COMP 9417
Machine Learning and Data Mining

Homework 2

Student Name	Hongyi Luo
Student Number	z5241868

Question1

(a) $L(\hat{\beta}_0, \hat{\beta}) = \sum_{i=1}^{n} y_i \ln\left(\frac{1}{s(\beta_0+\beta^T x_i)}\right) + (1-y_i)\ln\left(\frac{1}{1-s(\beta_0+\beta^T x_i)}\right)$

$\hat{\omega}, \hat{c} = arg \min_{\omega,c}\{\|\omega\|_1 + C\sum_{i=1}^{n}\log(1+\exp(-\tilde{y}_i(\omega^T x_i + c)))\}$ $\quad (\hat{\beta}_0, \hat{\beta}) = arg \min_{\beta_0,\beta}\{CL(\hat{\beta}_0, \hat{\beta}) + penalty(\beta)\}$

From the question, we can get the information that the $penalty(\beta) = \|\beta\|_1$. This question aims to prove that $\hat{\omega}, \hat{c} = (\hat{\beta}_0, \hat{\beta})$. These 2 equations have the same part which is $penalty(\beta) = \|\beta\|_1$. Therefore, we can prove $C\sum_{i=1}^{n}\log(1+\exp(-\tilde{y}_i(\omega^T x_i + c))) = CL(\hat{\beta}_0, \hat{\beta})$, which is also $\sum_{i=1}^{n}\log(1+\exp(-\tilde{y}_i(\omega^T x_i + c))) = L(\hat{\beta}_0, \hat{\beta})$, instead of checking the whole equation. Also, from the question, we can get that $y_i \in \{0,1\}$ in $L(\hat{\beta}_0, \hat{\beta})$, as well as $\tilde{y}_i \in \{-1,1\}$ in $\hat{\omega}, \hat{c}$, whereas $y_i \in \{0,1\}$. This means we can discuss in 2 situations. The first situation is when $\tilde{y}_i = -1\ and\ y_i = 0$. The second situation is when $\tilde{y}_i = 1\ and\ y_i = 1$.

For the first situation which is $\tilde{y}_i = -1\ and\ y_i = 0$. Put both y_i and $\tilde{y}_i$ into equations $\sum_{i=1}^{n}\log(1+\exp(-\tilde{y}_i(\omega^T x_i + c)))$ and $L(\hat{\beta}_0, \hat{\beta}) = \sum_{i=1}^{n} y_i \ln\left(\frac{1}{s(\beta_0+\beta^T x_i)}\right) + (1-y_i)\ln\left(\frac{1}{1-s(\beta_0+\beta^T x_i)}\right)$.

After that, we can get 2 new equations as showing below.

Here is the result of using $y_i = 0$ in $L(\hat{\beta}_0, \hat{\beta})$, which is $\sum_{i=1}^{n}\ln\left(\frac{1}{1-s(\beta_0+\beta^T x_i)}\right)$.

Here is the result of using $\tilde{y}_i = -1$ in $\sum_{i=1}^{n}\log(1+\exp(-\tilde{y}_i(\omega^T x_i + c)))$, which is $\sum_{i=1}^{n}\log(1+\exp(\omega^T x_i + c))$.

Also, using the $s(z) = (1+e^{-z})^{-1}$ to simplify $\sum_{i=1}^{n}\ln\left(\frac{1}{1-s(\beta_0+\beta^T x_i)}\right)$, we can get the following equation.

$$\sum_{i=1}^{n}\ln\left(\frac{1}{1-s(\beta_0+\beta^T x_i)}\right) = \sum_{i=1}^{n}\ln\left(1+e^{(\beta_0+\beta^T x_i)}\right) = \sum_{i=1}^{n}\log(1+\exp(\beta_0+\beta^T x_i))$$

Compare these 2 equations $\sum_{i=1}^{n}\log(1+\exp(\beta_0+\beta^T x_i))$ and $\sum_{i=1}^{n}\log(1+\exp(\omega^T x_i + c))$. Finally, we can get the result that $\hat{\beta}_0 = \hat{c}$ and $\hat{\beta} = \hat{\omega}$.

Use the same way in the second situation. For the second situation which is $\tilde{y}_i = 1\ and\ y_i = 1$. Put both y_i and $\tilde{y}_i$ into equations $\sum_{i=1}^{n}\log(1+\exp(-\tilde{y}_i(\omega^T x_i + c)))$ and $L(\hat{\beta}_0, \hat{\beta}) = \sum_{i=1}^{n} y_i \ln\left(\frac{1}{s(\beta_0+\beta^T x_i)}\right) + (1-y_i)\ln\left(\frac{1}{1-s(\beta_0+\beta^T x_i)}\right)$.

After that, we can get 2 new equations as showing below.

Here is the result of using $y_i = 1$ in $L(\hat{\beta}_0, \hat{\beta})$, which is $\sum_{i=1}^{n}\ln\left(\frac{1}{s(\beta_0+\beta^T x_i)}\right)$.

Here is the result of using $\tilde{y}_i = 1$ in $\sum_{i=1}^{n}\log(1+\exp(-\tilde{y}_i(\omega^T x_i + c)))$, which is $\sum_{i=1}^{n}\log(1-\exp(\omega^T x_i + c))$.

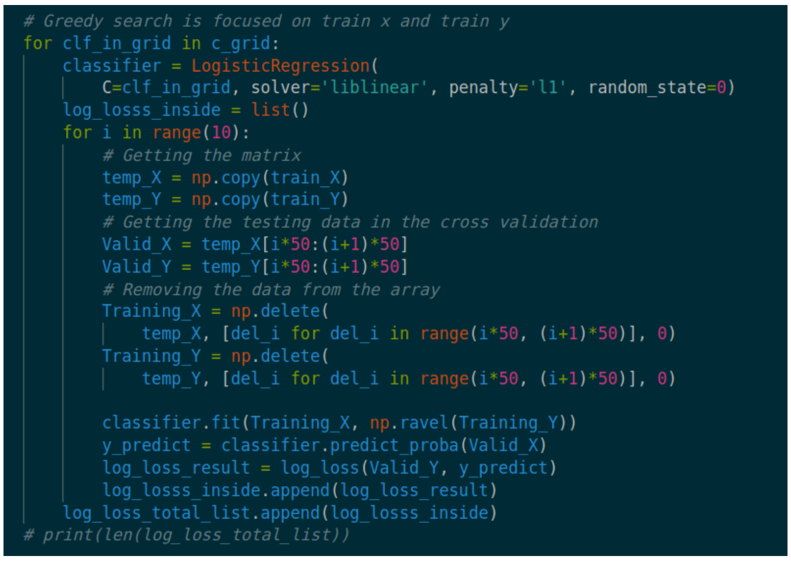
Also, using the $s(z) = (1+e^{-z})^{-1}$ to simplify $\sum_{i=1}^{n}\ln\left(\frac{1}{s(\beta_0+\beta^T x_i)}\right)$, we can get the following equation.

$$\sum_{i=1}^{n}\ln\left(\frac{1}{s(\beta_0+\beta^T x_i)}\right) = \sum_{i=1}^{n}\ln\left(1+e^{-(\beta_0+\beta^T x_i)}\right) = \sum_{i=1}^{n}\log(1-\exp(\beta_0+\beta^T x_i))$$

Compare these 2 equations $\sum_{i=1}^{n}\log(1-\exp(\beta_0+\beta^T x_i))$ and $\sum_{i=1}^{n}\log(1-\exp(\omega^T x_i + c))$. Finally, we can get the result that $\hat{\beta}_0 = \hat{c}$ and $\hat{\beta} = \hat{\omega}$.

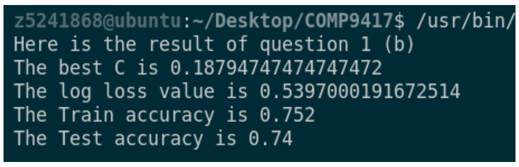
The parameter C in these equations is to balance the relationship between the complexity and accuracy of the model which is the same as the function of λ in LASSO. There are also different points between them. λ is the parameter of the penalty function in LASSO, which is $\lambda penalty(\beta)$. This means when λ goes larger, we focus more on the penalty function. However, question C is focused on $L\left(\hat{\beta}_0, \hat{\beta}\right)$. This means the function between C and λ is opposite, when C goes larger λ goes smaller. Otherwise, when C goes smaller λ goes larger.

(b) This part of the picture of code contains the implementation of cross-validation and the model training. Some data processing and C list generation can also be found in the python files submitted. Here I will only show you the main part of the code in question1 (b) as the following picture given below.



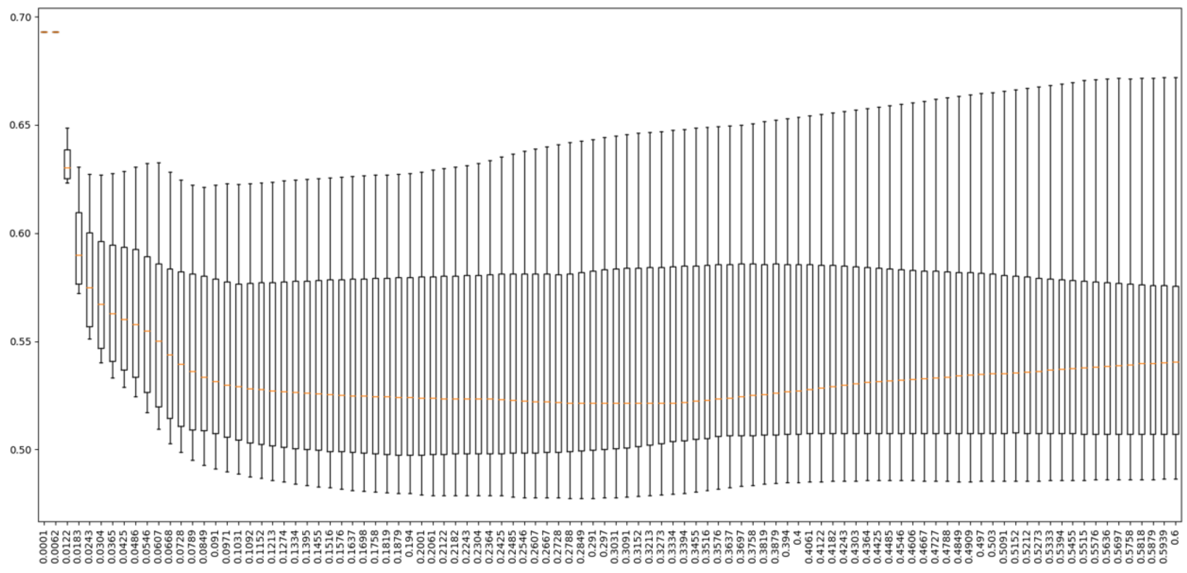
```python
# Greedy search is focused on train x and train y
for clf_in_grid in c_grid:
    classifier = LogisticRegression(
        C=clf_in_grid, solver='liblinear', penalty='l1', random_state=0)
    log_losss_inside = list()
    for i in range(10):
        # Getting the matrix
        temp_X = np.copy(train_X)
        temp_Y = np.copy(train_Y)
        # Getting the testing data in the cross validation
        Valid_X = temp_X[i*50:(i+1)*50]
        Valid_Y = temp_Y[i*50:(i+1)*50]
        # Removing the data from the array
        Training_X = np.delete(
            temp_X, [del_i for del_i in range(i*50, (i+1)*50)], 0)
        Training_Y = np.delete(
            temp_Y, [del_i for del_i in range(i*50, (i+1)*50)], 0)

        classifier.fit(Training_X, np.ravel(Training_Y))
        y_predict = classifier.predict_proba(Valid_X)
        log_loss_result = log_loss(Valid_Y, y_predict)
        log_losss_inside.append(log_loss_result)
    log_loss_total_list.append(log_losss_inside)
# print(len(log_loss_total_list))
```

By running this part of the code, we can attain the log loss result from different C values. Therefore, we can not only generate the plot but also get the best C value. The best C is the minimum value in the result list. From the picture, the best C value is 0.18794747474747472. The log loss value is 0.5397000191672514. The Train accuracy is 0.752. The test accuracy is 0.74.

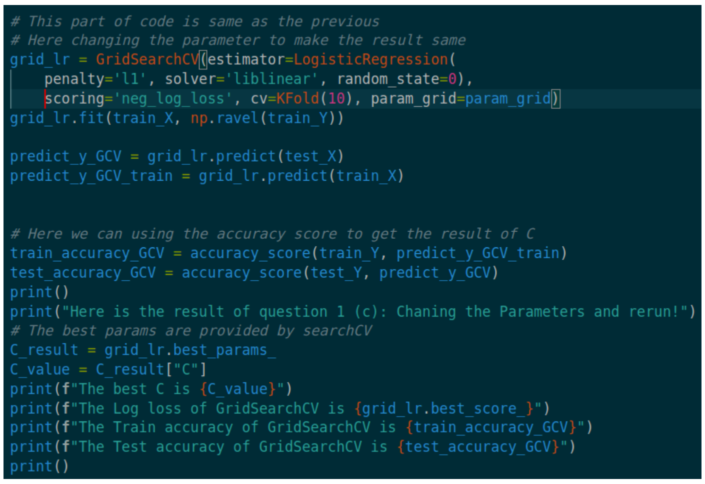


```
z5241868@ubuntu:~/Desktop/COMP9417$ /usr/bin/
Here is the result of question 1 (b)
The best C is 0.18794747474747472
The log loss value is 0.5397000191672514
The Train accuracy is 0.752
The Test accuracy is 0.74
```

By using the log loss result list, we can finally generate the plot in question1 (b) as showing below.

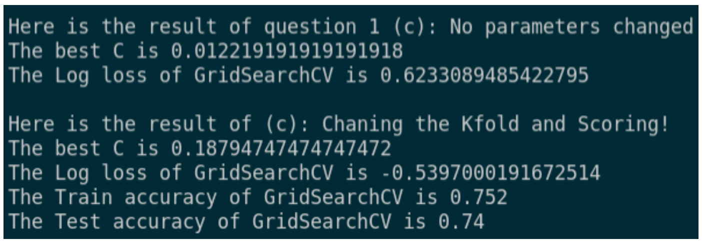


(c) Here is the code after changing parameters in the GridSearchCV function. Notice here, I adding the scoring parameter as log loss. Furthermore, for the cross-validation function, I used the K-fold method which K is 10 in part of code.



```python
# This part of code is same as the previous
# Here changing the parameter to make the result same
grid_lr = GridSearchCV(estimator=LogisticRegression(
    penalty='l1', solver='liblinear', random_state=0),
    scoring='neg_log_loss', cv=KFold(10), param_grid=param_grid)
grid_lr.fit(train_X, np.ravel(train_Y))

predict_y_GCV = grid_lr.predict(test_X)
predict_y_GCV_train = grid_lr.predict(train_X)


# Here we can using the accuracy score to get the result of C
train_accuracy_GCV = accuracy_score(train_Y, predict_y_GCV_train)
test_accuracy_GCV = accuracy_score(test_Y, predict_y_GCV)
print()
print("Here is the result of question 1 (c): Chaning the Parameters and rerun!")
# The best params are provided by searchCV
C_result = grid_lr.best_params_
C_value = C_result["C"]
print(f"The best C is {C_value}")
print(f"The Log loss of GridSearchCV is {grid_lr.best_score_}")
print(f"The Train accuracy of GridSearchCV is {train_accuracy_GCV}")
print(f"The Test accuracy of GridSearchCV is {test_accuracy_GCV}")
print()
```

Here is the difference between changing parameters or not. In the following picture, the first part is no changes in the parameter. We can find that this result is different from question1 (b). After changing the parameter of scoring(log-loss) and cross-validation (K-fold, K=10), we can get the same result as question1 (b).

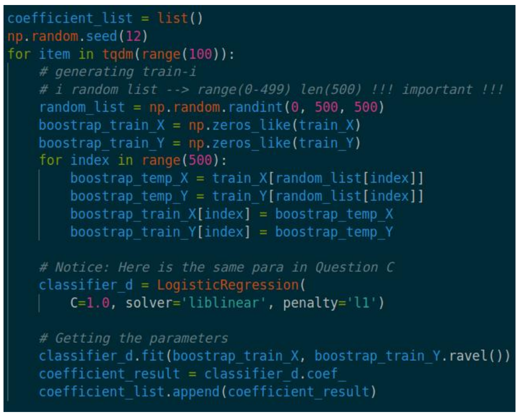


```
Here is the result of question 1 (c): No parameters changed
The best C is 0.012219191919191918
The Log loss of GridSearchCV is 0.6233089485422795

Here is the result of (c): Chaning the Kfold and Scoring!
The best C is 0.18794747474747472
The Log loss of GridSearchCV is -0.5397000191672514
The Train accuracy of GridSearchCV is 0.752
The Test accuracy of GridSearchCV is 0.74
```

This part is to analyze the reason why they are different. This may cause by 2 factors, which are data segmentation standard and scoring standard. For the previous part in (b), we do the data segmentation without considering the label of the data. But the default method of parameter cross-validation is Stratified K-Fold. Stratified K-Fold standard is based on the data distribution. This can make the target proportion of the divided data set approximate to the original data set, which is different from the K-fold method divide the data directly. This is the first reason.

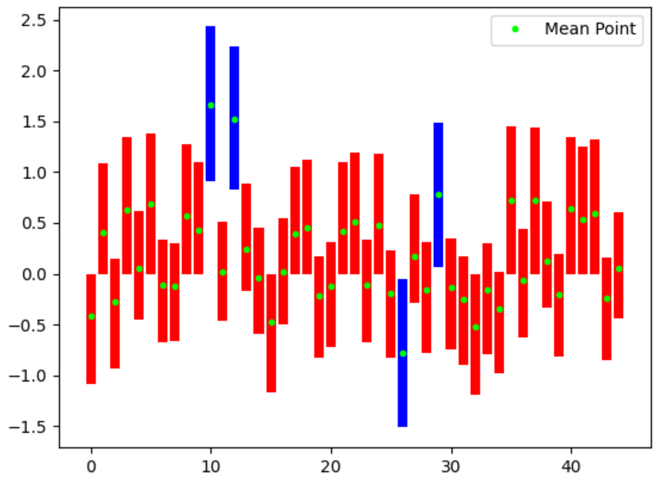
Consider the scoring standard, the default value is none, which means there is no scoring strategy. However, in question1 (b), we use the log loss function as the scoring standard. Therefore, to make it the same, we should take the same scoring standard with question1 (b) instead of using nothing. Therefore, here are 2 reasons which caused the different results with question1 (b).

(d) There is the code capture of quesion1 (d). This part of the code contains the implementation of the bootstrap method as well as the implementation of the logistic regression model.



```python
coefficient_list = list()
np.random.seed(12)
for item in tqdm(range(100)):
    # generating train-i
    # i random list --> range(0-499) len(500) !!! important !!!
    random_list = np.random.randint(0, 500, 500)
    boostrap_train_X = np.zeros_like(train_X)
    boostrap_train_Y = np.zeros_like(train_Y)
    for index in range(500):
        boostrap_temp_X = train_X[random_list[index]]
        boostrap_temp_Y = train_Y[random_list[index]]
        boostrap_train_X[index] = boostrap_temp_X
        boostrap_train_Y[index] = boostrap_temp_Y

    # Notice: Here is the same para in Question C
    classifier_d = LogisticRegression(
        C=1.0, solver='liblinear', penalty='l1')

    # Getting the parameters
    classifier_d.fit(boostrap_train_X, boostrap_train_Y.ravel())
    coefficient_result = classifier_d.coef_
    coefficient_list.append(coefficient_result)
```

After getting the coefficient list, we can draw the plot as showing below. The lime dot on the plot is the mean value of each bar. From this image, we can find that there are only four blue bars.



(e) Based on the bar plot given by question1 d, we can get the following results.

The number of red bars is more than the blue bar. Based on the theory of confidence interval, the red bar contains 0 points. This means the feature of the red bar is not important as the blue bar. In other words, the feature of blue bars is more important than the red bars. Also, if the bar is short and gathering near the zero points, this means these features are not useful than others.

As for the value C, this value is a kind of constraint of features. As for the value C, this value is a kind of constraint of features. If the current C value is small, this means for this feature we will add more penalties on it. This will cause a large number of features which are in red bars. Otherwise, for those features whose C value is large, this means the constraint is little. Furthermore, fewer constraints will cause fewer penalties. Finally, doing the regularization is important and necessary. This is because the number of red bars which are unimportant features is more than blue features. Therefore, reduce the influence brought by unimportant features is necessary to do the regularization.

Question2

(a) In the question a, the learning rate $\alpha = 0.1$. Based on the question, we can get the following equation to do the calculation.

$$f(x) = \frac{1}{2}\|Ax - b\|_2^2 = \frac{1}{2}(Ax - b)^T(Ax - b)$$

$$\nabla f(x) = A^T(Ax - b)$$

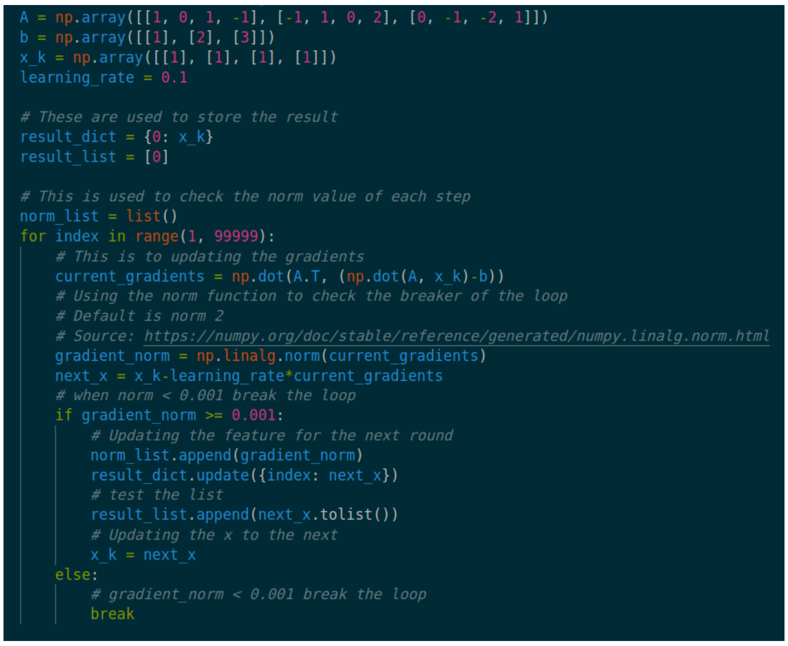
$$\nabla f\left(x^{(k)}\right) = A^T\left(Ax^{(k)} - b\right)$$

Therefore, for $x^{(k)}$ getting each gradient as the equation showing above.

Then, using the learning rate to update the $x^{(k)}$ as the equation showing below.

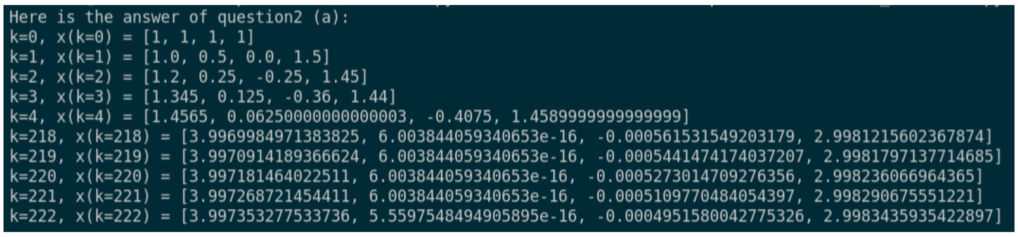
$$x^{(k)} = x^{(k-1)} - \alpha\nabla f\left(x^{(k-1)}\right)$$

After getting this equation, we can set it up in the python code as showing below.



```python
A = np.array([[1, 0, 1, -1], [-1, 1, 0, 2], [0, -1, -2, 1]])
b = np.array([[1], [2], [3]])
x_k = np.array([[1], [1], [1], [1]])
learning_rate = 0.1

# These are used to store the result
result_dict = {0: x_k}
result_list = [0]

# This is used to check the norm value of each step
norm_list = list()
for index in range(1, 99999):
    # This is to updating the gradients
    current_gradients = np.dot(A.T, (np.dot(A, x_k)-b))
    # Using the norm function to check the breaker of the loop
    # Default is norm 2
    # Source: https://numpy.org/doc/stable/reference/generated/numpy.linalg.norm.html
    gradient_norm = np.linalg.norm(current_gradients)
    next_x = x_k-learning_rate*current_gradients
    # when norm < 0.001 break the loop
    if gradient_norm >= 0.001:
        # Updating the feature for the next round
        norm_list.append(gradient_norm)
        result_dict.update({index: next_x})
        # test the list
        result_list.append(next_x.tolist())
        # Updating the x to the next
        x_k = next_x
    else:
        # gradient_norm < 0.001 break the loop
        break
```

After running the code, we can get the following result. In this picture, we can find the first and the last 5 values of $x^{(k)}$ as showing below. From this image, we can find the index of the last value is k=222. This k value is also the steps of the gradient update.



```
Here is the answer of question2 (a):
k=0, x(k=0) = [1, 1, 1, 1]
k=1, x(k=1) = [1.0, 0.5, 0.0, 1.5]
k=2, x(k=2) = [1.2, 0.25, -0.25, 1.45]
k=3, x(k=3) = [1.345, 0.125, -0.36, 1.44]
k=4, x(k=4) = [1.4565, 0.06250000000000003, -0.4075, 1.4589999999999999]
k=218, x(k=218) = [3.9969984971383825, 6.003844059340653e-16, -0.0005615315492031179, 2.9981215602367874]
k=219, x(k=219) = [3.9970914189366624, 6.003844059340653e-16, -0.0005441474174037207, 2.9981797137714685]
k=220, x(k=220) = [3.997181464022511, 6.003844059340653e-16, -0.0005273014709276356, 2.998236066964365]
k=221, x(k=221) = [3.997268721454411, 6.003844059340653e-16, -0.0005109770484054397, 2.998290675551221]
k=222, x(k=222) = [3.997353277533736, 5.5597548494905895e-16, -0.0004951580042775326, 2.9983435935422897]
```

(b) Question2 (b) is similar to question2 (a), the difference is the learning rate of question b needs to be updated. Therefore, using the given equation to get the learning rate equation. Using the gradient equation $\nabla f(x)$ in the question 1 to simplify the α_k.

$$\alpha_k = arg \min_{a \geq 0} f\left(x^{(k)} - \alpha \nabla f\left(x^{(k)}\right)\right)$$

$$g(\alpha) = \frac{1}{2}\left\|A\left(x^{(k)} - \alpha \nabla \mathrm{f}\left(x^{(k)}\right)\right) - b\right\|_2^2$$

$$= \frac{1}{2}\left(A\left(x^{(k)} - \alpha \nabla f\left(x^{(k)}\right)\right) - b\right)^T \left(A\left(x^{(k)} - \alpha \nabla f\left(x^{(k)}\right)\right) - b\right)$$

$$= \frac{1}{2}\left(Ax^{(k)} - \alpha A \nabla f\left(x^{(k)}\right) - b\right)^T \left(Ax^{(k)} - \alpha A \nabla f\left(x^{(k)}\right) - b\right)$$

$$= \frac{1}{2}\Bigg(\left(Ax^{(k)}\right)^T Ax^{(k)} - \alpha\left(A\nabla f\left(x^{(k)}\right)\right)^T Ax^{(k)} - b^T Ax^{(k)} - \left(Ax^{(k)}\right)^T \alpha A \nabla f\left(x^{(k)}\right)$$
$$+ \alpha\left(A\nabla f\left(x^{(k)}\right)\right)^T \alpha A\nabla f\left(x^{(k)}\right) + b^T \alpha A \nabla f\left(x^{(k)}\right) - \left(Ax^{(k)}\right)^T b + \alpha\left(A\nabla f\left(x^{(k)}\right)\right)^T b + b^T b\Bigg)$$

The previous simplify based on the size of the matrix. By doing this, we can get the function of α. Then doing the derivative of this equation.

$$
\begin{aligned}
g(\alpha)' = \Bigg(\frac{1}{2}\Big(&\left(Ax^{(k)}\right)^T Ax^{(k)} - \alpha\left(A\nabla f\left(x^{(k)}\right)\right)^T Ax^{(k)} - b^T Ax^{(k)} - \left(Ax^{(k)}\right)^T \alpha A\nabla f\left(x^{(k)}\right) \\
&+ \alpha\left(A\nabla f\left(x^{(k)}\right)\right)^T \alpha A\nabla f\left(x^{(k)}\right) + b^T\alpha A\nabla f\left(x^{(k)}\right) - \left(Ax^{(k)}\right)^T b + \alpha\left(A\nabla f\left(x^{(k)}\right)\right)^T b \\
&+ b^T b\Big)\Bigg)' \\
= -\frac{1}{2}\left(A\nabla f\left(x^{(k)}\right)\right)^T Ax^{(k)} &- \frac{1}{2}\left(Ax^{(k)}\right)^T A\nabla f\left(x^{(k)}\right) + \alpha\left(A\nabla f\left(x^{(k)}\right)\right)^T A\nabla f\left(x^{(k)}\right) + \frac{1}{2}b^T A\nabla f\left(x^{(k)}\right) \\
&+ \frac{1}{2}\left(A\nabla f\left(x^{(k)}\right)\right)^T b
\end{aligned}
$$

To simplify the equation of α, then let the $g(\alpha)' = 0$.

$$
\begin{aligned}
g(\alpha)' = -\frac{1}{2}\left(A\nabla f\left(x^{(k)}\right)\right)^T Ax^{(k)} - \frac{1}{2}\left(Ax^{(k)}\right)^T A\nabla f\left(x^{(k)}\right) + \alpha\left(A\nabla f\left(x^{(k)}\right)\right)^T A\nabla f\left(x^{(k)}\right) + \frac{1}{2}b^T A\nabla f\left(x^{(k)}\right) \\
+ \frac{1}{2}\left(A\nabla f\left(x^{(k)}\right)\right)^T b = 0
\end{aligned}
$$

$$
\begin{aligned}
\left(A\nabla f\left(x^{(k)}\right)\right)^T Ax^{(k)} + \left(Ax^{(k)}\right)^T A\nabla f\left(x^{(k)}\right) - 2\alpha\left(A\nabla f\left(x^{(k)}\right)\right)^T A\nabla f\left(x^{(k)}\right) - b^T A\nabla f\left(x^{(k)}\right) - \left(A\nabla f\left(x^{(k)}\right)\right)^T b \\
= 0
\end{aligned}
$$

Using the equation $\nabla f\left(x^{(k)}\right) = A^T\left(Ax^{(k)} - b\right)$ we got, in the question2 (a).

$$
\begin{aligned}
\left(A\nabla f\left(x^{(k)}\right)\right)^T Ax^{(k)} + \left(Ax^{(k)}\right)^T A\nabla f\left(x^{(k)}\right) - 2\alpha\left(A\nabla f\left(x^{(k)}\right)\right)^T A\nabla f\left(x^{(k)}\right) - b^T A\nabla f\left(x^{(k)}\right) - \left(A\nabla f\left(x^{(k)}\right)\right)^T b \\
= 0
\end{aligned}
$$

$$
\left(A\nabla f\left(x^{(k)}\right)\right)^T Ax^{(k)} + \left(Ax^{(k)}\right)^T A\nabla f\left(x^{(k)}\right) - b^T A\nabla f\left(x^{(k)}\right) - \left(A\nabla f\left(x^{(k)}\right)\right)^T b = 2\alpha\left(A\nabla f\left(x^{(k)}\right)\right)^T A\nabla f\left(x^{(k)}\right)
$$

$$
2\alpha\left(A\nabla f\left(x^{(k)}\right)\right)^T A\nabla f\left(x^{(k)}\right) = \left(A\nabla f\left(x^{(k)}\right)\right)^T Ax^{(k)} + \left(Ax^{(k)}\right)^T A\nabla f\left(x^{(k)}\right) - b^T A\nabla f\left(x^{(k)}\right) - \left(A\nabla f\left(x^{(k)}\right)\right)^T b
$$

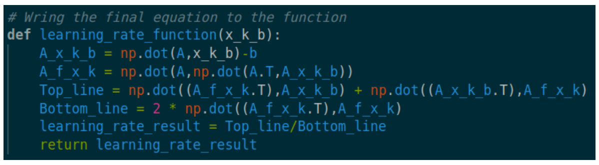
$$
\alpha = \frac{\left(A\nabla f\left(x^{(k)}\right)\right)^T Ax^{(k)} + \left(Ax^{(k)}\right)^T A\nabla f\left(x^{(k)}\right) - b^T A\nabla f\left(x^{(k)}\right) - \left(A\nabla f\left(x^{(k)}\right)\right)^T b}{2\left(A\nabla f\left(x^{(k)}\right)\right)^T A\nabla f\left(x^{(k)}\right)}
$$

$$
\begin{aligned}
\alpha &= \frac{\left(A\nabla f\left(x^{(k)}\right)\right)^T Ax^{(k)} + \left(Ax^{(k)}\right)^T A\nabla f\left(x^{(k)}\right) - b^T A\nabla f\left(x^{(k)}\right) - \left(A\nabla f\left(x^{(k)}\right)\right)^T b}{2\left(A\nabla f\left(x^{(k)}\right)\right)^T A\nabla f\left(x^{(k)}\right)} \\
&= \frac{\left(A\nabla f\left(x^{(k)}\right)\right)^T Ax^{(k)} + \left(\left(Ax^{(k)}\right)^T - b^T\right) A\nabla f\left(x^{(k)}\right) - \left(A\nabla f\left(x^{(k)}\right)\right)^T b}{2\left(A\nabla f\left(x^{(k)}\right)\right)^T A\nabla f\left(x^{(k)}\right)} \\
&= \frac{\left(A\nabla f\left(x^{(k)}\right)\right)^T \left(Ax^{(k)} - b\right) + \left(\left(Ax^{(k)}\right)^T - b^T\right) A\nabla f\left(x^{(k)}\right)}{2\left(A\nabla f\left(x^{(k)}\right)\right)^T A\nabla f\left(x^{(k)}\right)}
\end{aligned}
$$

Finally, we can get the equation which contains α as showing below.

$$
\alpha = \frac{\left(A\nabla f\left(x^{(k)}\right)\right)^T \left(Ax^{(k)} - b\right) + \left(\left(Ax^{(k)} - b\right)\right)^T A\nabla f\left(x^{(k)}\right)}{2\left(A\nabla f\left(x^{(k)}\right)\right)^T A\nabla f\left(x^{(k)}\right)}
$$

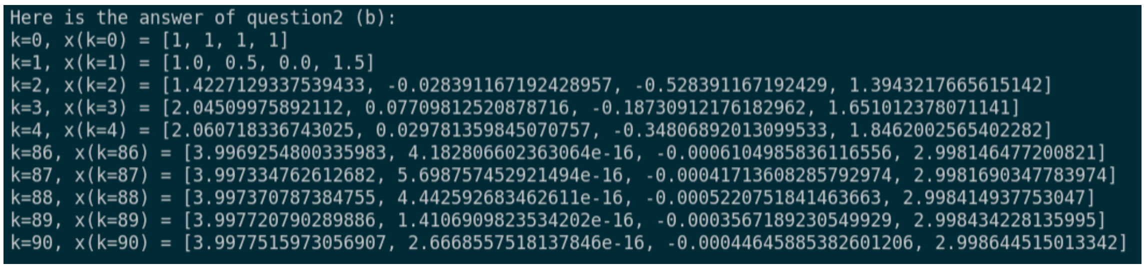
Then, setting up this equation in the python code. The following picture is the implementation of the equation.



```python
# Wring the final equation to the function
def learning_rate_function(x_k_b):
    A_x_k_b = np.dot(A,x_k_b)-b
    A_f_x_k = np.dot(A,np.dot(A.T,A_x_k_b))
    Top_line = np.dot((A_f_x_k.T),A_x_k_b) + np.dot((A_x_k_b.T),A_f_x_k)
    Bottom_line = 2 * np.dot((A_f_x_k.T),A_f_x_k)
    learning_rate_result = Top_line/Bottom_line
    return learning_rate_result
```

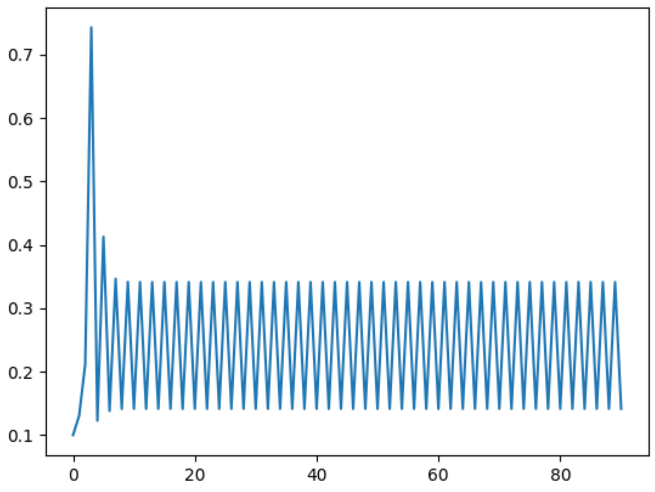
Other parts of the code are similar to question2 (a), you can also find it in the python files.

In this picture, we can find the first and the last 5 values of $x^{(k)}$ in the following picture.



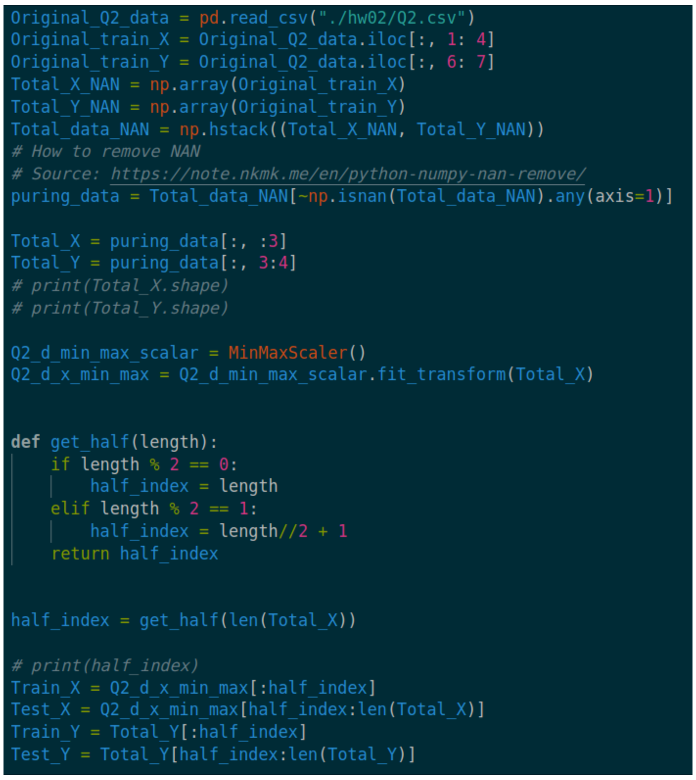
```
Here is the answer of question2 (b):
k=0, x(k=0) = [1, 1, 1, 1]
k=1, x(k=1) = [1.0, 0.5, 0.0, 1.5]
k=2, x(k=2) = [1.4227129337539433, -0.028391167192428957, -0.528391167192429, 1.3943217665615142]
k=3, x(k=3) = [2.04509975892112, 0.07709812520878716, -0.18730912176182962, 1.651012378071141]
k=4, x(k=4) = [2.060718336743025, 0.029781359845070757, -0.34806892013099533, 1.8462002565402282]
k=86, x(k=86) = [3.9969254800335983, 4.182806602363064e-16, -0.0006104985836116556, 2.998146477200821]
k=87, x(k=87) = [3.997334762612682, 5.698757452921494e-16, -0.00041713608285792974, 2.9981690347783974]
k=88, x(k=88) = [3.997370787384755, 4.442592683462611e-16, -0.0005220751841463663, 2.998414937753047]
k=89, x(k=89) = [3.997720790289886, 1.4106909823534202e-16, -0.0003567189230549929, 2.998434228135995]
k=90, x(k=90) = [3.9977515973056907, 2.6668557518137846e-16, -0.00044645885382601206, 2.998644515013342]
```

By running the code, the program can generate a list to store each alpha value for each step. Using the list, we can finally get the line graph of question2 (b).

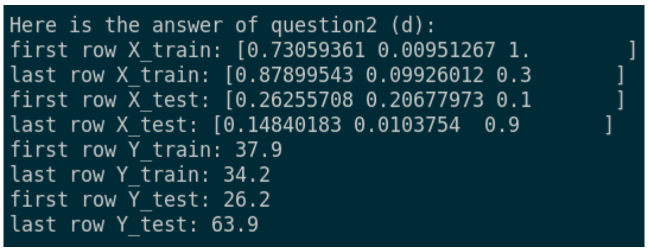


(c) Compare the result between question2 (a) and question2 (b), we can find the normal gradient descent algorithm using 222 steps to stop. Furthermore, the steepest gradient descent algorithm using 90 steps to stop. Therefore, we can find that the steepest descent algorithm is faster than the gradient descent. From the result given by question2 (b), we can find that the learning rate change sharply, which start at 0.1 then goes to more than 0.7. Then after several steps, the learning rate goes stable. Finally, the steepest descent algorithm can find the most suitable learning rate for the current model to make it converge in a fast way.

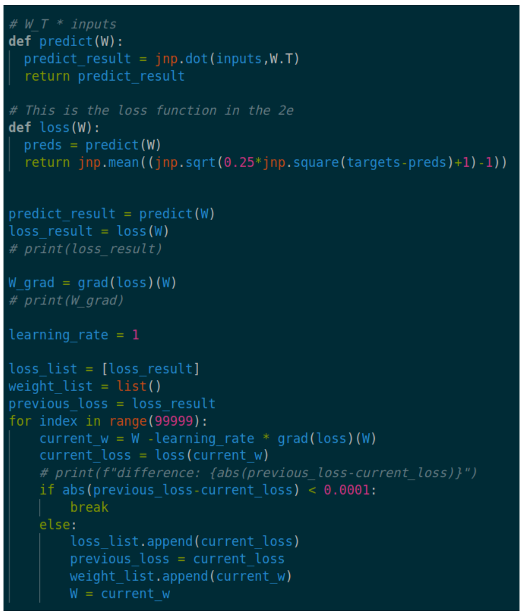
(d) Here is the data processing code in question2 (d).


```python
Original_Q2_data = pd.read_csv("./hw02/Q2.csv")
Original_train_X = Original_Q2_data.iloc[:, 1: 4]
Original_train_Y = Original_Q2_data.iloc[:, 6: 7]
Total_X_NAN = np.array(Original_train_X)
Total_Y_NAN = np.array(Original_train_Y)
Total_data_NAN = np.hstack((Total_X_NAN, Total_Y_NAN))
# How to remove NAN
# Source: https://note.nkmk.me/en/python-numpy-nan-remove/
puring_data = Total_data_NAN[~np.isnan(Total_data_NAN).any(axis=1)]

Total_X = puring_data[:, :3]
Total_Y = puring_data[:, 3:4]
# print(Total_X.shape)
# print(Total_Y.shape)

Q2_d_min_max_scalar = MinMaxScaler()
Q2_d_x_min_max = Q2_d_min_max_scalar.fit_transform(Total_X)


def get_half(length):
    if length % 2 == 0:
        half_index = length
    elif length % 2 == 1:
        half_index = length//2 + 1
    return half_index


half_index = get_half(len(Total_X))

# print(half_index)
Train_X = Q2_d_x_min_max[:half_index]
Test_X = Q2_d_x_min_max[half_index:len(Total_X)]
Train_Y = Total_Y[:half_index]
Test_Y = Total_Y[half_index:len(Total_Y)]
```

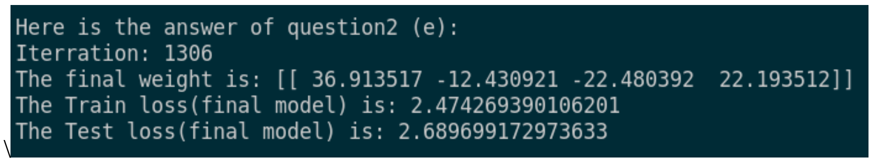
By running this part of the code, we can get the first row and last row of training data and testing data. You can find the result in the following picture showing below.


```
Here is the answer of question2 (d):
first row X_train: [0.73059361 0.00951267 1.        ]
last row X_train: [0.87899543 0.09926012 0.3       ]
first row X_test: [0.26255708 0.20677973 0.1       ]
last row X_test: [0.14840183 0.0103754  0.9       ]
first row Y_train: 37.9
last row Y_train: 34.2
first row Y_test: 26.2
last row Y_test: 63.9
```

(e) The code capture of question2 (e) is showing below.


```python
# W_T * inputs
def predict(W):
  predict_result = jnp.dot(inputs,W.T)
  return predict_result

# This is the loss function in the 2e
def loss(W):
  preds = predict(W)
  return jnp.mean((jnp.sqrt(0.25*jnp.square(targets-preds)+1)-1))


predict_result = predict(W)
loss_result = loss(W)
# print(loss_result)

W_grad = grad(loss)(W)
# print(W_grad)

learning_rate = 1

loss_list = [loss_result]
weight_list = list()
previous_loss = loss_result
for index in range(99999):
    current_w = W -learning_rate * grad(loss)(W)
    current_loss = loss(current_w)
    # print(f"difference: {abs(previous_loss-current_loss)}")
    if abs(previous_loss-current_loss) < 0.0001:
        break
    else:
        loss_list.append(current_loss)
        previous_loss = current_loss
        weight_list.append(current_w)
        W = current_w
```

By running the code given above, we can generate the result of the iterations and loss values.

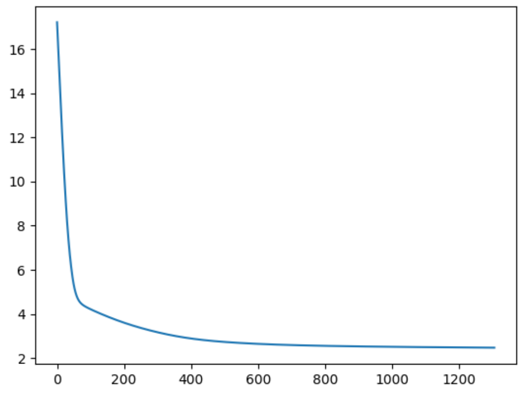


```
Here is the answer of question2 (e):
Iterration: 1306
The final weight is: [[ 36.913517 -12.430921 -22.480392  22.193512]]
The Train loss(final model) is: 2.474269390106201
The Test loss(final model) is: 2.689699172973633
```

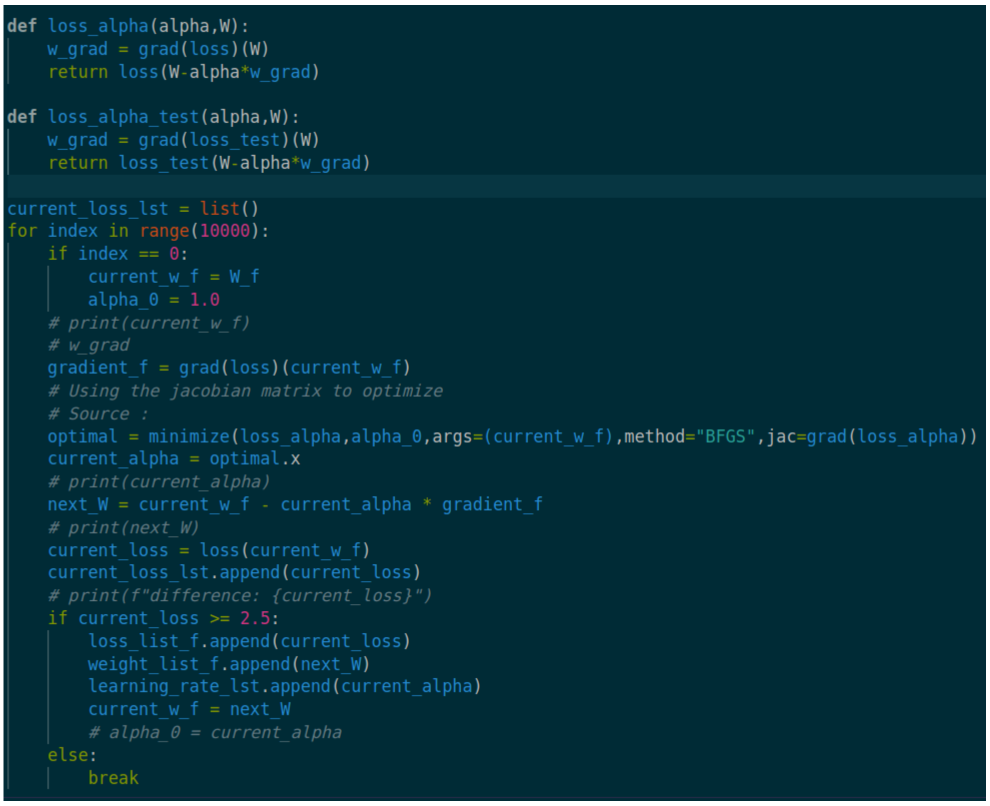
\

From this picture, we can find that the total number of iterations is 1306. The final weight is [36.913517, -12.430921, -22.480392, 22.193512]. Also, the training loss of the final model is 2.474269390106201 and the testing loss of the final model is 2.689699172973633.

Based on the loss data generated by the code, we can finally draw the plot of the loss value. The plot of the loss value for question2 (e) is showing below.



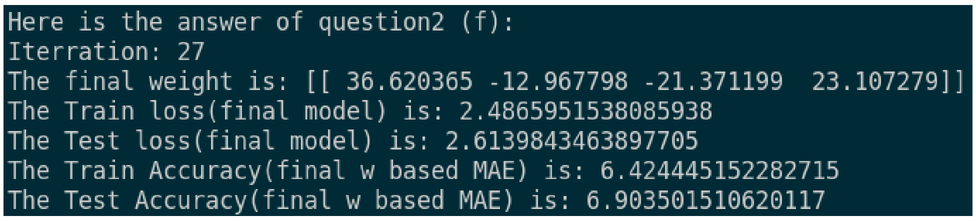
(f) The question2 (f), I used the minimize function provided by SciPy to update the learning rate, then implementing the steepest gradient descent algorithm. Here is the capture of code implementation of this part.



```python
def loss_alpha(alpha,W):
    w_grad = grad(loss)(W)
    return loss(W-alpha*w_grad)

def loss_alpha_test(alpha,W):
    w_grad = grad(loss_test)(W)
    return loss_test(W-alpha*w_grad)

current_loss_lst = list()
for index in range(10000):
    if index == 0:
        current_w_f = W_f
        alpha_0 = 1.0
    # print(current_w_f)
    # w_grad
    gradient_f = grad(loss)(current_w_f)
    # Using the jacobian matrix to optimize
    # Source :
    optimal = minimize(loss_alpha,alpha_0,args=(current_w_f),method="BFGS",jac=grad(loss_alpha))
    current_alpha = optimal.x
    # print(current_alpha)
    next_W = current_w_f - current_alpha * gradient_f
    # print(next_W)
    current_loss = loss(current_w_f)
    current_loss_lst.append(current_loss)
    # print(f"difference: {current_loss}")
    if current_loss >= 2.5:
        loss_list_f.append(current_loss)
        weight_list_f.append(next_W)
        learning_rate_lst.append(current_alpha)
        current_w_f = next_W
        # alpha_0 = current_alpha
    else:
        break
```

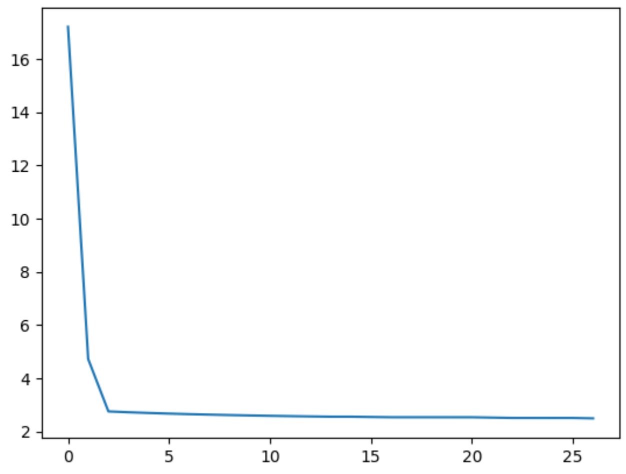
In this part of code, using the parameter provided by the Jacobian matrix to do the minimization, can speed up the iterations.

Here is the result of running the code. From this capture, we can find that the iteration is 27, as well as the final weight is [[36.620365 -12.967798 -21.371199 23.107279]]. From the picture, we can also get the training loss is 2.4865951538085938 and the testing loss is 2.6139843463897705. Finally, In this part, I decided to use the MAE to compute the training accuracy and testing accuracy. The Training accuracy is 6.424445152282715. The testing accuracy is 6.903501510620117. You can also find the same result from the picture given below.



```
Here is the answer of question2 (f):
Iterration: 27
The final weight is: [[ 36.620365 -12.967798 -21.371199  23.107279]]
The Train loss(final model) is: 2.4865951538085938
The Test loss(final model) is: 2.6139843463897705
The Train Accuracy(final w based MAE) is: 6.424445152282715
The Test Accuracy(final w based MAE) is: 6.903501510620117
```

Finally, we can use the loss list to generate the plot of question2 (f). From the picture, we can find that the iteration is less than question2 (e). Also, the structure of this line graph is steep. One important thing that should be mentioned here is that when we do the training without using the Jacobian matrix, we can find the learning rate is still 1. To get a good iteration result and solving this problem without using the Jacobian matrix, we can change the parameters in the JAX function. Here is the reason. Although the speed of JAX is faster than NumPy, the accuracy is not better than NumPy. We can change the type of the parameter to NumPy array to make the result more accurate.



(g) This part is going to introduce a new gradient-based algorithm which is called Adagrad. Compared with the SGD, the Adagrad has higher robustness [1]. Also, this algorithm has been used in large-scale neural networks training. For the Adagrad algorithm, it adds constraints to the learning rate [5].

Using the step to check the difference between SGD and Adagrad. Here are the steps of the traditional SGD method [4].

$$Step\ 1\ \ Calculate\ the\ Gradient: \hat{g} \leftarrow +\frac{1}{m}\nabla_0 \sum_i L\left(f\left(x^{(i)};\theta\right), y^{(i)}\right)$$

$$Step\ 2\ \ Updating\ parameter: \theta \leftarrow \theta - \epsilon\hat{y}$$

Here are the steps of Adagrad, which is different from SGD [2][4]:

$$Step\ 1\ \ Calculate\ the\ Gradient: g \leftarrow \frac{1}{m}\nabla_0 \sum_i L\left(f\left(x^{(i)};\theta\right), y^{(i)}\right)$$

$$Step\ 2\ \ accumulate\ the\ squared\ gradient: r \leftarrow r + g \odot g$$

$$Step\ 3\ \ Calculate\ the\ update: \Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{r}} \odot g$$

$$Step\ 4\ \ Apply\ the\ update: \theta \leftarrow \theta + \Delta\theta$$

Here in these equations, ϵ is the global learning rate, which needs to be set manually. δ is a constant, this value is to avoid the situation of value 0. The value r is used to accumulate the gradient, the default value of r is 0.

With the step increase, the value r goes larger. This will cause the learning rate to go smaller. Therefore, in the previous training Adagrad encourage convergence. However, with the time and training step grows, the learning rate to go smaller. This means the Adagrad finally will punish the converge. This is the reason why the training step goes slower and slower. Also, due to the accumulated value r, the Adagrad is good at dealing with the sparse gradient, which will give a better performance than SGD [3].

However, this method still has disadvantages. This method still needs us to set a global learning rate. If the global learning rate is too large, the adjustment of the gradient will be influenced. Furthermore, if the value r which adds the gradient will go very large, this will cause the training process to stop prematurely [5].

Although the Adagrad increases the robustness and efficiency, it still has problems that need to be optimized.

References:

[1] Duchi, J., Hazan, E. and Singer, Y., 2011. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, *12*(7).

Link: https://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf

[2] An Introduction to AdaGrad

Link: https://medium.com/konvergen/an-introduction-to-adagrad-f130ae871827

[3] An overview of gradient descent optimization algorithms

Link: https://ruder.io/optimizing-gradient-descent/index.html#adagrad

[4] Deep learning optimization method – Adagrad (Chinese Resource)

Link: https://zhuanlan.zhihu.com/p/38298197

[5] The comparison between SGD, Adagrad, Adadelta, Adam, Adamax, Nadam (Chinese Resource)

Link: https://zhuanlan.zhihu.com/p/22252270