Student Name: K.K.D.Kariyawasam
Student ID: 200289U

EN3150 Assignment 02

Learning from data and related challenges and classification

---

1 Logistic regression weight update process

1. Data was generated using the provided code and subsequently plotted for visualization purposes.

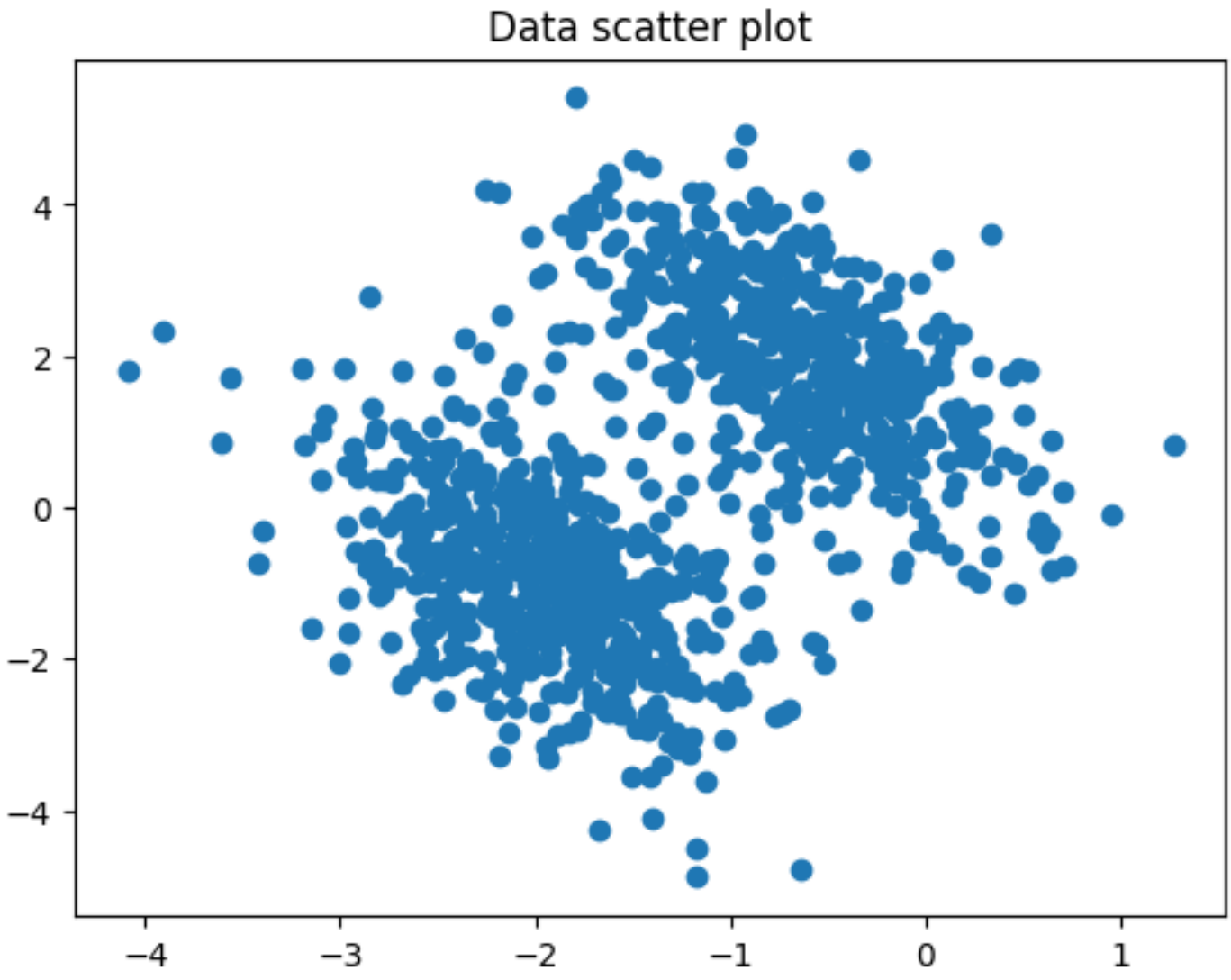

Figure 1: Scatter plot of the generated data

2. Implementation of batch gradient descent algorithm is shown below. In here batch gradient descent was used to optimizing the weights of a binary classification using the binary cross entropy loss.

Gradient Descent

- The algorithm iteratively updates the model weights W to minimize the loss function.
- The learning rate (`learning_rate`) controls the step size for weight updates.

Loss Calculation

- The loss is computed using the log loss (binary cross-entropy) function, which is common for binary classification tasks.
- The loss is tracked and stored in the `loss_history` list to monitor the convergence of the algorithm.

```
# Performing gradient descent based weight update
N = len(y)

for i in range(iterations):
    # Predict the target values using the sigmoid function
    y_pred = sigmoid(np.dot(X,W))
    # Calculate the gradient of the loss function
    grad = (np.dot(np.ones(N).T, np.dot(np.diag(y_pred - y),X)) / N)
    # Update the weights
    W -= learning_rate*grad
    # Calculate the loss and append to the loss_history
    loss = np.sum(log_loss(y, y_pred))/N
    loss_history.append(loss)

# Plot the loss curve
plt.plot(range(iterations),loss_history)
plt.title("Batch gradient descent method Loss")
plt.xlabel("Iterations")
plt.ylabel("Loss")
plt.show()

```

3. The above calculated loss is then plotted to observe the change of loss with each iteration. Resulted plot is shown below.

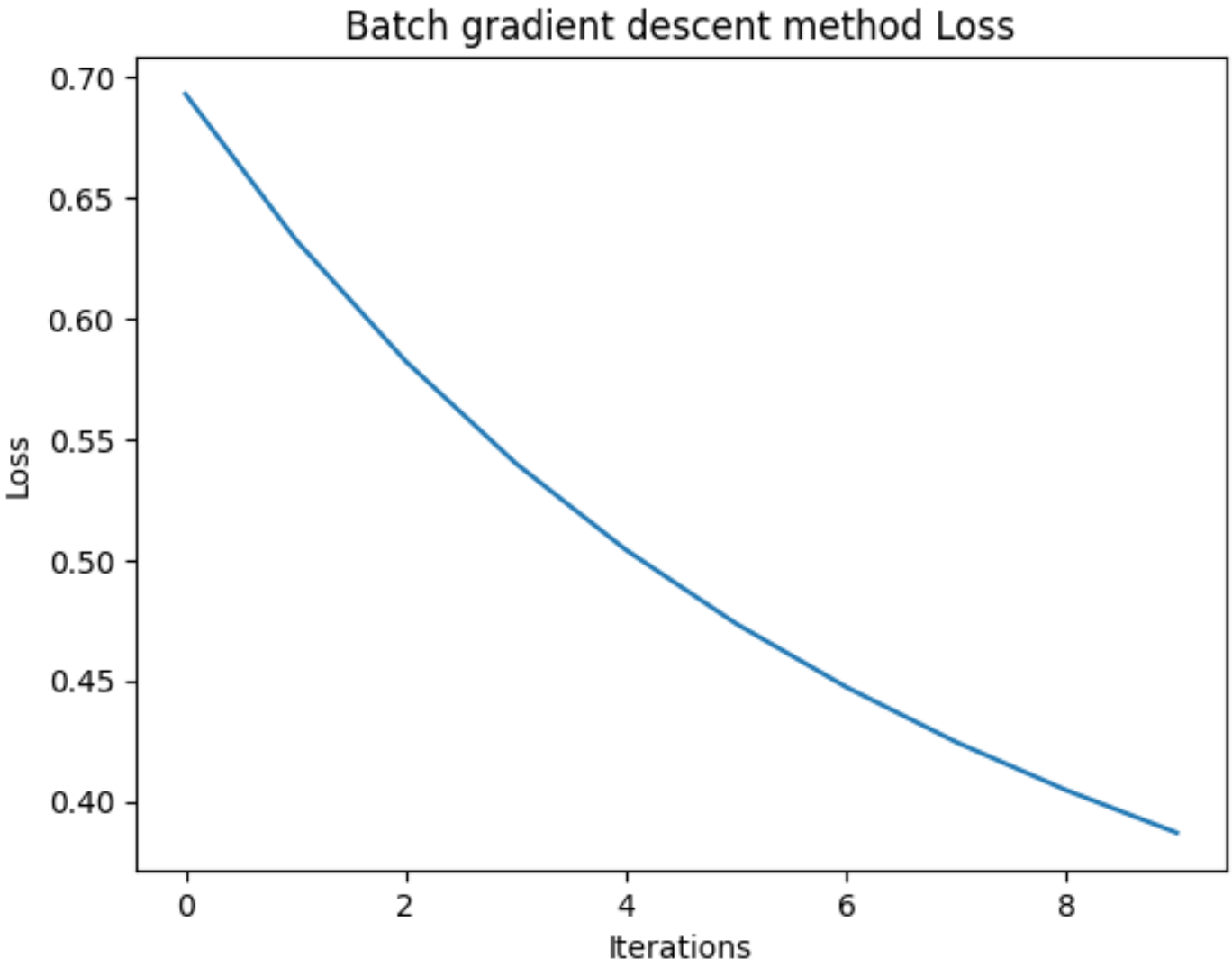

Figure 2: Loss curve for batch gradient descent method

- The weight updating process was done for 10 iterations.
- According to the results obtained (above curve) the loss is gradually decreasing with the iterations.
- During the first iteration the loss is at a value of 0.6931 and after the tenth iteration value of the loss is 0.38699.
- The loss does not exhibit stabilization after the tenth iteration, suggesting that the optimization process may not have converged completely. There is a possibility that further iterations could lead to a reduction in the loss.

4. In this part, Newton's method was used to update the weights for the same data. Similar to the previous step loss was calculted and plotted for the visualization purpose.

```
# Performing Newton's method based weight update
from numpy.linalg import inv

# Initialize weights as zeros
W = np.zeros(X.shape[1])
loss_history_newton = []

for i in range(iterations):
    # Predict the target values using the sigmoid function
    y_pred = sigmoid(np.dot(X, W))
    # Calculate si values
    s = (y_pred - y) * (1 - y_pred - y)
    # Calculating S
    S = np.diag(s)
    # Calculating the gradient
    grad_1 = inv(np.dot(np.dot(X.T, S), X) / len(y))
    grad_2 = np.dot(np.dot(np.ones(len(y)).T, np.diag(y_pred - y)), X) /
    len(y)
    grad_newton = np.dot(grad_1, grad_2)
    # Updating weights
    W -= grad_newton
    # Calculating loss
    loss = np.sum(log_loss(y, y_pred))/N
    loss_history_newton.append(loss)

# Plotting the loss curve
plt.plot(range(iterations), loss_history_newton)
plt.title("Newton's Method Loss")
plt.xlabel("Iterations")
plt.ylabel("Loss")
plt.show()

```

5. The above calculated loss is then plotted to observe the change of loss with each iteration. Resulted plot is shown below.

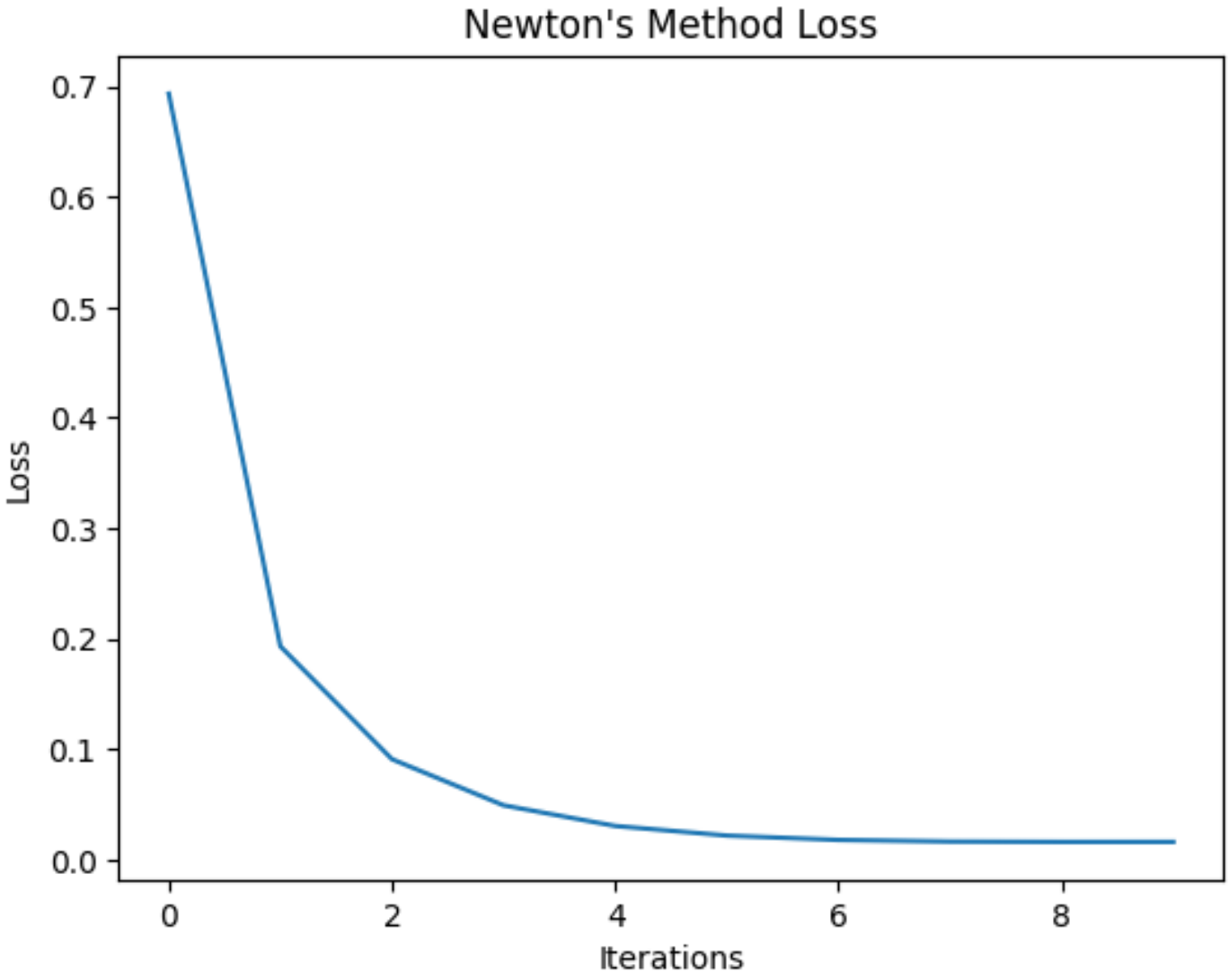

Figure 3: Loss curve for Newton's method

- The weight updating process was done for 10 iterations.
- According to the results obtained (above curve) the loss is gradually decreasing with the iterations which is same as batch gradient descent.
- During the first iteration the loss is at a value of 0.6931 and after the tenth iteration value of the loss is 0.0163.
- It can be clearly see that the loss has converged after the tenth iteration.

6. To compare the above to methods of updating weights both the loss curves are plotted in a single plot.

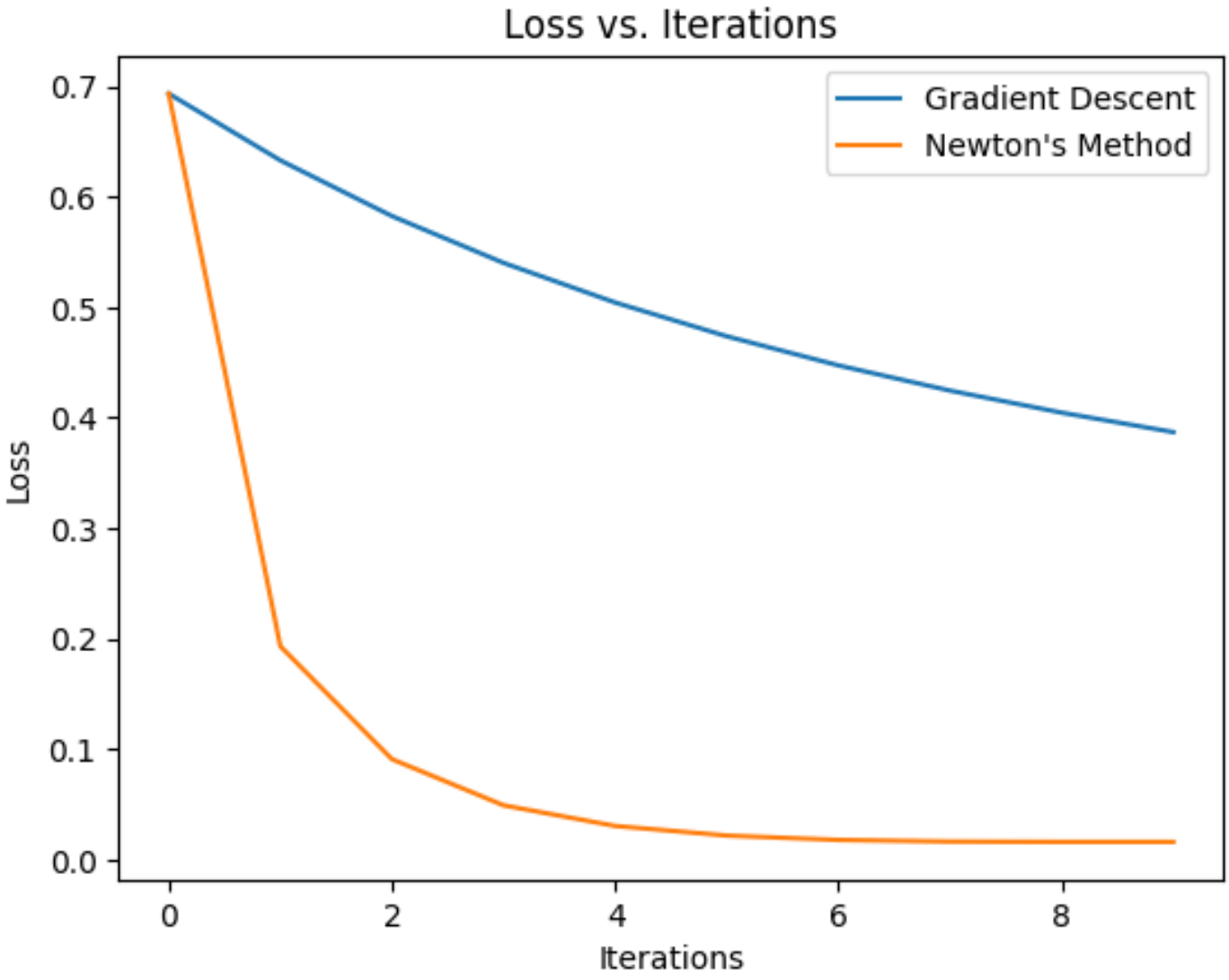

Figure 4: Both loss curves in a single plot

According to the results obtained after ten iterations we can see that when using the Newton's method loss has converged while when using batch gradient descent loss has not converged for the generated data. This means that Newton's method converges faster than the batch gradient descent method. In summary, Newton's method demonstrates higher convergence speed and loss reduction compared to batch gradient descent for the given data and binary cross-entropy loss. However, this can be different when the dataset changes.

2 Perform grid search for hyper-parameter tuning

2. Data shuffling is an important step used before splitting it into training and testing sets to ensures that the data points are not in any specific order that might introduce patterns when training the model. By applying this permutation to the feature matrix X and matrix Y we can rearrange the data points in random order.

3. - A Lasso logistic regression with created using `LogisticRegression` class in scikit learn library. A scikit-learn pipeline is created, including data scaling and the Lasso logistic regression estimator.
   - Then a parameter grid for hyperparameter tuning is created for regularization strength parameter(C).

```
# Create a lasso LogisticRegression classifier
logistic = LogisticRegression(penalty='l1', solver='liblinear',
    multi_class='auto')

# Define a Standard Scaler to normalize inputs
scaler = StandardScaler()
# Create a pipeline to chain preprocessing steps and the logistic
    classifier
pipe = Pipeline(steps=[("scaler", scaler), ("logistic", logistic)])
# Define a parameter grid for hyperparameter tuning
param_grid = {
    "logistic__C": np.logspace(-2, 2, 9),
}
```

4. - Hyperparameter tuning is performed to find the optimal C value, and the model's performance is evaluated. This was done by performing the gridsearch using the `GridSearchCV` function.
   - By hyperparameter tuning we can find the best parameter values to increase the performance of the model.

```
# Perform grid search for hyperparameter tuning
search = GridSearchCV(pipe, param_grid, cv=5, scoring='accuracy',
    verbose=1)
search.fit(X_train, y_train)
# Get the best model
best_model = search.best_estimator_
# Predict using the best model
y_pred = best_model.predict(X_test)
# Calculated accuracy for the test set
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy on the test set: ", accuracy)

print("Best parameter (CV score=%0.3f):" % search.best_score_)
print(search.best_params_['logistic__C'])
```

It is important to note that sometimes grid search is computationally expensive specially when parameter grid is too large.

- Accuracy on the test set: 0.85

- Best C value is 0.31622776601683794 with a cross validation score of 0.824

5. Classification accuracy with respect to hyperparameter C and the plot obtained is shown below.

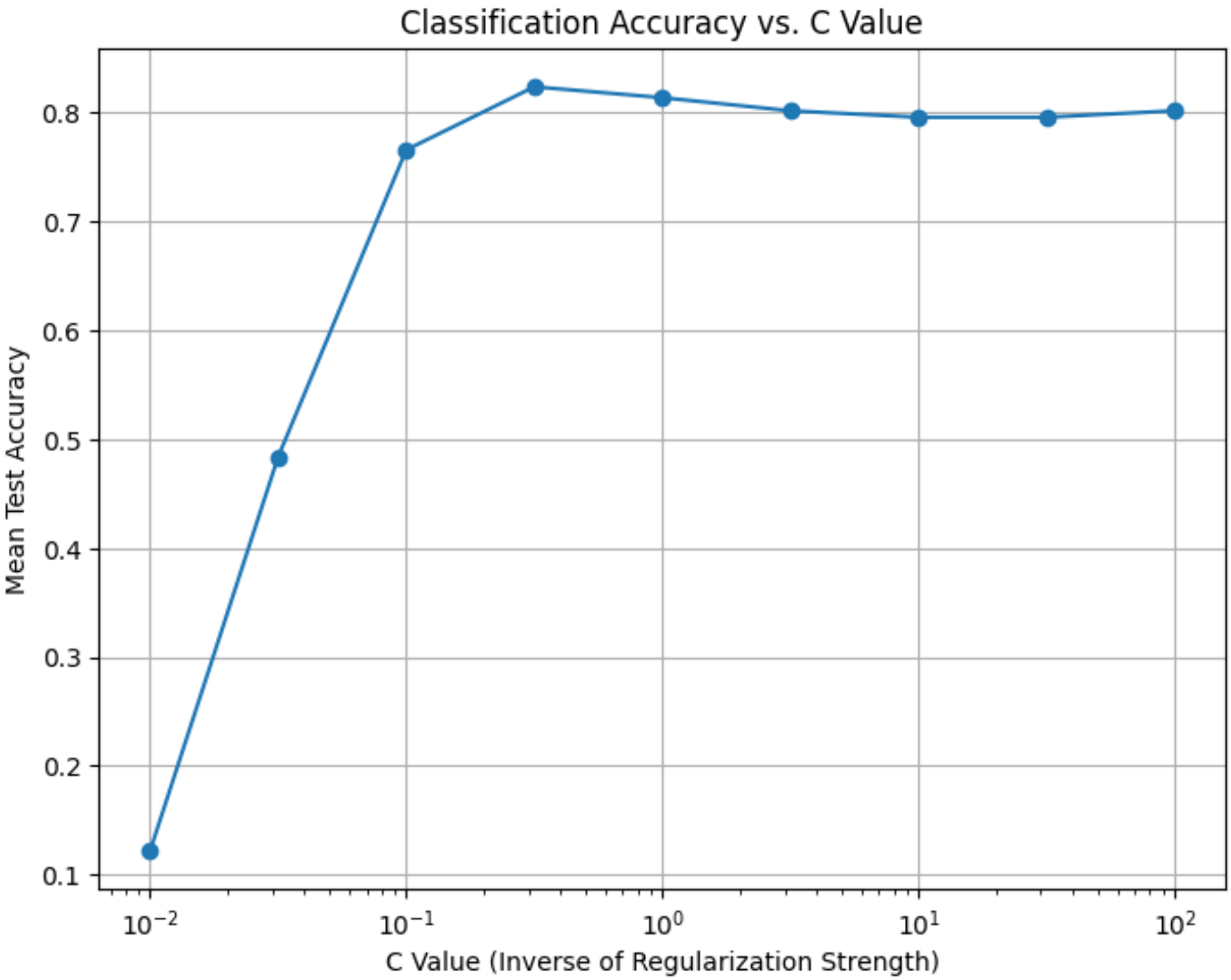

Figure 5: Classification accuracy with C value

- According to the obtained graph when C increases from 0.01 to 100 test accuracy increases generally.
- The test accuracy reaches its peak when C = 0.316 and beyond this point, as C increases accuracy decrease by some amount and levels near a value of 0.8.
- This indicates that accuracy doesn't continue to increase as the C value increases and it implies it is important to balance between model complexity and generalization.
- By this way we can avoid over fitting and increase performance on unseen data.

6. Macro averaged f1 score, precision and recall was calculated and the confusion matrix also calcuted using the following code.

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

# Predict on the test set using the best C value
best_model = search.best_estimator_
y_pred = best_model.predict(X_test)

# Calculate confusion matrix
confusion = confusion_matrix(y_test, y_pred)

# Calculate precision, recall, and F1-score
precision = precision_score(y_test, y_pred, average='macro')
recall = recall_score(y_test, y_pred, average='macro')
f1 = f1_score(y_test, y_pred, average='macro')

print(f"Confusion Matrix:\n{confusion}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"F1-score: {f1:.4f}")
```

Confusion matrix

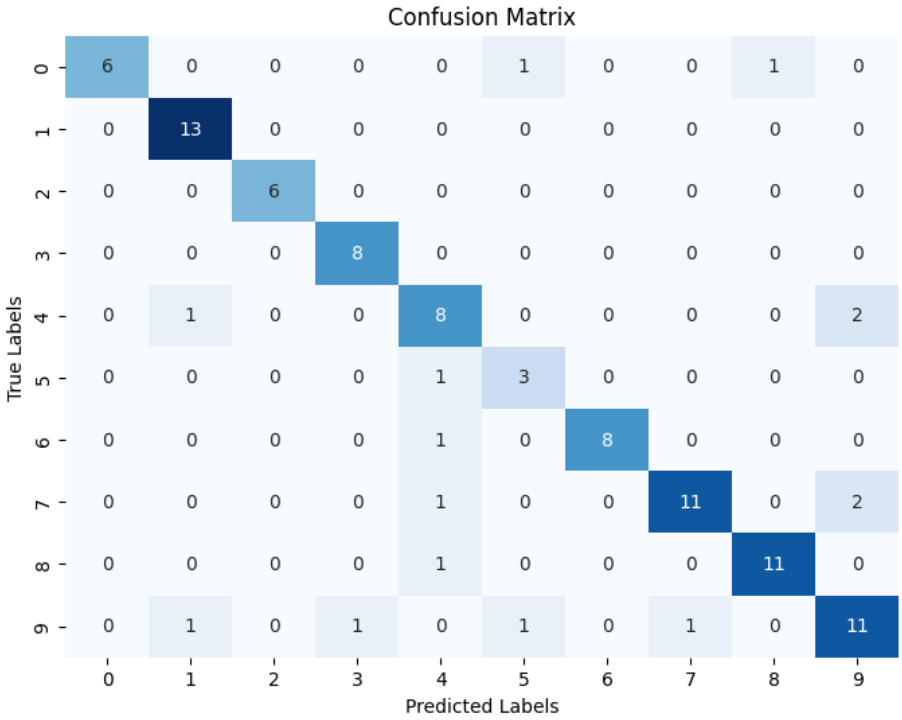

Figure 6: Confusion matrix

Confusion matrix is a valuable representation to get an idea about the performance of a classification model. The created model's performance is good based on the obtained confusion matrix.

It was able to correctly identify most of the classes (according to the high values in the diagonal which represent the correct predictions of each class). Also, it has correctly classified class 1 with 13 true positives. But, there are some misclassifications as well. When considering class 9, where the model incorrectly classified it in 4 instances. Overall, the model demonstrates good performance. Code to obtain the above confusion matrix plot.

```
import matplotlib.pyplot as plt
import seaborn as sns

# Create a heatmap of the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(confusion, annot=True, fmt='d', cmap='Blues', cbar=False)
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('Confusion Matrix')
plt.show()
```

- **Precision:** 0.8589
- **Recall:** 0.8552
- **F1-score:** 0.8527

Precision measures the accuracy of positive predictions, while measures the completeness of positive predictions. F1-score balanced measure which consist of both precision and recall. For this model precision, recall and F1 score are at a value near 0.85 which indicates model is performing well.

3 Logistic regression

1. (a) To calculate 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 using the logistic regression formula:

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

Substituting estimated coefficients $w_0 = -6$, $w_1 = 0.05$, and $w_2 = 1$, and the values $x_1 = 40$ and $x_2 = 3.5$:

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

$$P = 0.37754$$

So, probability of getting a GPA of 3.5 with 40 hours of study is 0.37754.

(b) To find how many hours of study need to achieve a 50% of chance of receiving a A+ in the class we can follow the below steps.

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

Since chance is 50%, P=0.5. Hence by substituting P and estimated coefficients $w_0 = -6$, $w_1 = 0.05$ to the above equation,

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

$$-(-6 + 0.05 \cdot x_1 + 3.5) = 0 \implies x_1 = 50$$

Need to study 50 hours.