DEPARTMENT OF ELECTRONIC AND TELECOMMUNICATION
UNIVERSITY OF MORATUWA

EN 3150: Pattern recognition

This is offered as a "EN 3150: Pattern Recognition" module's partial completion.

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

200686J : Vishagar A.

2^{nd} of October, 2023

Abstract

This report deals with the explanation of the solutions for the given questions in the assignment 01 of the EN3150 module. The solutions are explained in a way that it is easy to understand and follow. The solutions are explained with the help of the code snippets and the results. We are mainly focussed on the learning from data and related challenges and classification.

Contents

1 Logistic Regression & Weight Updating.

In this section the code was given to generate data and we are supposed to implement the Batch Gradient Descent and Newton's Method to find the optimal weights for the given data. The data was generated using the following code.

```
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 ) # Clipto 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_BGD= []
loss_history_Newton = []

one_matrix = np.ones((X.shape[0], 1))
W=W.reshape(3,1)
y=y.reshape(len(y),1)
```

1.1 Batch Gradient Descent

And after the data was generated we are supposed to implement the Batch Gradient Descent we used the following equation.

$$w_{t+1} \leftarrow w_t - \frac{\alpha}{N} \cdot (\mathbf{1}^T diag(sigm(\mathbf{w}^T(t)\mathbf{x}_i) - y_i)\mathbf{X})^T \tag{1}$$

and the implementation of the above equation and the results are shown below.

```
#performing Batch Gradient Descent
for i in range ( iterations ):
    y_pred = sigmoid ( np. dot (X , W))
    loss = log_loss ( y , y_pred )
    loss_history_BGD . append ( np. sum ( np.mean(loss) ))
    residual = y_pred - y
    residual = residual.reshape(len( residual))
    diagonal_residual = np. diag ( residual )
    gradient = one_matrix. T @ diagonal_residual@ X
    gradient = gradient.T / y . size
    W -= learning_rate * gradient
    print ( f'Iteration {i} : Loss { loss_history_BGD [i]}')
```

```
Iteration 0 : Loss 0.6931471805599454
Iteration 1 : Loss 0.6328211522065751
Iteration 2 : Loss 0.5823972949578101
Iteration 3 : Loss 0.5400400231011129
Iteration 4 : Loss 0.5042051102371915
Iteration 5 : Loss 0.4736389534183553
Iteration 6 : Loss 0.44734329598374695
Iteration 7 : Loss 0.4245298511469433
Iteration 8 : Loss 0.4045770169455826
Iteration 9 : Loss 0.38699321002389653
```

Figure 1: Loss funstions after BGD

```
plt. figure ()
plt. plot ( np. arange ( iterations ), loss_history_BGD )
plt. title ('Logistic Regression Loss per Iteration')
plt. xlabel ('Iterations')
plt. ylabel ('Loss')
plt. show ()
```

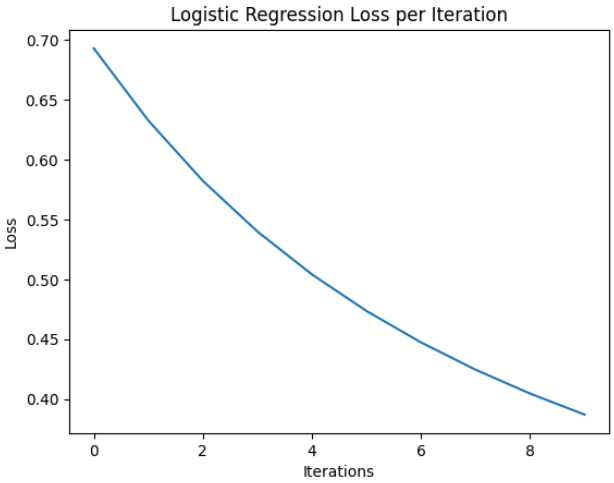

Figure 2: Loss vs Iterations

1.2 Newton's Method

And after the data was generated we are supposed to implement the Newton's Method we used the following equation.

$$w_{t+1} \leftarrow w_t - \frac{\alpha}{N} \cdot (\mathbf{1}^T diag(sigm(\mathbf{w}^T(t)\mathbf{x}_i) - y_i)\mathbf{X})^T \tag{2}$$

$$\alpha = \frac{1}{N} \cdot \mathbf{X}^T\mathbf{S}\mathbf{X} \tag{3}$$

```
# perform Newton's method
W = np. zeros (X. shape [1])
W=W.reshape(3,1)

for i in range ( iterations ):
    y_pred = sigmoid ( np. dot (X , W))
    s = ( (y_pred - y) * (1 - y_pred - y))
    s = s.reshape(len(s))
    S = np.diag(s)
    mulfact_a = X.T @ S @ X
    mulfact_a = mulfact_a / y . size
    mulfact_a = np.linalg.inv(mulfact_a)
    loss = log_loss ( y , y_pred )
    loss_history_Newton . append ( np. sum
        ( np.mean(loss) ))
    residual = y_pred - y
    residual = residual.reshape(len(
        residual))
    diagonal_residual = np. diag ( residual
         )
    mulfact_b= one_matrix. T @
        diagonal_residual@ X
    mulfact_b = mulfact_b.T / y . size
    W -= mulfact_a @ mulfact_b
    print ( f'Iteration {i} : Loss {
        loss_history_Newton [i]}')
```

```
Iteration 0 : Loss 0.6931471805599454
Iteration 1 : Loss 0.19326424132410205
Iteration 2 : Loss 0.09113723316903423
Iteration 3 : Loss 0.04938058810733571
Iteration 4 : Loss 0.030716074598479824
Iteration 5 : Loss 0.022022549931970507
Iteration 6 : Loss 0.01807356045399721
Iteration 7 : Loss 0.016618032722628157
Iteration 8 : Loss 0.0163234577557707
Iteration 9 : Loss 0.01630719622923481
```

Figure 3: Loss functions after Newton's Method

```
# Plot the loss function over iterations
plt. figure ()
plt. plot ( np. arange ( iterations ),
    loss_history_Newton )
plt. title ('Logistic Regression Loss per
    Iteration')
plt. xlabel ('Iterations')
plt. ylabel ('Loss')
plt. show ()
```

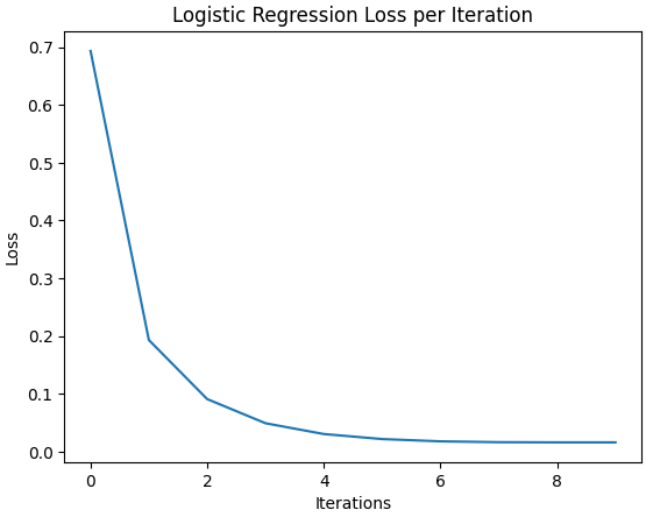

Figure 4: Loss vs Iterations

1.3 Comparison between the two methods

- In the Newton's method the learnig rate changes with the iteration and in the Batch Gradient Descent the learning rate is constant.
- The Newton's method converges to the optimal value faster than the Batch Gradient Descent.

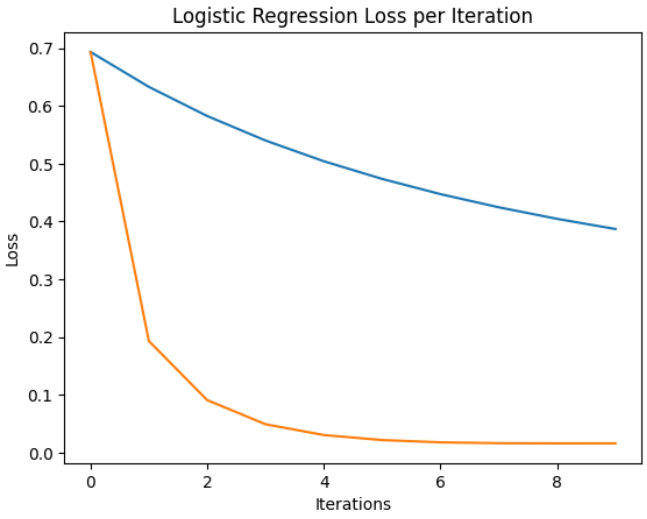

Figure 5: Comparison of the two methods

2 Grid search for hyperparameter tuning

In this section we are supposed to perform a logistic regression for image classification, create a pipeline, define a parameter grid for hyper parameter tuning and perform a grid search for the hyper parameter tuning. The code for the above mentioned tasks are shown below.

```
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)
```

2.1 Usage of Permutation

In this section we used the permutation to generate random indices to generate new order of the data by shuffling the data rows randomly.

We used this permutation function for both X and Y where we can ensure that we will get a new order of the data for both X and Y.

And also we used this permutation to avoid order based bias in the data which could lead us to poor generalization of the model.

2.2 Usage of Pipeline and GridSearch

Here we were supposed to use lasso logistic regression for image classification and create a pipeline that includes the scaling and the Lasso logistic regression estimator.

And then we were asked to define a parameter grid for hyper parameter tuning and perform a grid search for the hyper parameter tuning. we selected 9 equally spaced values staring from 0.01 to 100. (-2,2 in log scale)

We used the following code to create the pipeline.

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


param_grid = {'lasso_logistic__C': np.
    logspace(-2, 2, 9)}
#Initialize GridSearchCV
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)

# Predict on the test set
y_pred = grid_search.predict(X_test)

# Evaluate accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Test Accuracy:", accuracy)
```

And got the best value for and the releated test accuracy as,

- Best value for C : 0.31622776601683794
- Test accuracy : 0.85

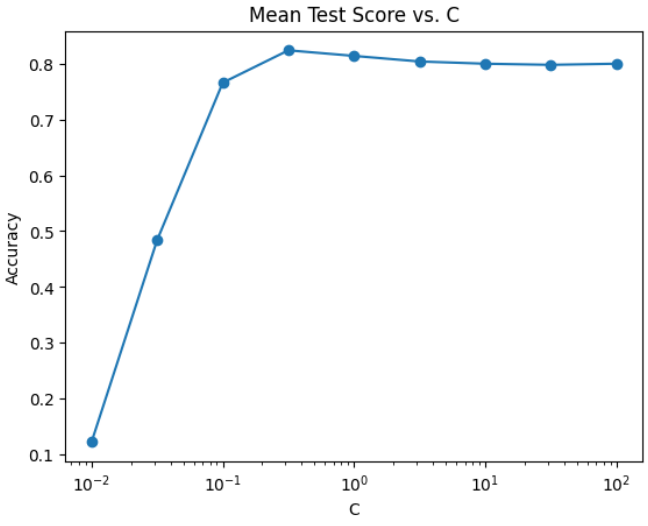

Figure 6: Mean test score vs C

And for the evaluation metrics we used **Precision**, **Recall**, **confusion matrix** and **F1 score** and following were the relevant codes and results.

```
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)
accuracy = accuracy_score(y_test, y_pred)
confusion_matrix = confusion_matrix(y_test,
     y_pred)
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 the metrics
print('Accuracy:', accuracy)
print('Confusion Matrix:\n',
    confusion_matrix)
print('Precision:', precision)
print('Recall:', recall)
print('F1 Score:', f1)

# 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()
```

```
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
```

Figure 7: Evaluation metrics

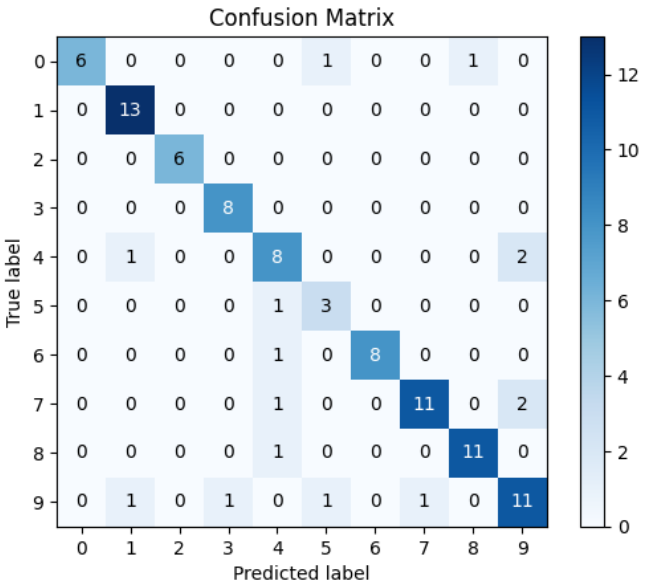

Figure 8: Confusion matrix

Considering the above results we can say that the model is performing well and the model is not overfitting or underfitting. Because the **accuracy of the model is high** and the **precision, recall and the F1 score are also high**. And also the **confusion matrix** shows that the model is performing well by only predicting the correct classes.

3 Logistic Regression

- x_1 = Number of Hours studied
- x_2 = Student's GPA
- y = 1 if the student Got A^+ and 0 otherwise
- w_0 = -6
- w_1 = 0.05
- w_2 = 1

The Regression model,

$$Pr(y) = \frac{1}{1 + e^{-(w_0 + w_1 x_1 + w_2 x_2)}} \tag{4}$$

therefore,

3.1 Answer for (a)

- $x_1 = 40$
- $x_2 = 3.5$

$$Pr(y) = \frac{1}{1 + e^{-(w_0 + w_1 x_1 + w_2 x_2)}} \tag{5}$$

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

$$Pr(y) = \frac{1}{1 + e^{-(-6 + 2 + 3.5)}} \tag{7}$$

$$Pr(y) = \frac{1}{1 + e^{(0.5)}} \tag{8}$$

$$Pr(y) = \frac{1}{1 + 1.648721} \tag{9}$$

$$Pr(y) = 0.37754 \tag{10}$$

3.2 Answer for (b)

- Pr(y) = 0.5
- $x_2 = 3.5$

$$Pr(y) = \frac{1}{1 + e^{-(w_0 + w_1 x_1 + w_2 x_2)}} \tag{11}$$

$$0.5 = \frac{1}{1 + e^{-(w_0 + w_1 x_1 + w_2 x_2)}} \tag{12}$$

$$e^{-(w_0 + w_1 x_1 + w_2 x_2)} = 1 \tag{13}$$

$$w_0 + w_1 x_1 + w_2 x_2 = 0 \tag{14}$$

$$-6 + 0.05 \times x_1 + 1 \times 3.5 = 0 \tag{15}$$

$$x_1 = 50 \tag{16}$$

4 References

- Sci-kit Learn Documentation on Logistic regression
- Pipelining
- Grid Search

5 Github Repository

Following is the link to my Github repository for this assignment.

Github/EN3150_Assignment_02