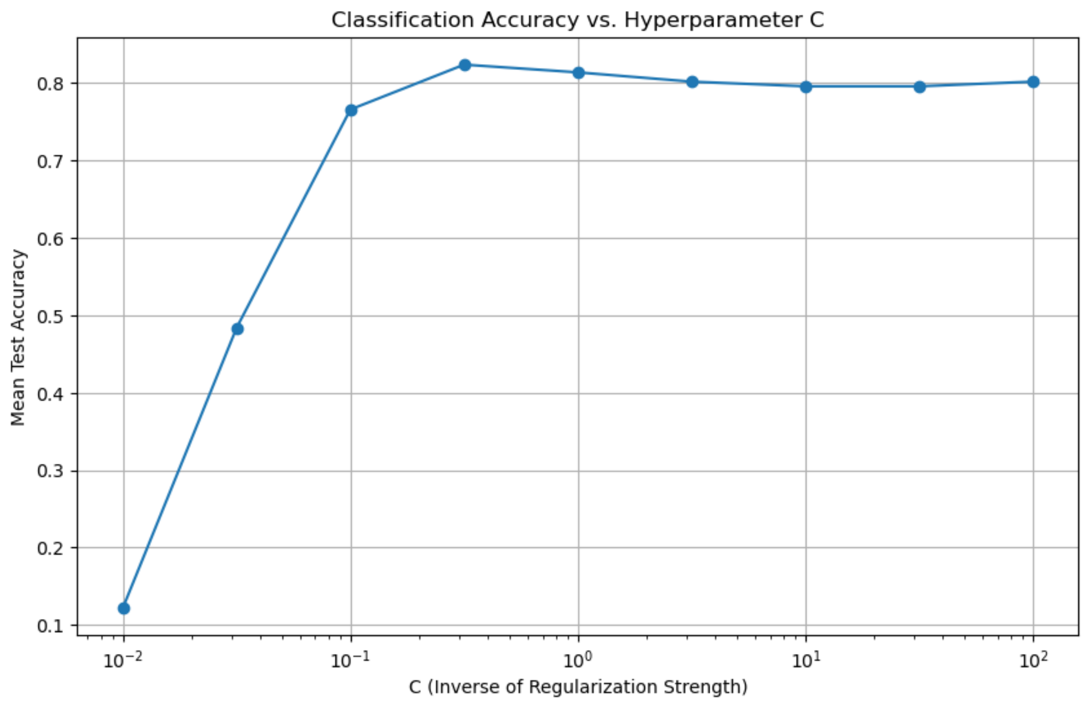


Figure 4 — Plot the classification accuracy with respect to hyperparameter C

The plot clearly indicates that the highest accuracy is achieved when the value of the hyper-parameter C approaches approximately 0.3.

2.6 Calculate confusion matrix, precision, recall and F1-score. Comment on your results.

Listing 2.5 — Calculate confusion matrix

```
from sklearn.metrics import confusion_matrix, precision_score,
recall_score, f1_score
# Build the final model with the best hyperparameter
final_model = LogisticRegression(penalty='l1', solver='liblinear',
multi_class='auto', C=best_params)
final_model.fit(X_train, y_train)

# Predict on the test data
y_pred = final_model.predict(X_test)
# Calculate the confusion matrix
confusion_matrix_result = confusion_matrix(y_test, y_pred)

# Calculate precision, recall, and F1-score
precision_result = precision_score(y_test, y_pred, average='weighted')
recall_result = recall_score(y_test, y_pred, average='weighted')
f1_score_result = f1_score(y_test, y_pred, average='weighted')

print("Confusion␣Matrix:")
print(confusion_matrix_result)
print(f"Precision:␣{precision_result:.2f}")
print(f"Recall:␣{recall_result:.2f}")
print(f"F1-Score:␣{f1_score_result:.2f}")
```

```
Confusion Matrix:
[[ 5  0  0  0  0  1  1  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  9  0  0  0  0  1]
 [ 0  0  0  1  0  3  0  0  0  0]
 [ 0  0  0  0  1  0  7  0  1  0]
 [ 0  0  0  0  0  0  0 11  0  3]
 [ 0  0  0  0  0  1  0  0 11  0]
 [ 0  0  0  1  1  0  0  0  0 13]]
Precision: 0.87
Recall: 0.86
F1-Score: 0.86
```

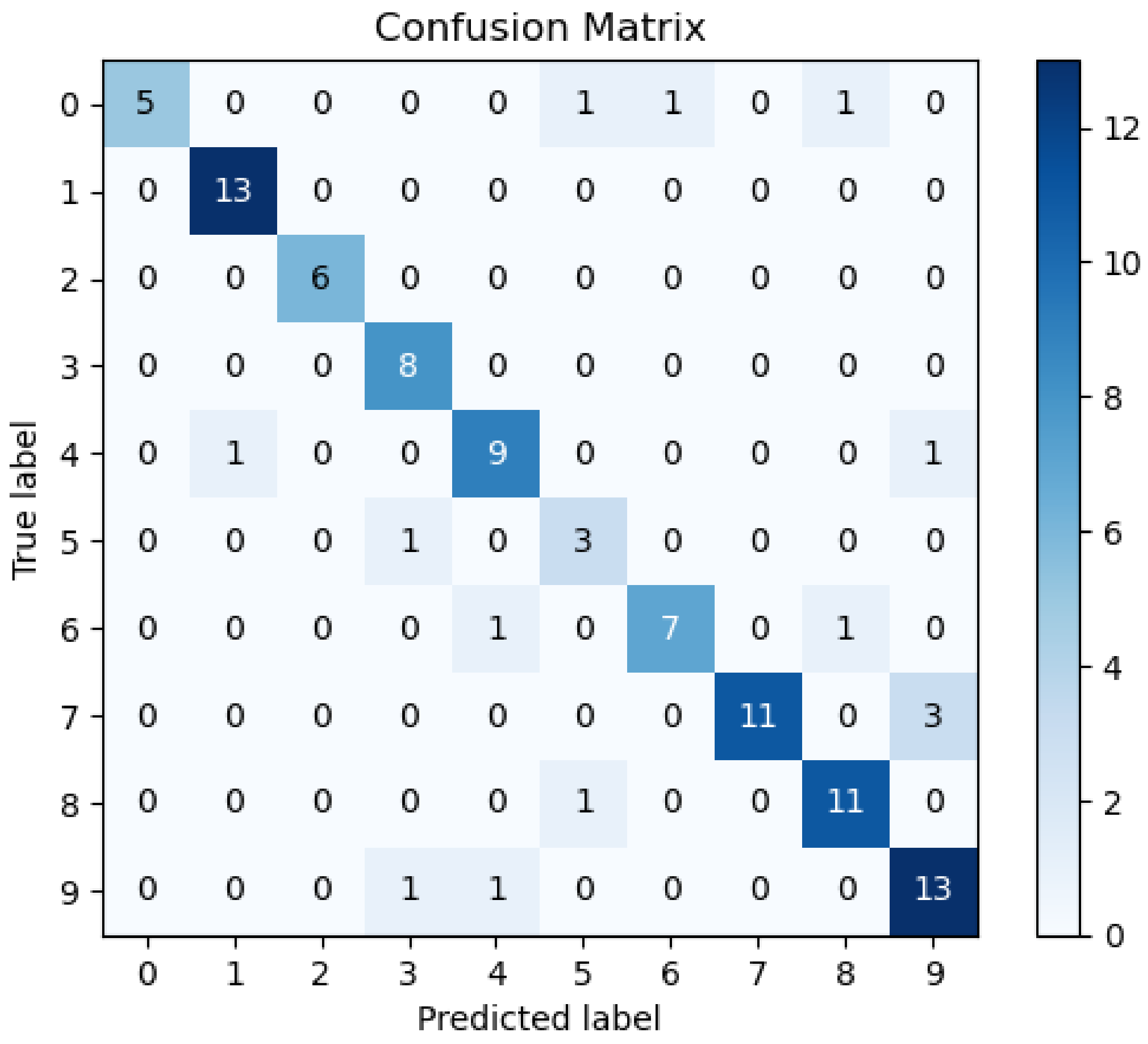
As a verification step, we employ an alternative method to compute the confusion matrix.

Listing 2.6 — Display confusion matrix

```python
import scikitplot as skplt
import matplotlib.pyplot as plt

# Assuming you have y_test and y_pred defined

# Plot the confusion matrix without normalization
skplt.metrics.plot_confusion_matrix(y_test, y_pred, normalize=False)
plt.title('Confusion␣Matrix')
plt.show()
```



True label \ Predicted label	0	1	2	3	4	5	6	7	8	9
0	5	0	0	0	0	1	1	0	1	0
1	0	13	0	0	0	0	0	0	0	0
2	0	0	6	0	0	0	0	0	0	0
3	0	0	0	8	0	0	0	0	0	0
4	0	1	0	0	9	0	0	0	0	1
5	0	0	0	1	0	3	0	0	0	0
6	0	0	0	0	1	0	7	0	1	0
7	0	0	0	0	0	0	0	11	0	3
8	0	0	0	0	0	1	0	0	11	0
9	0	0	0	1	1	0	0	0	0	13

Figure 5 — Visualization of the data before and after normalization

The model has achieved an accuracy of approximately 0.85, indicating that it correctly classifies approximately 85% of the test data samples. This suggests the model's efficiency in distinguishing between different digits within the MNIST dataset.

Regarding the Confusion Matrix:

- The confusion matrix provides detailed insights into the model's performance for each digit type.
- Key observations:

◦ Diagonal elements (from top-left to bottom-right) represent the number of correctly classified samples for each digit.
◦ Off-diagonal elements signify instances of incorrect classifications.

In terms of Precision:

- Precision is approximately 0.87, which measures the ratio of correctly predicted positive instances out of the total predicted positive instances. In multi-class classification, precision is calculated separately for each class and then averaged. It reflects the model's ability to minimize false positives.

Regarding Recall:

- The recall is approximately 0.86, measuring the ratio of correctly predicted positive instances out of the total actual positive instances. Similar to precision, recall is calculated for each class separately and then averaged. It indicates the model's ability to capture all relevant instances of each class.

Concerning the F1 Score:

- The F1 score is approximately 0.86, representing the harmonic mean of precision and recall. It assesses the model's performance in managing both false positives and false negatives. A high F1 score indicates a well-balanced model.

In summary, the model exhibits strong classification performance, as evidenced by its accuracy and the detailed analysis provided by precision, recall, and the F1 score.

Based on ”James, G., Witten, D., Hastie, T., Tibshirani, R. (2013). An introduction to statistical learning (Vol. 112, p. 18). New York: springer.”
1. Consider a dataset collected from a statistics class that includes information on students. The dataset includes variables x1 representing the number of hours studied, x2 representing the undergraduate GPA, and y indicating whether the student received an A + in the class. After conducting a logistic regression analysis, we obtained the following estimated coefficients: w0 = −6, w1 = 0.05, and w2 = 1.
Using programming

(a) What is the estimated probability that a student, who has studied for 40 hours and has an undergraduate GPA of 3.5, will receive an A + in the class?

(b) To achieve a 50% chance of receiving an A + in the class, how many hours of study does a student like the one in part (1a) need to complete?

Listing 3.1 — Estimated probability

```python
import math

# Given coefficients and input features
w0 = -6
w1 = 0.05
w2 = 1
x1 = 40
x2 = 3.5

# Calculate the estimated probability using logistic regression
probability = 1 / (1 + math.exp(-(w0 + w1 * x1 + w2 * x2)))
print(f"The␣estimated␣probability␣is:␣{probability:.2f}")

# Calculate the required hours to achieve a 50% chance of receiving an A+
required_hours = (6 - 3.5) / 0.05
print(f"To␣achieve␣a␣50%␣chance␣of␣receiving␣an␣A+,␣a␣student␣needs
to␣study␣for␣{required_hours:.2f}␣hours.")
```

```
The estimated probability is: 0.38
To achieve a 50% chance of receiving an A+, a student needs to study for 50.00 hours.
```

Using Equations

The logistic regression model is typically expressed as:

$$P(Y=1) = \frac{1}{1+e^{-(w_0+w_1x_1+w_2x_2)}}$$

Where: - $P(Y=1)$ is the estimated probability that the student receives an A+ in the class. - w_0 is the intercept coefficient. - w_1 is the coefficient for the number of hours studied (x_1). - w_2 is the coefficient for the undergraduate GPA (x_2).

(a) What is the estimated probability that a student, who has studied for 40 hours and has an undergraduate GPA of 3.5, will receive an A + in the class?

$$P(Y=1) = \frac{1}{1+e^{-(w_0+w_1x_1+w_2x_2)}}$$

Given: - $w_0 = -6$ - $w_1 = 0.05$ - $w_2 = 1$ - $x_1 = 40$ - $x_2 = 3.5$

Substitute these values into the equation:

$$P(Y=1) = \frac{1}{1+e^{-(-6+0.05(40)+1(3.5))}}$$

Calculate the exponent:

$$P(Y=1) = \frac{1}{1+e^{-(-6+2+3.5)}}$$

Now, calculate the probability:

$$P(Y=1) = \frac{1}{1+e^{-(-0.5)}}$$

$$P(Y=1) = \frac{1}{1+e^{0.5}}$$

Using this, calculate the estimated probability:

$$P(Y=1) \approx \frac{1}{1+1.6487212707} \approx \frac{1}{2.6487212707} \approx 0.3775$$

So, the estimated probability that a student, who has studied for 40 hours and has an undergraduate GPA of 3.5, will receive an A+ in the class is approximately 0.3775 or 37.75%.

b) To achieve a 50% chance of receiving an A + in the class, how many hours of study does a student like the one in part (1a) need to complete?

$$0.5 = \frac{1}{1+e^{-(w_0+w_1x_1+w_2x_2)}}$$

Now, you want to find the value of x_1 (hours of study) that satisfies this equation for the given values of w_0, w_1, and w_2 and $x_2 = 3.5$:

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

Simplify the equation:

$$1 + e^{-(-6+0.05x_1+3.5)} = 2$$

Now, subtract 1 from both sides:

$$e^{-(-6+0.05x_1+3.5)} = 1$$

Take the natural logarithm (ln) of both sides to isolate the exponential term:

$$-6 + 0.05x_1 + 3.5 = 0$$

Now, solve for x_1:

$$0.05x_1 = 6 - 3.5$$

$$0.05x_1 = 2.5$$

$$x_1 = \frac{2.5}{0.05}$$

$$x_1 = 50$$

So, to achieve a 50% chance of receiving an A+ in the class, a student with an undergraduate GPA of 3.5 would need to study for approximately 50 hours.

Github Repository Link :- PATTERN-RECOGNITION-Assignment-02

***