EN3150 Assignment 02

Learning from data and related challenges and classification

Name: Mirihagalla M.K.D.M.
Index : 200397A

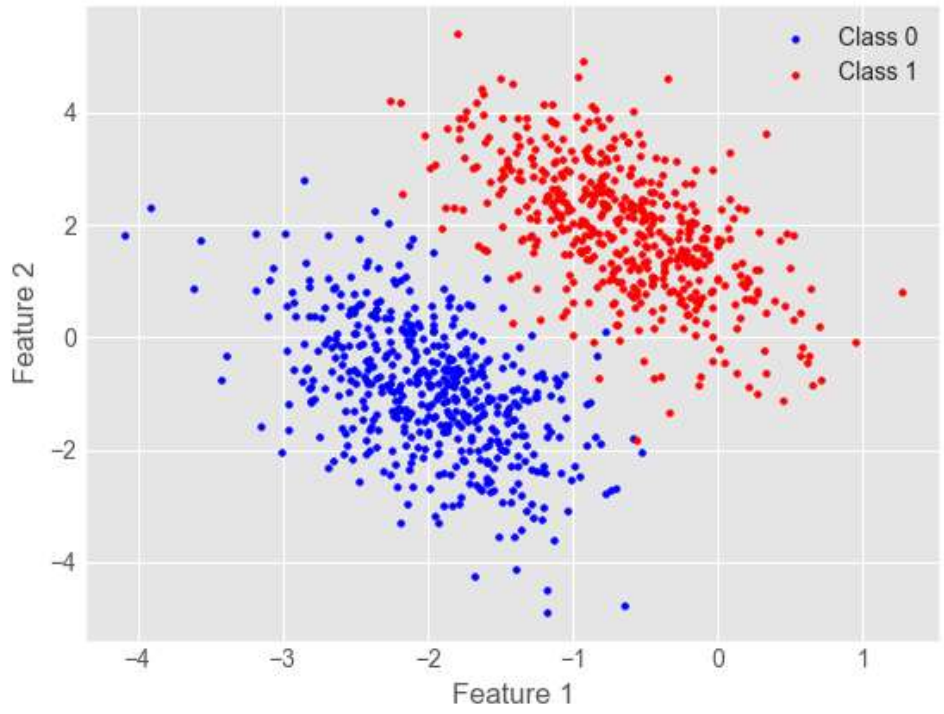
Q1: Logistic regression weight update process

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

```python
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs

# Set a random seed for reproducibility
np.random.seed(0)

# Define the centers for generating synthetic data
centers = [[-5, 0], [0, 1.5]]

# Generate synthetic data using make_blobs function
X, y = make_blobs(n_samples=1000, centers=centers, random_state=40)

# Apply a linear transformation to the data
transformation = [[0.4, 0.2], [-0.4, 1.2]]
X = np.dot(X, transformation)

# Add a bias term (1) to the feature matrix
X = np.c_[np.ones((X.shape[0], 1)), X]

# Initialize coefficients for logistic regression
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
    # Clip the predictions to avoid taking the log of 0 or 1
    y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
    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  # Learning rate for gradient descent
iterations = 10     # Number of iterations for optimization
loss_history = []   # List to store loss values over iterations
```

First Let us visualize these data points with corresponding labels.

```
import matplotlib.pyplot as plt

# Separate the data into two arrays based on the labels (0 or 1)
X_class_0 = X[y == 0]
X_class_1 = X[y == 1]

# Create a scatter plot for each class with smaller points
plt.scatter(X_class_0[:, 1], X_class_0[:, 2], label='Class 0', c='blue', s=7)
plt.scatter(X_class_1[:, 1], X_class_1[:, 2], label='Class 1', c='red', s=7)

# Label the axes
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')

# Add a legend
plt.legend()

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




```
Shape of X:  (1000, 3)
Shape of y:  (1000,)
Shape of W:  (3,)
```

1.2.

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

$$\boldsymbol{w}_{(t+1)} \leftarrow \boldsymbol{w}_{(t)} - \alpha \frac{1}{N} \left(\mathbf{1}_N^T \, \mathrm{diag}(\mathrm{sigm}(\boldsymbol{w}_{(t)}^T \boldsymbol{x}_i) - \boldsymbol{y}_i) \boldsymbol{X} \right)^T .$$

Here, $\boldsymbol{X}$ is data matrix of dimension of $\boldsymbol{N} \times (\boldsymbol{D} + \mathbf{1})$. Here, $\boldsymbol{N}$ is total number of data samples and $\boldsymbol{D}$ is number of features. Now, $\boldsymbol{X}$ is given by,

$$\boldsymbol{X} = \begin{bmatrix} 1 & x_{1,1} & x_{2,1} & \cdots & x_{D,1} \\ 1 & x_{1,2} & x_{2,2} & \cdots & x_{D,2} \\ \vdots & \vdots & \ddots & \vdots & \\ 1 & x_{1,i} & x_{2,i} & \cdots & x_{D,i} \\ \vdots & \vdots & \ddots & \vdots & \\ 1 & x_{1,N} & x_{2,N} & \cdots & x_{D,N} \end{bmatrix} = \begin{bmatrix} \boldsymbol{x}_1^T \\ \boldsymbol{x}_2^T \\ \vdots \\ \boldsymbol{x}_i^T \\ \vdots \\ \boldsymbol{x}_N^T \end{bmatrix} .$$

According to the formular given let us create a function to update the weight matrix in each iteration. And a function for calculating the loss in each iteration. Then we will loop through 10 iteration and see the results. Results are as follows.

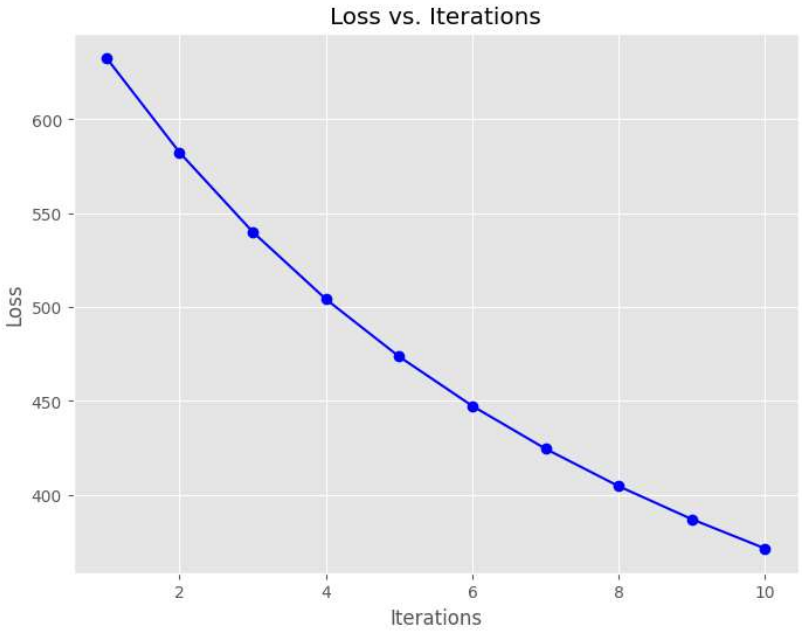
```
Final Values of W after 10 iterations
-------------------------------------
W0 :  0.009031764884538475
W1 :  0.262301155857097
W2 :  0.4994938404035172
```

```
# Define the learning rate and number of iterations
learning_rate = 0.1
iterations = 10
loss_history = []

N = X.shape[0]
ones_matrix = np.ones((N, 1))
ones_matrix_T = ones_matrix.T

# Reshape the weight vector W to (3, 1)
W = W.reshape(3, 1)
```

```
def update_w():
    global W
    global errors
    for j in range(N): # iterate over each data point
        linear_combination = np.dot(W.T, (X[j].reshape(3, 1)))
        error = sigmoid(linear_combination) - y[j]
        errors = np.append(errors, error)
    D = np.diag(errors)
    P1 = ones_matrix_T @ D
    P2 = P1 @ X
    P3 = P2.T
    W -= learning_rate * (1 / N) * P3

def calulate_loss():
    total_loss = 0
    for j in range(N):
        linear_combination = np.dot(W.T, (X[j].reshape(3, 1)))
        y_pred = sigmoid(linear_combination)
        loss = log_loss(y[j], y_pred)
        total_loss = total_loss + loss
    return total_loss

for t in range(iterations):
    errors = np.array([]) # empty array to store errors
    update_w()
    loss_history.append(calulate_loss())

print("Final Values of W after 10 iterations : ")
print('W0 : ', W[0][0])
print('W1 : ', W[1][0])
print('W2 : ', W[2][0])
print()
```

1.3 Plot the loss with respect to number of iterations.

```
loss_history = np.reshape(loss_history, (iterations, 1))
iteration_numbers = np.arange(1, iterations + 1)
plt.figure(figsize=(8, 6))
plt.plot(iteration_numbers, loss_history, marker='o', linestyle='-')
plt.title('Loss vs. Iterations')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.grid(True)
plt.show()
```



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 number of iterations as t = 10. Batch Newton's method weight update is given below,

$$\boldsymbol{w}_{(t+1)} \leftarrow \boldsymbol{w}_{(t)} - \left(\frac{1}{N}\,\boldsymbol{X}^T\boldsymbol{S}\boldsymbol{X}\right)^{-1}\left(\frac{1}{N}\,\left(\mathbf{1}_N^T\,\mathrm{diag}(\mathrm{sigm}(\boldsymbol{w}_{(t)}^T\boldsymbol{x}_i) - \boldsymbol{y}_i)\boldsymbol{X}\right)^T\right).$$

and is S given by,

$$\boldsymbol{S} = \mathrm{diag}(s_1,\, s_2, \ldots, s_N),$$
$$\boldsymbol{s_i} = \left(\mathrm{sigm}(\boldsymbol{w}_{(t)}^T\boldsymbol{x}_i) - \boldsymbol{y}_i\right)\left(1 - \mathrm{sigm}(\boldsymbol{w}_{(t)}^T\boldsymbol{x}_i) - \boldsymbol{y}_i\right).$$

After performing the gradient descent with Newton's method, I could acquire the following results after 10 iterations.

```
Final Values of W after 10 iterations (Newton's Method)
-------------------------------------------------------
W0 :  3.145757812061996
W1 :  2.9679207749775514
W2 :  1.4806766332837635
```

This is the code Snippets I used for this calculation.

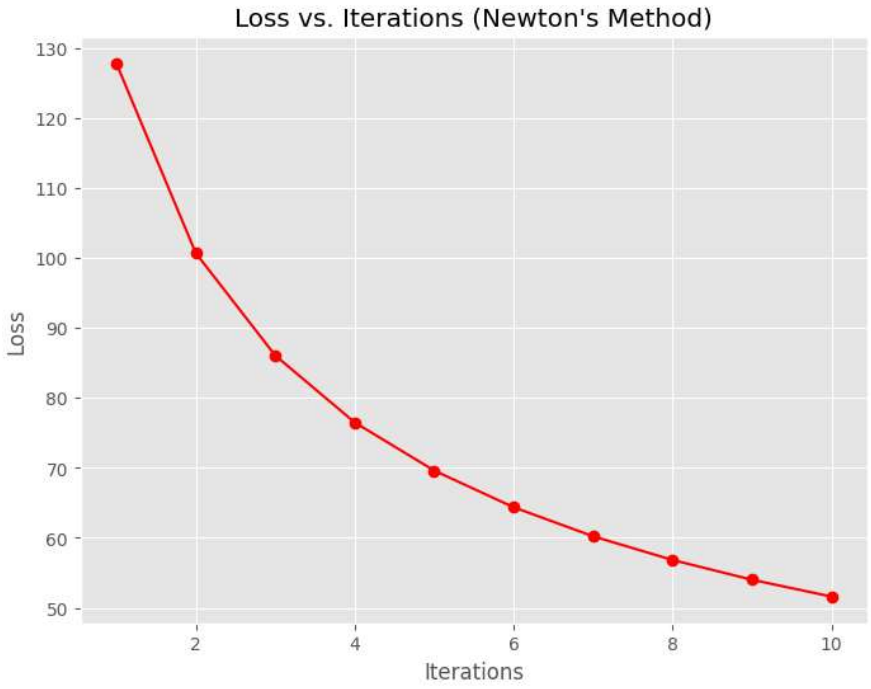
```
import numpy as np
from numpy.linalg import inv

learning_rate = 0.1
iterations = 10

loss_history_newton = []
N = X.shape[0]

W_newton = np.zeros(X.shape[1])
W_newton = W_newton.reshape(3, 1)

Mat_1N = np.ones((N, 1))
Mat_1N_T = Mat_1N.T

# Define a function to compute P1
def compute_P1(X, W, y):
    S_vect = np.array([])
    for i in range(N):
        temp3 = np.dot(W.T, (X[i].reshape(3, 1)))
        S_i = (sigmoid(temp3) - y[i]) * (1 - sigmoid(temp3) - y[i])
        S_vect = np.append(S_vect, S_i)
    S = np.diag(S_vect)
    P1 = np.dot(X.T, S)
    P1 = np.dot(P1, X)
    P1 = P1 / N
    return P1

# Define a function to compute P2
def compute_P2(X, W_newton, y):
    V = np.array([])
    for j in range(N):
        temp = np.dot(W_newton.T, (X[j].reshape(3, 1)))
        Mat_Error = sigmoid(temp) - y[j]
        V = np.append(V, Mat_Error)
    D = np.diag(V)
    P2 = np.dot(Mat_1N_T, D)
    P2 = np.dot(P2, X)
    P2 = P2.T
    P2 = P2 / N
    return P2
```

```
# Perform gradient descent
for t in range(iterations):
    P1 = compute_P1(X, W, y)
    P1_inverse = inv(P1)
    P2 = compute_P2(X, W_newton, y)

    # Update the weights using Newton's method
    W_newton = W_newton - np.dot(P1_inverse, P2)

    # Calculate the total loss for this iteration
    total_loss = 0
    for j in range(N):
        temp2 = np.dot(W_newton.T, (X[j].reshape(3, 1)))
        y_pred = sigmoid(temp2)
        loss = log_loss(y[j], y_pred)
        total_loss = total_loss + loss

    loss_history_newton.append(total_loss)

print("Final Values of W after 10 iterations (Newton's Method) ")
print("-------------------------------------------------------")
print('W0 : ', W_newton[0][0])
print('W1 : ', W_newton[1][0])
print('W2 : ', W_newton[2][0])
print()
```

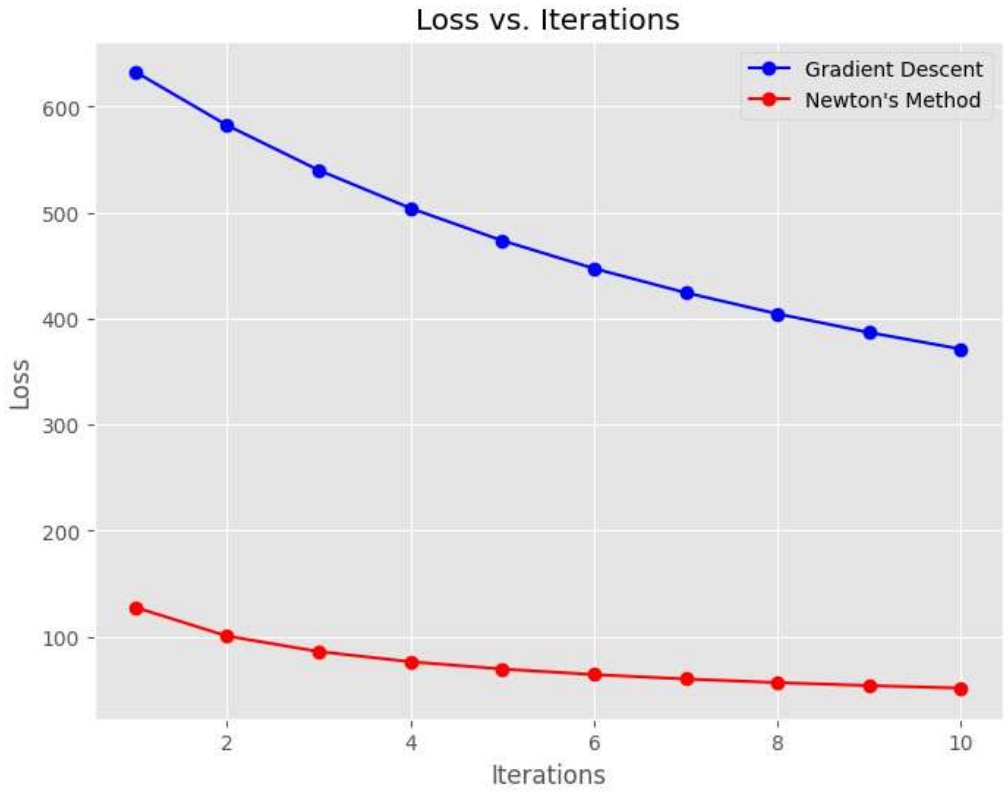
---

5. Plot the loss with respect to number of iterations.

```
import matplotlib.pyplot as plt

# Plot the Loss vs. Iterations for Newton's method
plt.style.use('ggplot')
plt.figure(figsize=(8, 6))
iteration_numbers = np.arange(1, iterations + 1)
plt.plot(iteration_numbers, np.reshape(loss_history_newton, (iterations, 1)),
marker='o', linestyle='-',color='red')
plt.title("Loss vs. Iterations (Newton's Method)")
plt.xlabel("Iterations")
plt.ylabel("Loss")
plt.grid(True)
plt.show()
```



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



Explanation based on the observations.

Let us compare the loss value vs the iteration.

Iteration Number	Gradient Descent	Newton's Method
1	632.821152	127.859691
2	582.397295	100.717651
3	540.040023	86.078530
4	504.205110	76.502893
5	473.638953	69.620454
6	447.343296	64.379284
7	424.529851	60.226147
8	404.577017	56.837487
9	386.993210	54.009567
10	371.387546	51.606868

As we can see the loss of the gradient descent method is quite high compared to newton's method. That means that if we use the newton method for this data set with learning rate of 0.01, we can achieve a faster convergence rate.

After only 10 iterations newton method is able to catch 51 loss value. To accomplish that loss gradient descent method has to iterate a lot more than that.

After running the algorithm for 1000 iterations I observed the following loss value

```
loss_history[-5:-1, 0]
✓ 0.0s
array([54.4748951 , 54.44939636, 54.42393473, 54.3985101 ])
```

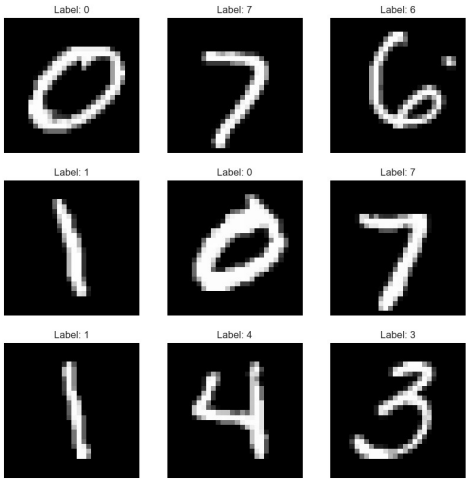
That is close to 50 but still high. Meaning that after much larger no of iterations, loss reduces but it extremely lows in speed.

Q2: Perform Grid Search for Hyperparameter Tuning

2.1. Use the code given to 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

train_samples = 500
X, y = fetch_openml("mnist_784", version=1, return_X_y=True, as_frame=False)
random_state = check_random_state(0)

# Permute the data to shuffle it
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)
```

First Let us visualize couple of images.



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

- We use a fixed random **seed** to ensure that the shuffling process is predictable and can be replicated in the future.
- The permutation array contains shuffled indices that represent a random rearrangement of the data points. It is vital for **breaking** any inherent order or **patterns** in the dataset.
- Shuffling is essential to eliminate biases that might arise from the original data order and create a more evenly distributed training and testing dataset.
- Importantly, the same shuffling order is applied to both the feature matrix (X) and the target labels (y). This guarantees that the relationships between features and labels remain consistent after shuffling.
- The outcome is a randomized, balanced dataset that is ideal for machine learning, as it **prevents** any **unintentional biases** or patterns that might have existed in the original data order.

2.3. Use lasso logistic regression for image classification as

"$\boldsymbol{LogisticRegression(penalty = 'l1', solver = 'liblinear', multi\_class = 'auto')}$".

Next, create a pipeline that includes the scaling, the Lasso logistic regression estimator, and a parameter grid for hyperparameter tuning (C value).

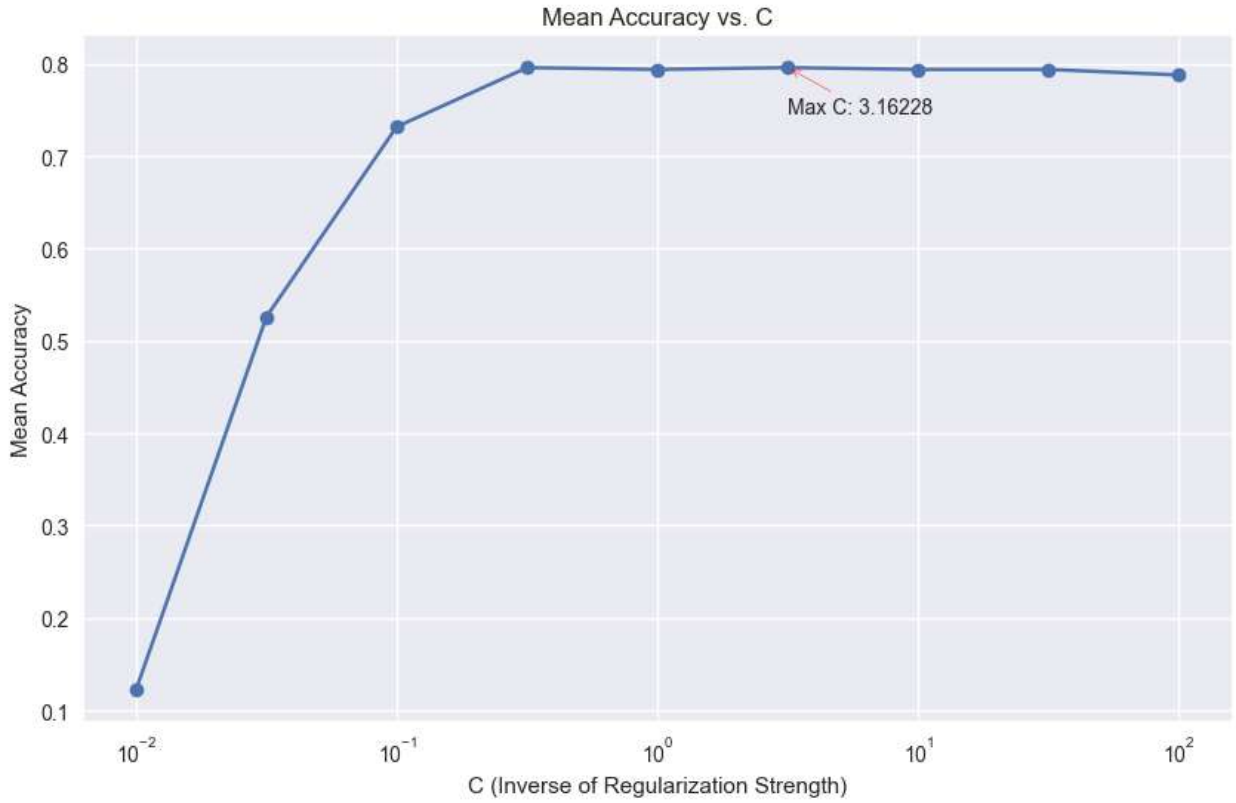
```
# Create a pipeline with scaling and Lasso logistic regression
pipeline = Pipeline([
    ('scaler', StandardScaler()),  # Standardize the data
    ('lasso_logistic', LogisticRegression(penalty='l1', solver='liblinear',
multi_class='auto'))
])

# Create a parameter grid for the grid search
param_grid = {
    'lasso_logistic__C': np.logspace(-2, 2, 9)
}
```

2.4. Use $\boldsymbol{GridSearchCV}$ to perform a grid search over the range (e.g., $\boldsymbol{np.logspace(-2, 2, 9)}$) of to find optimal value of hyperparameter $\boldsymbol{C}$.

```
# Create a grid search cross-validation object
grid_search = GridSearchCV(pipeline, param_grid, cv=5)

# Fit the model to the data and find the best hyperparameters
grid_search.fit(X_train, y_train)

# Get the best hyperparameters
best_params = grid_search.best_params_

# Print the best hyperparameters
print("Best Hyperparameters:", best_params)
```

```
Best Hyperparameters: {'lasso_logistic__C': 0.31622776601683794}
```

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

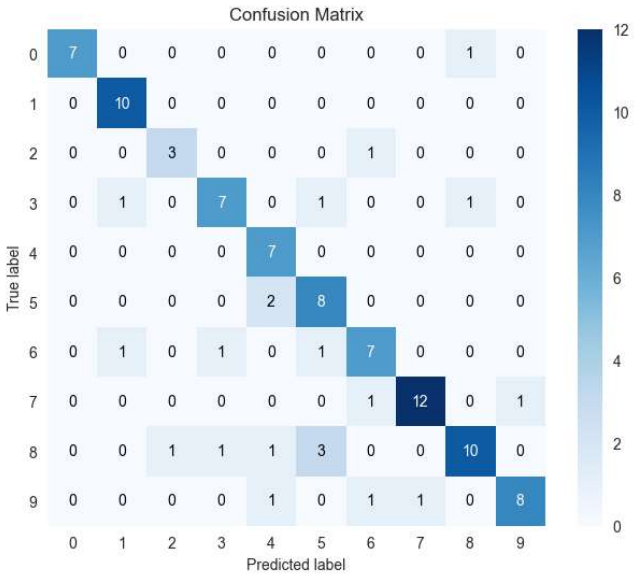


Comments

Almost all the C values give approximately 0.8 accuracy but out of them the best one was selected as 3.16228.

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

```
from sklearn.metrics import precision_score, recall_score, f1_score

y_pred = grid_search.predict(X_test)
precision = precision_score(y_test, y_pred, average='weighted')
recall = recall_score(y_test, y_pred, average='weighted')
f1 = f1_score(y_test, y_pred, average='weighted')
import scikitplot as skplt
import matplotlib.pyplot as plt
# Plot the confusion matrix
skplt.metrics.plot_confusion_matrix(y_test, y_pred, normalize=False, figsize=(8, 6))
plt.title("Confusion Matrix")
plt.show()
# Print precision, recall, and F1-score
print(f'Precision: {precision:.6f}')
print(f'Recall: {recall:.6f}')
print(f'F1-score: {f1:.6f}')
```



Comments

This data set has high precision, recall, f1 score values. Meaning that it has low no of false positives and low no of false negatives.

When we look at the confusion matric it is obvious that there are couple of misclassifications. But the highest no of misclassifications happens for 8 and 5 , that make sense because this model thinks that the shape more like same.

Precision: 0.807537

Recall: 0.790000

F1-score: 0.790000

3. Logistic regression

3.1. Consider a dataset collected from a statistics class that includes information on students. The dataset includes variables x_1 representing the number of hours studied, x_2 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: $w_0 = -6, w_1 = 0.05, and\ w_2 = 1$.

x_1 = No of hours studied.
x_2 = GPA
y = Recieved A^+

Weights

$$\begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} -6 \\ 0.05 \\ 1 \end{bmatrix}$$

Regression model for Predicting probabilities.

$$Pr(y) = Sigmoid\ (w_0 + w_1 x_1 + w_2 x_2)$$

$$= \frac{1}{1 + e^{-(w_0 + w_1 x_1 + w_2 x_2)}}$$

(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?

(a) $x_1 = 40$
$x_2 = 3.5$

$$Pr(y) = \frac{1}{1 + e^{-(-6 + 0.05 \times 40 + 1 \times 3.5)}}$$

$$= 37.5\ \%$$

(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?

(b) $Pr(y) = 0.5$
$x_1 = ?$
$x_2 = 3.5$

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

$x_1 = 50$ hrs