Electronic and Telecommunication engineering

University of Moratuwa

EN3150 - Pattern Recognition

A01 Learning from data and related challenges and classification

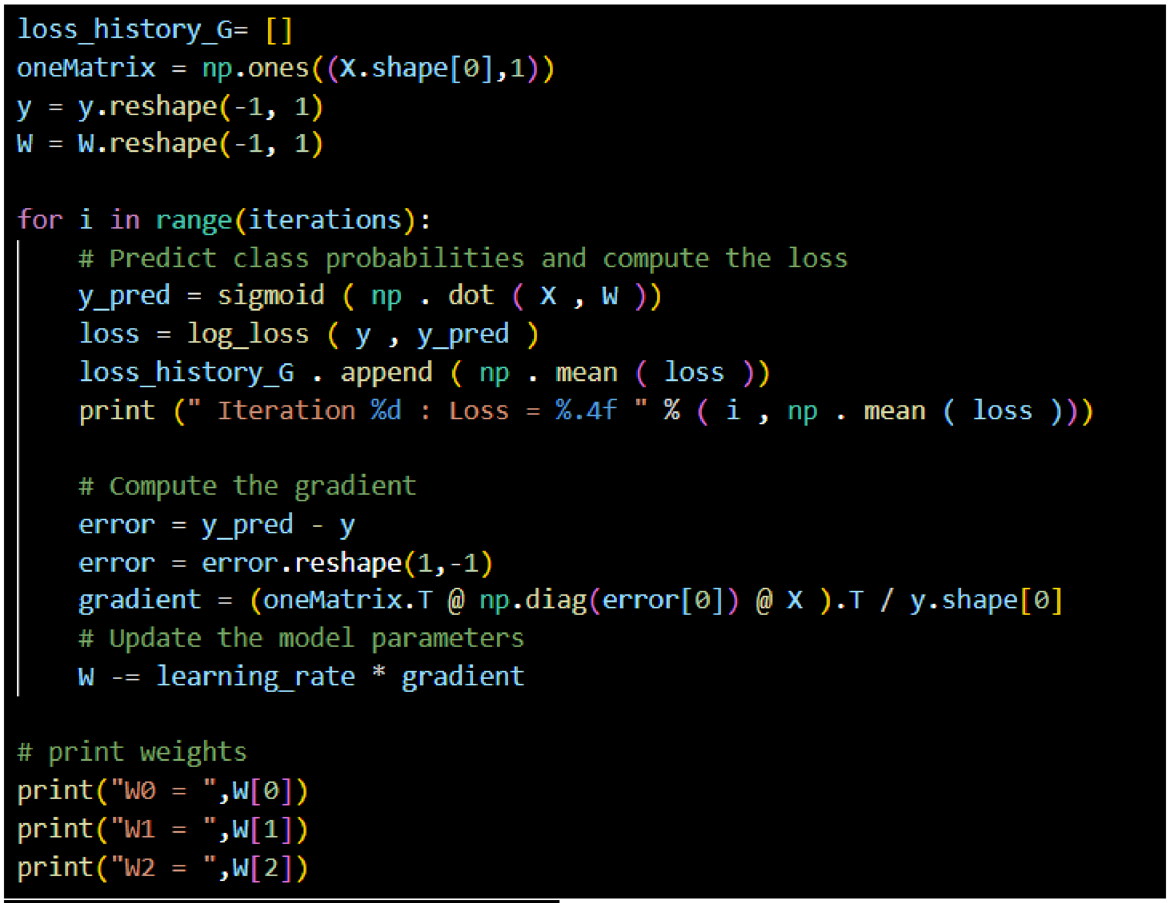
RANAVIRAJA R.W.H.M.T.A. - Index No. 200520X

GitHub: GitHub

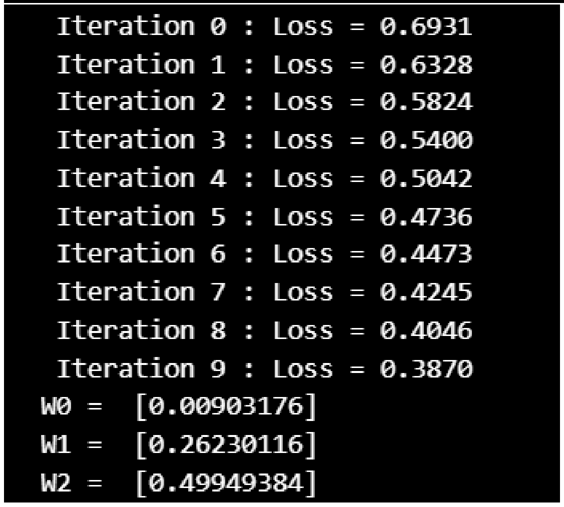
2nd October 2023

1. Logistic regression weight update process

2)

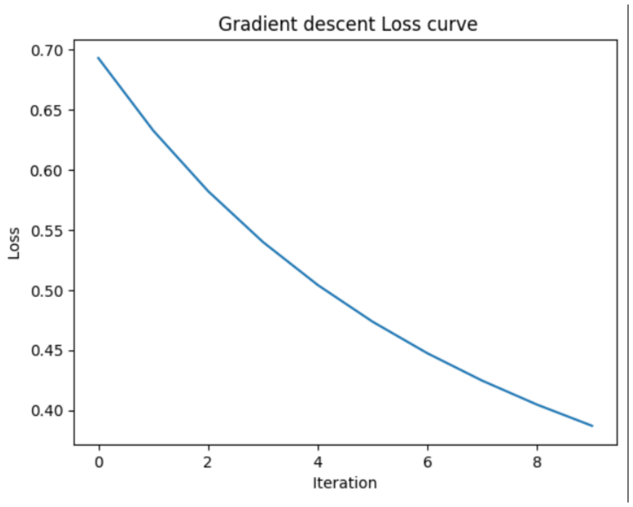


```python
loss_history_G= []
oneMatrix = np.ones((X.shape[0],1))
y = y.reshape(-1, 1)
W = W.reshape(-1, 1)

for i in range(iterations):
    # Predict class probabilities and compute the loss
    y_pred = sigmoid ( np . dot ( X , W ))
    loss = log_loss ( y , y_pred )
    loss_history_G . append ( np . mean ( loss ))
    print (" Iteration %d : Loss = %.4f " % ( i , np . mean ( loss )))

    # Compute the gradient
    error = y_pred - y
    error = error.reshape(1,-1)
    gradient = (oneMatrix.T @ np.diag(error[0]) @ X ).T / y.shape[0]
    # Update the model parameters
    W -= learning_rate * gradient

# print weights
print("W0 = ",W[0])
print("W1 = ",W[1])
print("W2 = ",W[2])
```

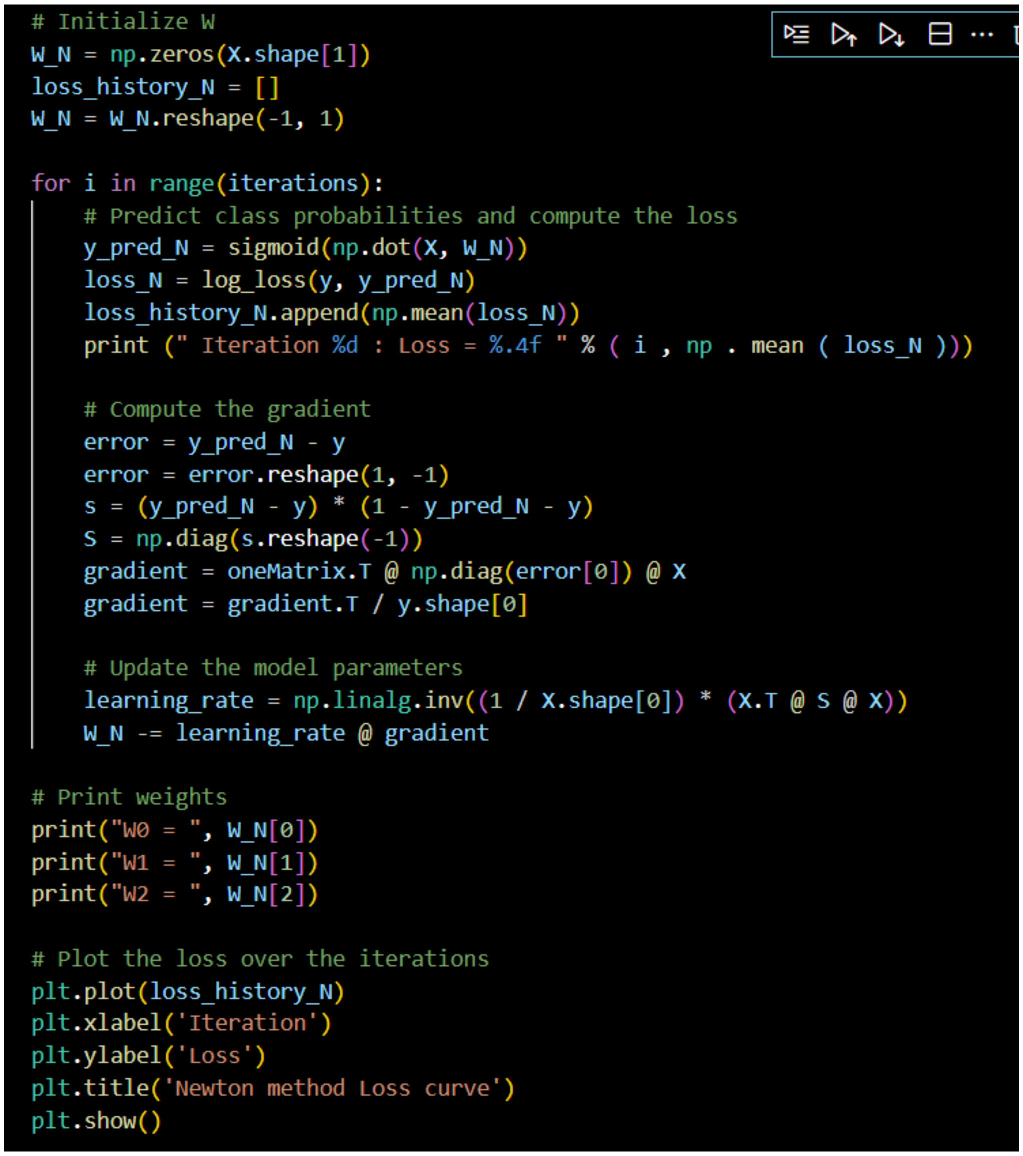


```
   Iteration 0 : Loss = 0.6931
   Iteration 1 : Loss = 0.6328
   Iteration 2 : Loss = 0.5824
   Iteration 3 : Loss = 0.5400
   Iteration 4 : Loss = 0.5042
   Iteration 5 : Loss = 0.4736
   Iteration 6 : Loss = 0.4473
   Iteration 7 : Loss = 0.4245
   Iteration 8 : Loss = 0.4046
   Iteration 9 : Loss = 0.3870
  W0 =  [0.00903176]
  W1 =  [0.26230116]
  W2 =  [0.49949384]
```

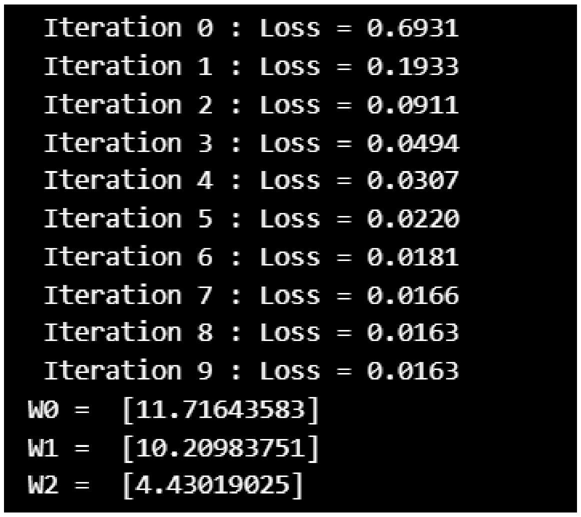
3)



4).

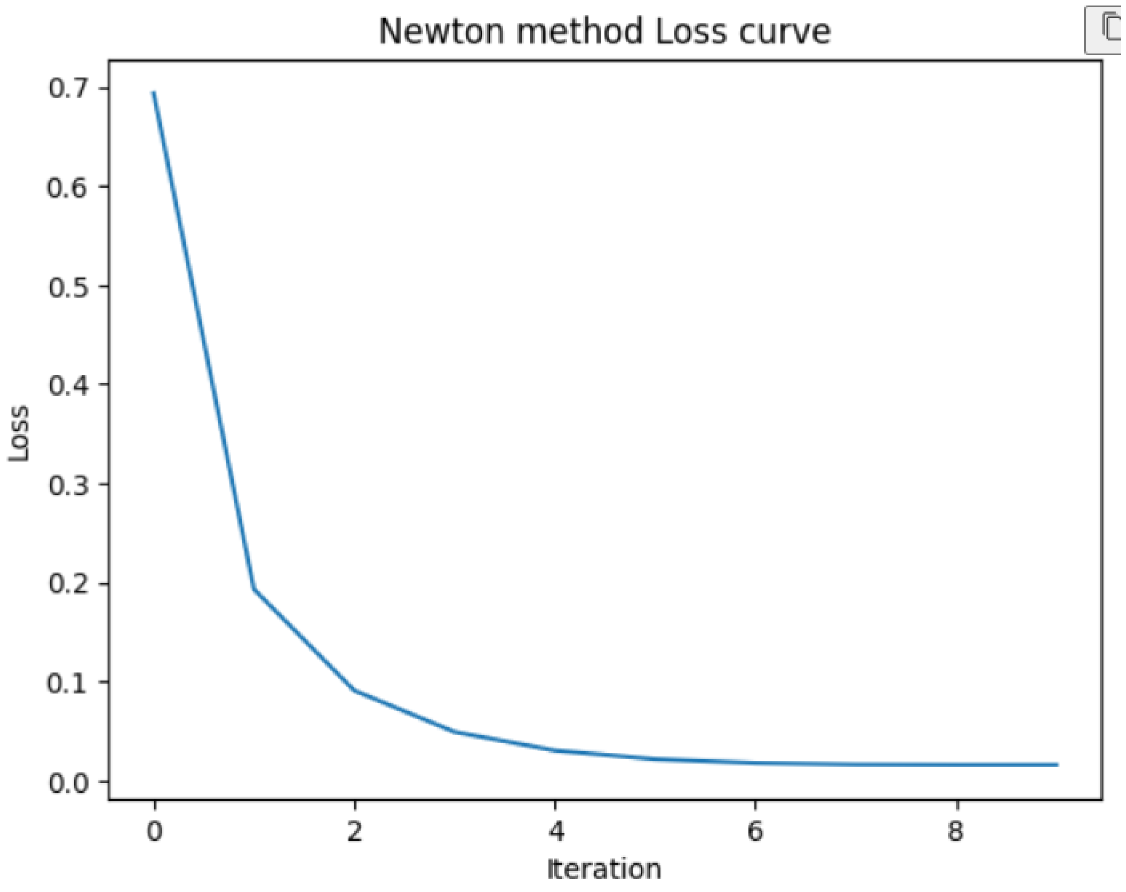


```
# Initialize W
W_N = np.zeros(X.shape[1])
loss_history_N = []
W_N = W_N.reshape(-1, 1)

for i in range(iterations):
    # Predict class probabilities and compute the loss
    y_pred_N = sigmoid(np.dot(X, W_N))
    loss_N = log_loss(y, y_pred_N)
    loss_history_N.append(np.mean(loss_N))
    print (" Iteration %d : Loss = %.4f " % ( i , np . mean ( loss_N )))

    # Compute the gradient
    error = y_pred_N - y
    error = error.reshape(1, -1)
    s = (y_pred_N - y) * (1 - y_pred_N - y)
    S = np.diag(s.reshape(-1))
    gradient = oneMatrix.T @ np.diag(error[0]) @ X
    gradient = gradient.T / y.shape[0]

    # Update the model parameters
    learning_rate = np.linalg.inv((1 / X.shape[0]) * (X.T @ S @ X))
    W_N -= learning_rate @ gradient

# Print weights
print("W0 = ", W_N[0])
print("W1 = ", W_N[1])
print("W2 = ", W_N[2])

# Plot the loss over the iterations
plt.plot(loss_history_N)
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.title('Newton method Loss curve')
plt.show()
```

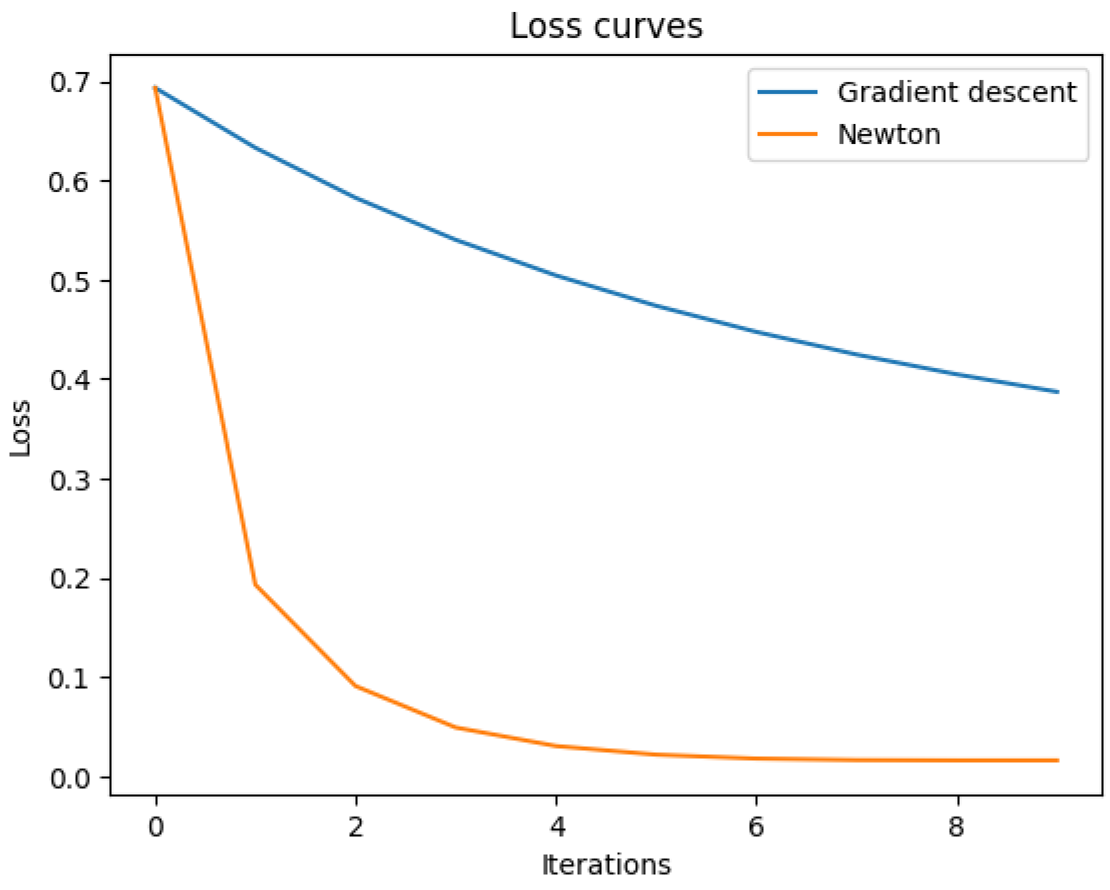


```
  Iteration 0 : Loss = 0.6931
  Iteration 1 : Loss = 0.1933
  Iteration 2 : Loss = 0.0911
  Iteration 3 : Loss = 0.0494
  Iteration 4 : Loss = 0.0307
  Iteration 5 : Loss = 0.0220
  Iteration 6 : Loss = 0.0181
  Iteration 7 : Loss = 0.0166
  Iteration 8 : Loss = 0.0163
  Iteration 9 : Loss = 0.0163
W0 =  [11.71643583]
W1 =  [10.20983751]
W2 =  [4.43019025]
```

5).



6).

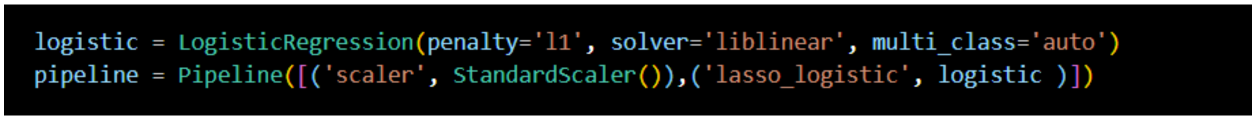


In Newton's Method Convergence is significantly faster than the Gradient Descent Convergence. Gradient Descent Convergence is smoothly decreasing in each iteration. In Newton's Method Convergence it converges to lower loss value than the Gradient Descent Convergence loss value.

2. Perform grid search for hyper-parameter tuning

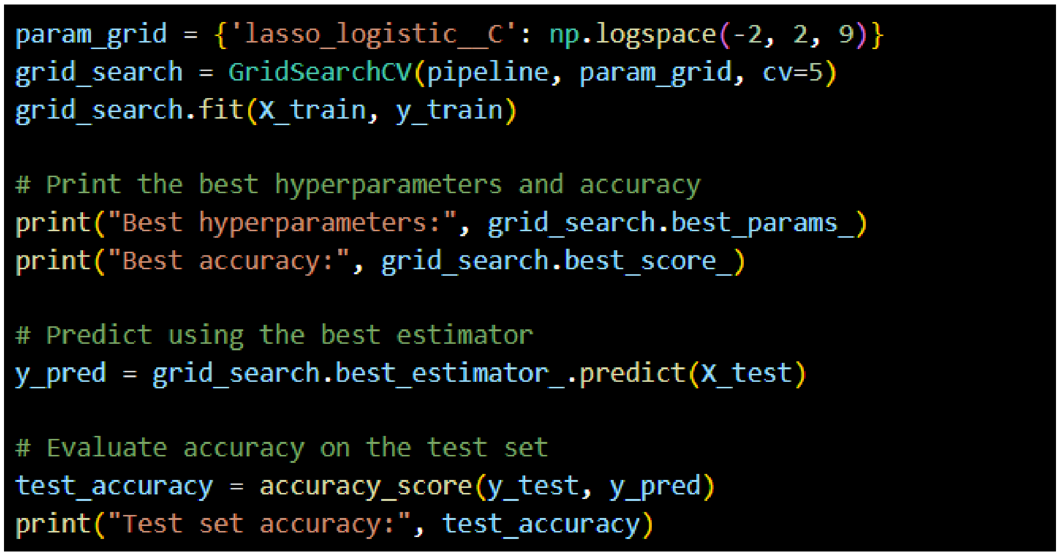
2). The purpose of "X = X[permutation]" and " y = y[permutation]" is to shuffle the rows of the X array and Y array randomly. This helps to reorder the data set in random manner. It prevents the test and training data from become biased.

3).

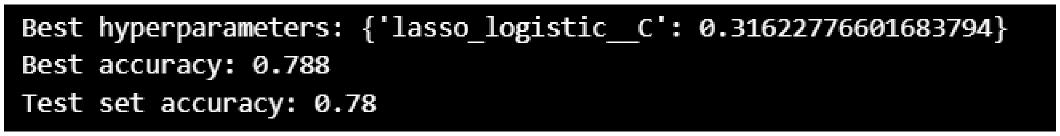


```
logistic = LogisticRegression(penalty='l1', solver='liblinear', multi_class='auto')
pipeline = Pipeline([('scaler', StandardScaler()),('lasso_logistic', logistic )])
```

4).

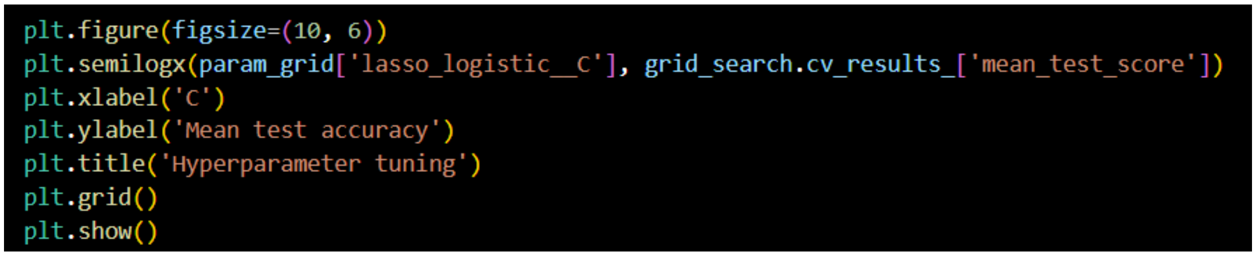


```
param_grid = {'lasso_logistic__C': np.logspace(-2, 2, 9)}
grid_search = GridSearchCV(pipeline, param_grid, cv=5)
grid_search.fit(X_train, y_train)

# Print the best hyperparameters and accuracy
print("Best hyperparameters:", grid_search.best_params_)
print("Best accuracy:", grid_search.best_score_)

# Predict using the best estimator
y_pred = grid_search.best_estimator_.predict(X_test)

# Evaluate accuracy on the test set
test_accuracy = accuracy_score(y_test, y_pred)
print("Test set accuracy:", test_accuracy)
```

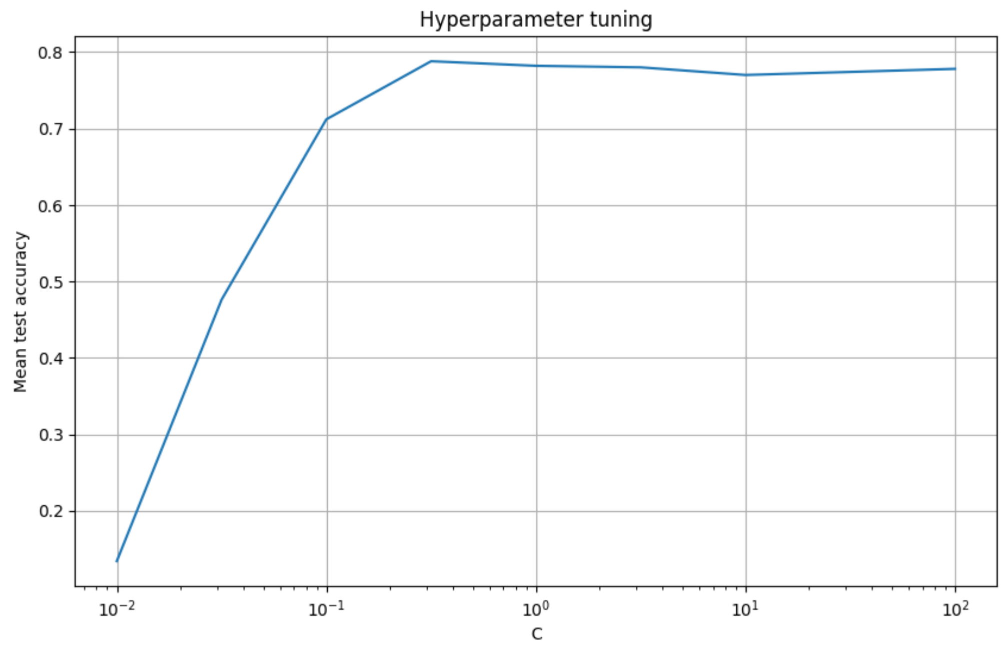


```
Best hyperparameters: {'lasso_logistic__C': 0.31622776601683794}
Best accuracy: 0.788
Test set accuracy: 0.78
```

5).

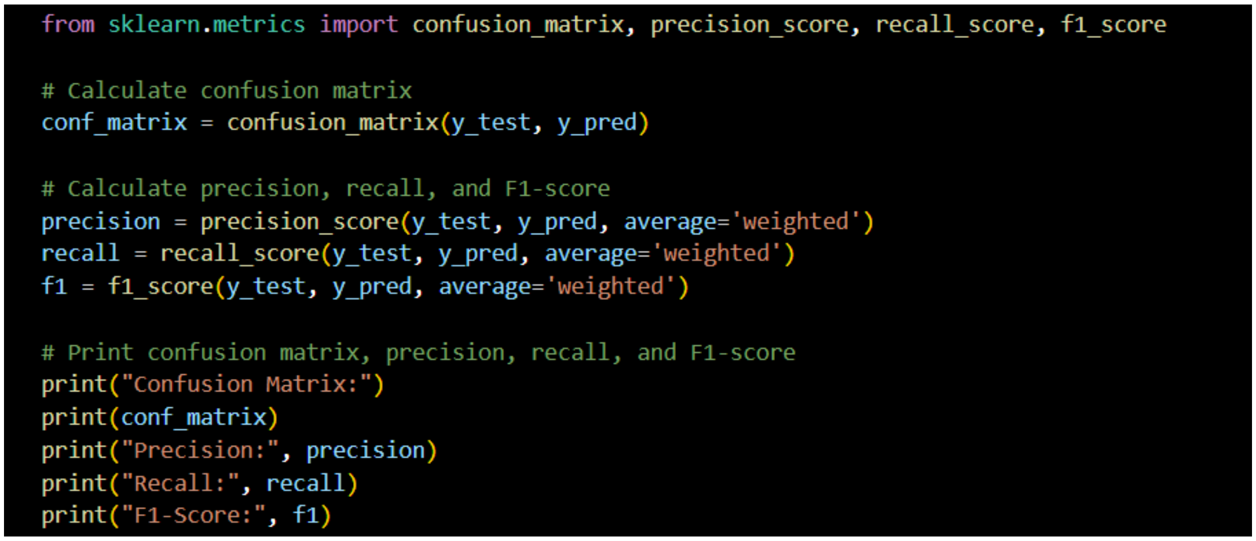


```
plt.figure(figsize=(10, 6))
plt.semilogx(param_grid['lasso_logistic__C'], grid_search.cv_results_['mean_test_score'])
plt.xlabel('C')
plt.ylabel('Mean test accuracy')
plt.title('Hyperparameter tuning')
plt.grid()
plt.show()
```

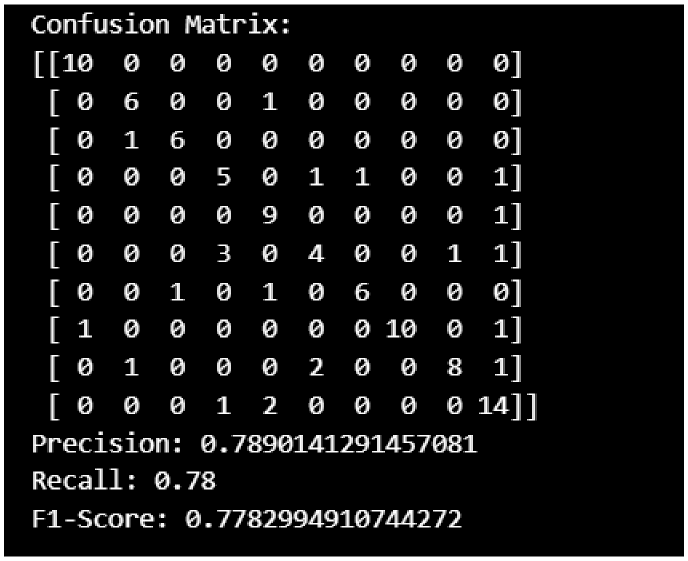


In the first part of the graph the accuracy of the classification increase with the c value. But after certain value (c= 0.31622776601683794) it starts to decrease. Therefore, the best accuracy given when its value at the highest point.

6).



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

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

# Calculate precision, recall, and F1-score
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')

# Print confusion matrix, precision, recall, and F1-score
print("Confusion Matrix:")
print(conf_matrix)
print("Precision:", precision)
print("Recall:", recall)
print("F1-Score:", f1)
```



```
Confusion Matrix:
[[10  0  0  0  0  0  0  0  0  0]
 [ 0  6  0  0  1  0  0  0  0  0]
 [ 0  1  6  0  0  0  0  0  0  0]
 [ 0  0  0  5  0  1  1  0  0  1]
 [ 0  0  0  0  9  0  0  0  0  1]
 [ 0  0  0  3  0  4  0  0  1  1]
 [ 0  0  1  0  1  0  6  0  0  0]
 [ 1  0  0  0  0  0  0 10  0  1]
 [ 0  1  0  0  0  2  0  0  8  1]
 [ 0  0  0  1  2  0  0  0  0 14]]
Precision: 0.7890141291457081
Recall: 0.78
F1-Score: 0.7782994910744272
```

This classification is contained with 10-classes. That is why the confusion matrix contain 10 x 10 matrix. The rows represent the actual classes, while the columns represent the predicted classes.

Precision measures the ratio of true positives to the total number of predicted positives. A precision score of 0.789 indicates that the model is good at avoiding false positives, meaning that when it predicts a digit, it's correct about 78.9% of the time on average.

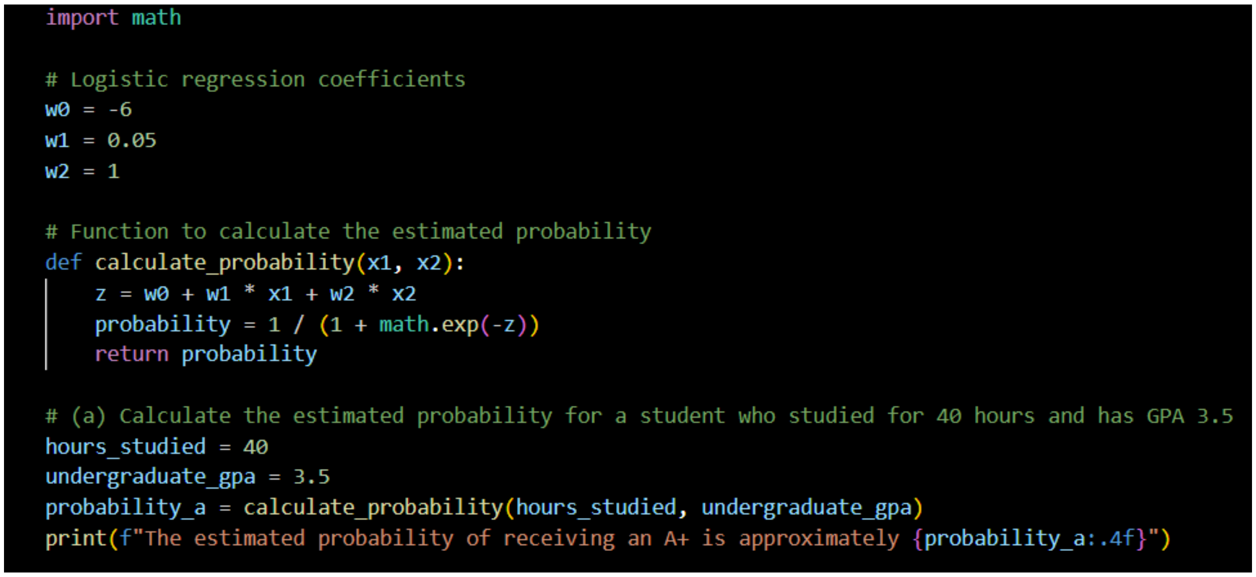
Recall measures the ratio of true positives to the total number of actual positives. A recall score of 0.78 indicates that the model is effective at identifying most of the instances of each digit, capturing about 78% of the positive instances on average.

The F1-score is the harmonic mean of precision and recall, providing a balanced measure of a model's performance. An F1-score of 0.778 suggests that the model achieves a good balance between precision and recall.

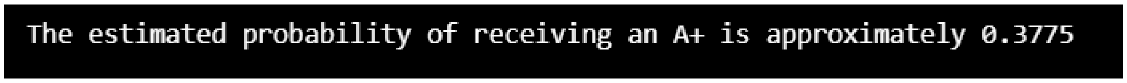
3. Logistic regression

1). (a) Z = w0 + w1 * X1 + w2 * X2

Probability = $\frac{1}{1+e^{-(Z)}}$

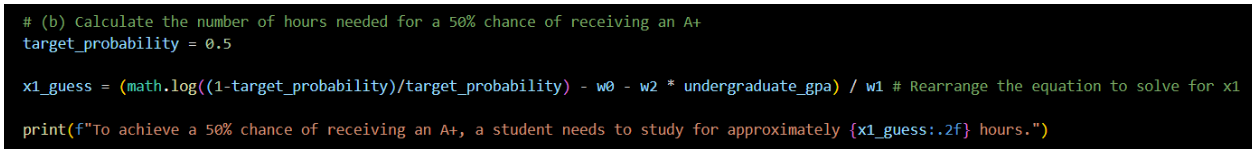


```python
import math

# Logistic regression coefficients
w0 = -6
w1 = 0.05
w2 = 1

# Function to calculate the estimated probability
def calculate_probability(x1, x2):
    z = w0 + w1 * x1 + w2 * x2
    probability = 1 / (1 + math.exp(-z))
    return probability

# (a) Calculate the estimated probability for a student who studied for 40 hours and has GPA 3.5
hours_studied = 40
undergraduate_gpa = 3.5
probability_a = calculate_probability(hours_studied, undergraduate_gpa)
print(f"The estimated probability of receiving an A+ is approximately {probability_a:.4f}")
```

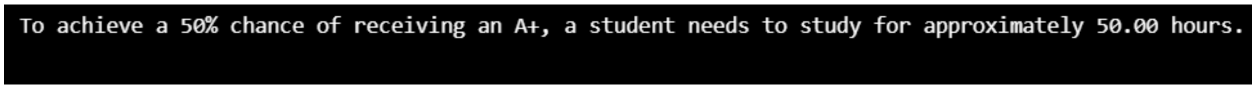


```
The estimated probability of receiving an A+ is approximately 0.3775
```

(b)



```python
# (b) Calculate the number of hours needed for a 50% chance of receiving an A+
target_probability = 0.5

x1_guess = (math.log((1-target_probability)/target_probability) - w0 - w2 * undergraduate_gpa) / w1 # Rearrange the equation to solve for x1

print(f"To achieve a 50% chance of receiving an A+, a student needs to study for approximately {x1_guess:.2f} hours.")
```



```
To achieve a 50% chance of receiving an A+, a student needs to study for approximately 50.00 hours.
```