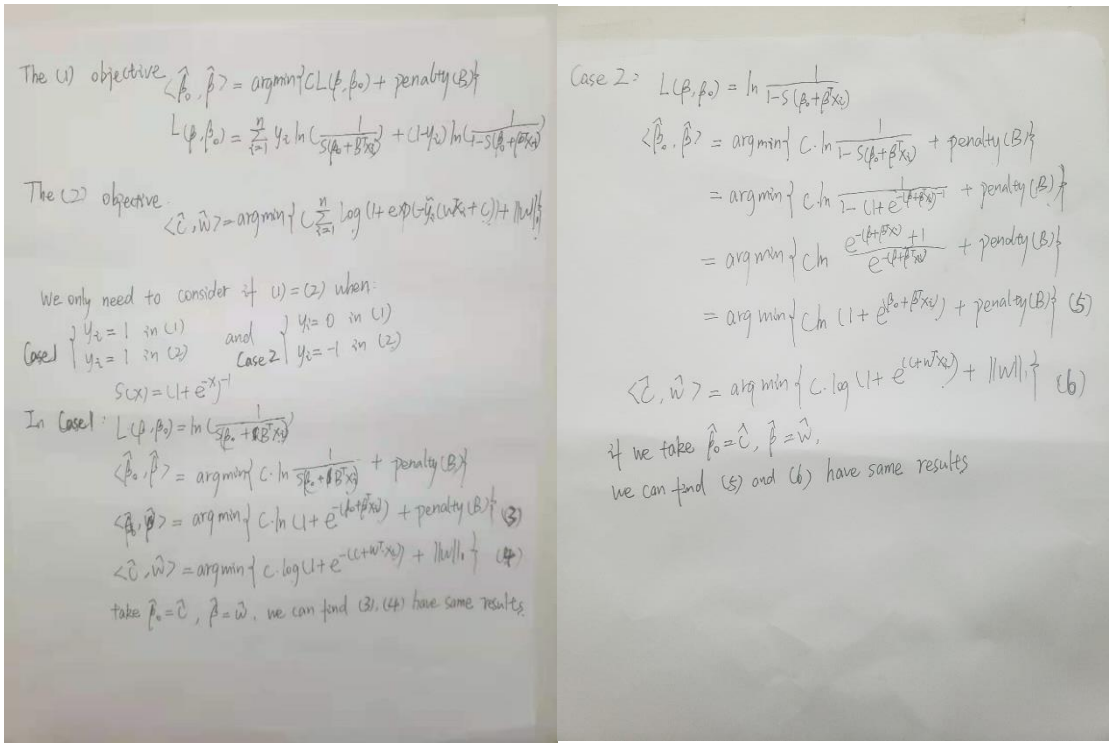
COMP9417 Homework2

Zid: z5239803

Name: Zhengyang Jia

Q1:

(a)



(b)

```
[28] file_array= np.genfromtxt('/content/drive/MyDrive/9417 HW2/Q1.csv',delimiter=",")
```

```
[31] file_arr = file_array[1:]
     file_train, file_test = file_arr[:500], file_arr[500:]
     print(len(file_train))
     print(len(file_test))
```

```
     500
     150
```

```
[32] X_train, Y_train = file_train[:,:45], file_train[:,45:]
     X_test, Y_test = file_test[:,:45], file_test[:,45:]
     C_Grid = np.linspace(0.0001, 0.6, 100)
     print(len(C_Grid))
```

```
     100
```

```
[33] L=[]
     for item in C_Grid:
         Clf = sklearn.linear_model.LogisticRegression(C=item, solver='liblinear',penalty='l1')
         T=[]
         for t in range(10):
             a, b = 50*t, 50*t+50
             current_X_test = X_train[a:b,:]
             current_Y_test = Y_train[a:b,:]
             current_X_train = np.vstack((X_train[0:a,:], X_train[b:500,:]))
             current_Y_train = np.vstack((Y_train[0:a,:], Y_train[b:500,:]))

             Clf.fit(current_X_train, current_Y_train.ravel())
             pred_Y = Clf.predict_proba(current_X_test)
             loss = sklearn.metrics.log_loss(current_Y_test, pred_Y)
             T.append(loss)
         L.append(T)
```

```
[34] Loss_mean=[]
     for i in range(100):
         mean = sum(L[i])/10
         Loss_mean.append(mean)
```

```
[35] print(Loss_mean)
     print(min(Loss_mean))
     print(Loss_mean.index(min(Loss_mean)))
```

```
[0.6931471805599453, 0.6931471805599453, 0.6326478156782008, 0.5945706656978669, 0.5805321434523152, 0.5737897871482163, 0.5700325041508955, 0.5677360834754358,
0.5397255142709293
31
```

```
[36] opti_C= C_Grid[31]
     opti_C
```

```
0.18794747474747472
```

We found that when the index of C_grid is 31 (32th element), the mean of log_loss is minimum, which is around 0.539725. The corresponding C value is 0.18794747474747472.

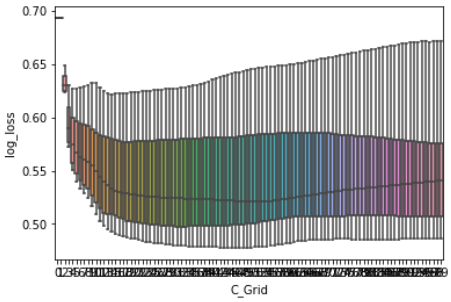
```
[37] Clf = sklearn.linear_model.LogisticRegression(C=opti_C, solver='liblinear',penalty='l1')
     Clf.fit(X_train, Y_train)

     pred_train_Y = Clf.predict_proba(X_train)
     pred_test_Y = Clf.predict_proba(X_test)
     Accuracy_Train = Clf.score(X_train, Y_train)
     Accuracy_Test = Clf.score(X_test, Y_test)
     print(Accuracy_Train)
     print(Accuracy_Test)
```

```
0.752
0.74
/usr/local/lib/python3.7/dist-packages/sklearn/utils/validation.py:760: DataConversionWarning: A column-vector y was passed when a 1d array was expected
  y = column_or_1d(y, warn=True)
```

The Train_Accuracy is 0.752, the Test_Accuracy is 0.74

The boxplot looks like this:

```
[38] import seaborn as sns
     import matplotlib.pyplot as plt
     plt.xlabel('C_Grid')
     plt.ylabel('log_loss')
     plt.xticks(np.linspace(0.0001, 0.6, 100))
     ax = sns.boxplot(data=L)
     plt.show()
```



(c)

Two reasons for different results:

Reason-1: In 1(b) we used log_loss function to evaluate, while here in 1(c), we use different scoring method, the default value of the GridSearchCV parameter scoring = None, which makes the result different to 1(b).

Reason-2: The data segmentation methods. In 1(b) we segmented the 500-row train_set into 10 parts sequentially, while the method GridSearchCV here have different rules, it segments train_set due to the label-y. It tries to make the data with label y=0 and y=1 have same proportion in each validation. Thus, it can't have same segmentation method with1(b), which leads to the difference.

▾ **Q1c**

```
[100] K=C_Grid.tolist()
      param_grid = {'C':K}
```

```
[101] grid_lr = GridSearchCV(estimator=LogisticRegression(penalty='l1',
                solver='liblinear'), cv=KFold(n_splits=10), param_grid =param_grid, scoring = 'neg_log_loss',)
      grid_lr.fit(X_train, Y_train.ravel())
```

```
      GridSearchCV(cv=KFold(n_splits=10, random_state=None, shuffle=False),
                   error_score=nan,
                   estimator=LogisticRegression(C=1.0, class_weight=None, dual=False,
                                                fit_intercept=True,
                                                intercept_scaling=1, l1_ratio=None,
                                                max_iter=100, multi_class='auto',
                                                n_jobs=None, penalty='l1',
                                                random_state=None, solver='liblinear',
                                                tol=0.0001, verbose=0,
                                                warm_start=False),
                   iid='deprecated',...
                                    0.09705353535353535, 0.10311313131313131,
                                    0.10917272727272727, 0.11523232323232323,
                                    0.12129191919191919, 0.12735151515151513,
                                    0.1334111111111111, 0.13947070707070705,
                                    0.145530303030303, 0.15158989898989897,
                                    0.15764949494949493, 0.1637090909090909,
                                    0.16976868686868685, 0.1758282828282828, ...]},
                   pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
                   scoring='neg_log_loss', verbose=0)
```

After changing some parameters:

cv= KFold(n_splits=10), param_grid= {"C": C_grid},

scoring="neg_log_loss. We can get the same C value with 1(b)

```
[102] grid_lr.best_params_

      {'C': 0.18794747474747472}
```

(d)

There are 10000 iteration, and in each iteration, we emerge 500 random row to train the model. Then, we get the vector coeffient, there are 10000 these vector in total, we need to calculate coef values of the position 5% and 95% of each sorted feature, along with the the mean of each coef.

Q1d

```
[112] model = sklearn.linear_model.LogisticRegression(C=1, solver='liblinear',penalty='l1')
      np.random.seed(12)
```

```
[113] Coe=[]
      for i in range(10000):
          train_x_cr, train_y_cr =[], []
          for j in range(500):
              id= np.random.randint(0,499)
              train_x_cr.append(X_train[id])
              train_y_cr.append(Y_train[id].ravel())
          train_y_cr = np.array(train_y_cr)
          model.fit(train_x_cr, train_y_cr.ravel())
          coefficient = model.coef_
          coeffs = coefficient.tolist()
          coeff = coeffs[0]
          Coe.append(coeff)
```

```
[114] print(len(Coe))
      Arr_Coe = np.array(Coe)
      len(Arr_Coe[:,0])

      10000
      10000
```

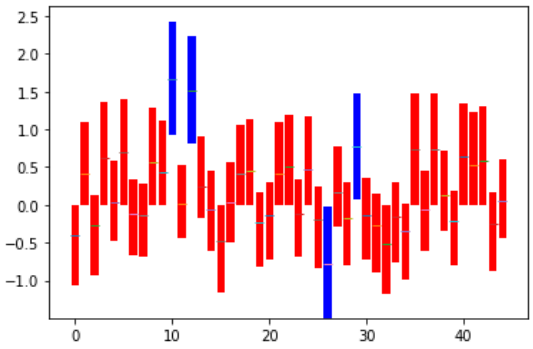
```
[115] Boundary_Up=[]
      Boundary_Down=[]
      for i in range(45):
          Single_Boundary_up =[]
          Single_Boundary_down =[]
          T= Arr_Coe[:,i]
          T.sort()
          percent_5 = T[500]
          percent_95 = T[9500]
          #Single_Boundary_down.append(percent_5)
          #Single_Boundary_up.append(percent_95)
          Boundary_Down.append(percent_5)
          Boundary_Up.append(percent_95)
```

```
[116] Mean=[]
      for i in range(45):
          summ = sum(Arr_Coe[:,i])
          mean = summ/10000
          Mean.append(mean)
```

```
[117] H =[]
      for i in range(45):
          h= Boundary_Up[i]-Boundary_Down[i]
          H.append(h)
```

From the below figure, we can find that the 10^{th}, 12^{th}, 26^{th}, 29^{th} features are painted blue, which means these four features are more significant to predict y than other 'painted red' features.

```
[118] data=  Boundary_Up
      size  =  45
      x  =  np.arange(size)
      for  i  in  range(45):
          if  Boundary_Down[i]*Boundary_Up[i]>0:
              plt.bar(i,  height=H[i],bottom=Boundary_Down[i],  label='b',  color="b")
          else:
              plt.bar(i,  height=H[i],bottom=Boundary_Down[i],  label='b',  color="r")
          plt.plot(i,Mean[i],"_")
      #plt.bar(x=[i*0.8  for  i  in  range(45)],height=data,  bottom=Boundary_Down)
      plt.show()
```



(e)

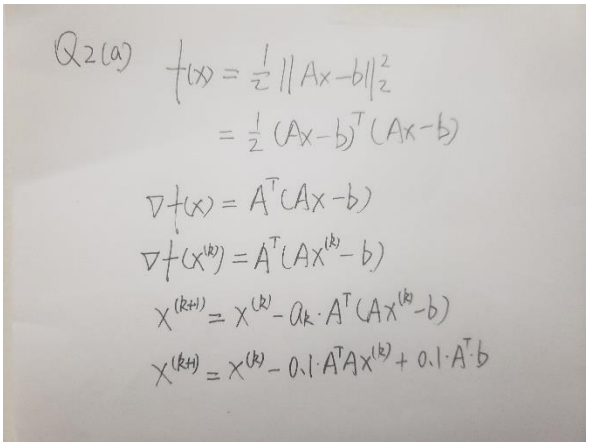
The red bar features have limited effect on prediction, they are less important. The blue bar features have considerate effect on prediction, so they are more important. If we have too many red bars, we can gradually decrease value C to decrease the punishment. If we have too many blue bars, we can gradually increase value C to increase punishment.

If regularization is necessary depends on whether we have too many red bars. If we have too many red bars(0 in the range), regularization is necessary, else regularization is not necessary.

Q2:

(a)

For the gradient step(3), the derivation I made is like this:



After getting the derivative of f(x), I used vect2 to represent the f(x) derivative. When $\| \nabla f(x^{(k)}) \|_2^2 < 0.001$, we end loop.

```
[31] A= np.array([[1,0,1,-1],[-1,1,0,2],[0,-1,-2,1]])
     B= np.array([[1],[2],[3]])
```

```
[32] def Derv_F(x):
         Vect1 = A.dot(x)-B
         A_transpose = A.T
         Vect2 = A_transpose.dot(Vect1)
         x_current = x-0.1*Vect2
         return Vect2, x_current
```

```
[33] X=np.array([[1],[1],[1],[1]])
     Norm2_derv=1
     X_array=[[1,1,1,1]]
     while Norm2_derv>=0.001:
         derv= Derv_F(X)[0]
         Norm2_derv = 0
         for i in range(4):
             Norm2_derv = Norm2_derv + derv[i]*derv[i]
         Norm2_derv = math.sqrt(Norm2_derv)
         X = Derv_F(X)[1]
         T = []
         for j in X:
             T.append(float(j))
         X_array.append(T)
```

We now know that X_array has 224 elements, the last X_array index is 223

```
[34] len(X_array)

     224
```

```
[35] for i in range(5):
         print("k = " ,end = "")
         print(i, end=",    x")
         print(f"({i}) = ", end="")
         print(X_array[i])
```

```
k = 0,  x(0) = [1, 1, 1, 1]
k = 1,  x(1) = [1.0, 0.5, 0.0, 1.5]
k = 2,  x(2) = [1.2, 0.25, -0.25, 1.45]
k = 3,  x(3) = [1.345, 0.125, -0.36, 1.44]
k = 4,  x(4) = [1.4565, 0.06250000000000003, -0.4075, 1.4589999999999999]
```

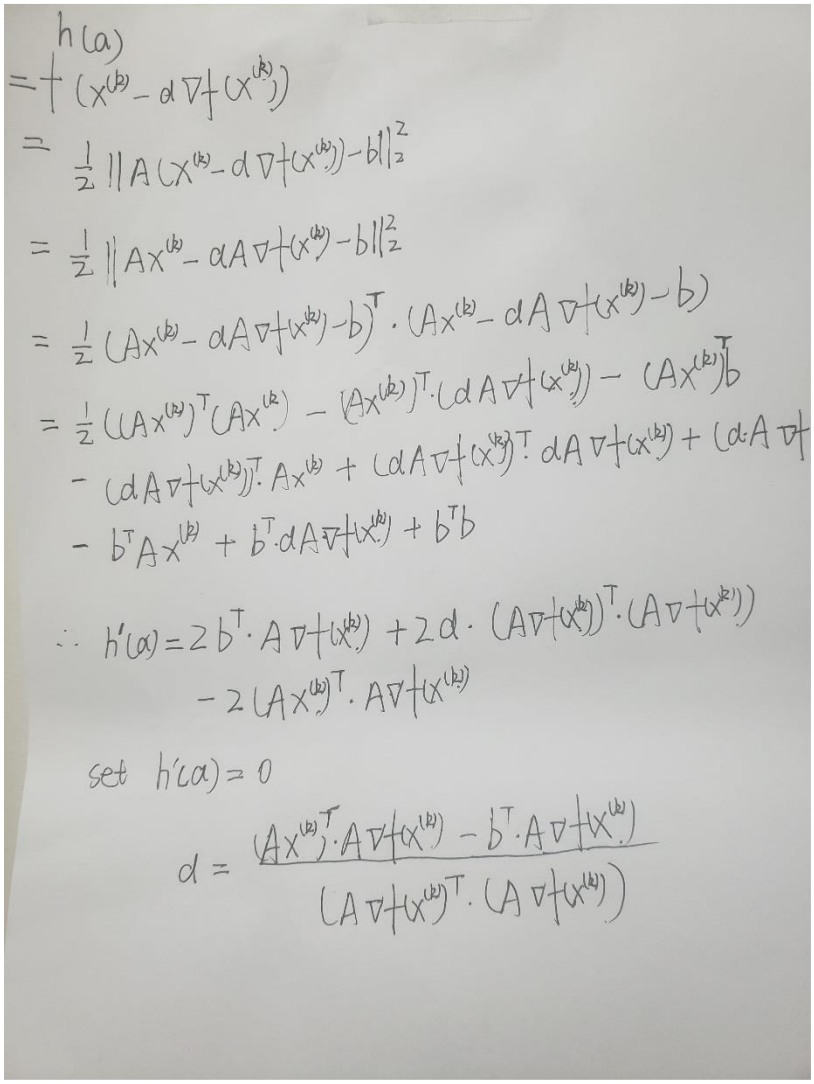
```
[36] for  i  in  range(219,224):
         print("k  =  "  ,end  =  "")
         print(i,  end=",     x")
         print(f"({i})  =  ",  end="")
         print(X_array[i])
```

```
k = 219,   x(219) = [3.9970914189366624, 6.003844059340653e-16, -0.0005441474174037207, 2.9981797137714685]
k = 220,   x(220) = [3.997181464022511, 6.003844059340653e-16, -0.0005273014709276356, 2.998236066964365]
k = 221,   x(221) = [3.997268721454411, 6.003844059340653e-16, -0.0005109770484054397, 2.998290675551221]
k = 222,   x(222) = [3.997353277533736, 5.5597548494905895e-16, -0.0004951580042775326, 2.9983435935422897]
k = 223,   x(223) = [3.9974352158901034, 5.5597548494905895e-16, -0.0004798286928255045 6, 2.9983948732757533]
```

(b)

the choice of α^k in 2(b) differs 2(a).

The deduction of α^k is shown here:

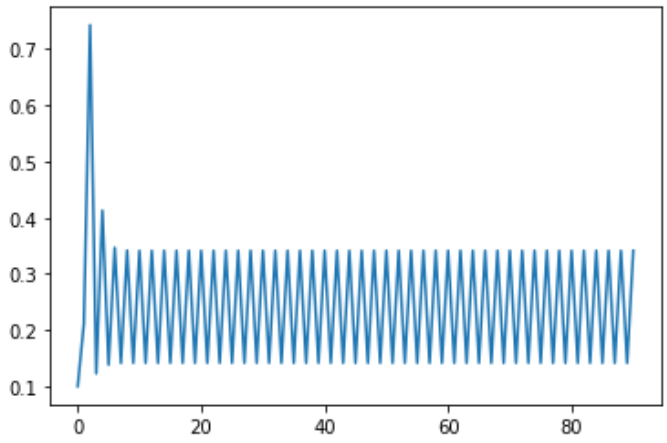


[55]
```
def Derv_Fb(x,a,i):
    Vect0 = (A.dot(x)).T
    Vect1 = A.dot(x)-B
    A_transpose = A.T
    Vect2 = A_transpose.dot(Vect1)
    A2 = A.dot(A_transpose)
    Vect3 = (B.T).dot(A2)
    Vect4 = A2.dot(Vect1)
    Hx = A.dot(Vect2)

    Nume1= (A.dot(x)).T
    Nume2= B.T
    Numerator = Nume1.dot(Hx) - Nume2.dot(Hx)
    Denominator = (Hx.T).dot(Hx)
    a = Numerator/Denominator
    if i ==0 :
        x = x- 0.1*Vect2
    else:
        x = x- a*Vect2
    return Vect2, x, a
```

[56]
```
X=np.array([[1],[1],[1],[1]])
Norm2_derv=1
A_arr =[]
X_array=[[1,1,1,1]]
n=0
insta_a = 0.1
A_arr.append(insta_a)
while Norm2_derv>=0.001:
    List = Derv_Fb(X,insta_a,n)
    derv= List[0]
    X = List[1]
    insta_a = List[2]
    T = []
    Norm2_derv = 0
    for i in range(4):
        Norm2_derv = Norm2_derv + derv[i]*derv[i]
    Norm2_derv = math.sqrt(Norm2_derv)

    if n>0:
        A_arr.append(float(insta_a))
    if n>0:
        for j in X:
            T.append(float(j))
        X_array.append(T)
    n=n+1
```

```
[57] print(len(X_array))
     plt.plot(A_arr)
     plt.show
```

```
91
<function matplotlib.pyplot.show>
```




```
[58] for i in range(5):
         print("k = " ,end = "")
         print(i, end=",    a")
         print(f"({i}) = ", end="")
         print('{:f}'.format(A_arr[i]), end="")

         print(i, end=",    x")
         print(f"({i}) = ", end="")
         print(X_array[i])
```

```
k = 0,   a(0) = 0.1000000,   x(0) = [1, 1, 1, 1]
k = 1,   a(1) = 0.2113561,   x(1) = [1.4227129337539433, -0.028391167192428957, -0.528391167192429, 1.3943217665615142]
k = 2,   a(2) = 0.7431132,   x(2) = [2.04509975892112, 0.07709812520878717, -0.18730912176182957, 1.651012378071141]
k = 3,   a(3) = 0.1227443,   x(3) = [2.060718336743025, 0.02978135984507077, -0.34806892013099533, 1.8462002565402282]
k = 4,   a(4) = 0.4127864,   x(4) = [2.3888890702766545, -0.031685268933381604, -0.28257452679180955, 1.8589823170601292]
```

```
[59] for i in range(87,91):
         print("k = " ,end = "")
         print(i, end=",    a")
         print(f"({i}) = ", end="")
         print('{:f}'.format(A_arr[i]), end="")

         print(i, end=",    x")
         print(f"({i}) = ", end="")
         print(X_array[i])
```

```
k = 87,   a(87) = 0.14143287,   x(87) = [3.997370787384753, 1.817230908463318e-16, -0.0005220751841464417, 2.998414937753046]
k = 88,   a(88) = 0.34136288,   x(88) = [3.997720790289887, 1.817230908463318e-16, -0.0003567189230538991 7, 2.998434228135995]
k = 89,   a(89) = 0.14143289,   x(89) = [3.997751597305691, 1.817230908463318e-16, -0.0004464588538253864, 2.998644515013342]
k = 90,   a(90) = 0.34136290,   x(90) = [3.998050906486835, 1.817230908463318e-16, -0.0003050524643968114 7, 2.998661011415628 7]
```

(c)

Gradient Descent: focuses descent in the negative gradient direction.

Steepest Gradient Descent: The directional of steepest descent (or ascent) is the direction amongst all nearby directions that lowers or raises the value of f the most.

The direction of gradient descent method is negative gradient, while the direction of steepest descent method is the direction:

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

In 2(b), we used less steps to converge the loss, which indicates steepest gradient in such case is better than gradient descent.

Steepest gradient can converge faster than gradient descent. Besides steepest gradient is easier to find appropriate stepsize.

Why it is a reasonable condition to terminate:

When the derivative of f(x) is very close to 0, we assume we got the appropriate alpha, we also don't want over fitting. If the threshold is too small, we will have overfitting problems.

so setting $\| \nabla f(x^{(k)}) \|_2^2 < 0.001$ is reasonable.

(d)

Q2d

```
[99] f_array= np.genfromtxt('/content/drive/MyDrive/9417 HW2/Q2.csv',delimiter=",")
     f_array=f_array[1:,:]
```

```
[100] # Delete rows with Nan
      f_new = []
      for row in f_array:
          count = 0
          for item in row:
              if np.isnan(item):
                  count = count+1
          if count==0:
              f_new.append(row)
```

```
[101] len(f_new)
      f_new= np.array(f_new)
      Set_X = f_new[:,1:4]
      Set_Y = f_new[:,6]

      from sklearn.preprocessing import MinMaxScaler
      sc = MinMaxScaler(feature_range=(0,1))
      Set_X = sc.fit_transform(Set_X)
      X_train, X_test = Set_X[:204], Set_X[204:]
      Y_train, Y_test = Set_Y[:204], Set_Y[204:]
```

(e)

```
[36] import jax.numpy as jnp
     from jax import grad, jit, vmap
     from jax import random
```

```
[37] def predict(W, inputs):
           return jnp.dot(inputs, W)
```

```
[38] def Loss_new(W):
         pred = predict(W, X_new)
         inner_sqrt = 1/4 *jnp.square(Y_train-pred)+1.0
         mean_Los = jnp.mean(jnp.sqrt(inner_sqrt)-1.0)
         return mean_Los
```

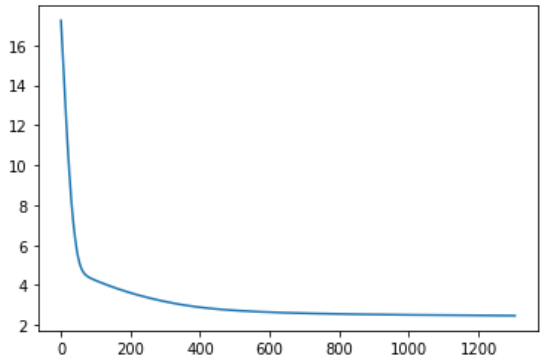
```
[39]   X_new=[]
       for i in range(len(X_train)):
             x = X_train[i]
             Lx = list(x)
             Lx.insert(0,1)
             x = jnp.array(Lx)
             X_new.append(x)
       X_new = jnp.array(X_new)
```

```
[40] diff_loss = 1
     List_W = []
     List_loss = []
     W = jnp.array([1.0,1.0,1.0,1.0])
     List_W.append(W)
     List_loss.append(Loss_new(W))
     k = 0
     while diff_loss>=0.0001:
         loss1 = Loss_new(W)
         W_grad = grad(Loss_new)(W)
         W = W - W_grad
         List_W.append(W)
         loss2 = Loss_new(W)
         List_loss.append(loss2)
         diff_loss = abs(loss2-loss1)
         k += 1
```

```
[41]  # Use the final W for test validation
      X_test_new=[]
      for i in range(len(X_test)):
            x = X_test[i]
            Lx = list(x)
            Lx.insert(0,1)
            x = jnp.array(Lx)
            X_test_new.append(x)
      X_test_new = jnp.array(X_test_new)
      pred = predict(W, X_test_new)
      Last_Test_Loss = jnp.mean(jnp.sqrt(1/4 *jnp.square(Y_test-pred)+1.0)-1.0)
      last_test_loss= float(Last_Test_Loss)
```

```
[42] print("It took",len(List_loss),"iterations to converge the loss.")
     plt.plot(List_loss)
     plt.show()
```

```
It took 1306 iterations to converge the loss.
```




```
[43] print("Final train loss:", List_loss[-1])
     print("Final test loss: ", last_test_loss)
     print("Final weight vector:",List_W[-1])
```

```
Final train loss: 2.4738414
Final test loss:  2.695838689804077
Final weight vector: [ 37.048077 -12.680338 -22.374311  22.20594 ]
```

(g)

Adagrad is an optimizer with parameter-specific learning rates, which are adapted relative to how frequently a parameter gets updated during training. The more updates a parameter receives, the smaller the updates.

```
tf.keras.optimizers.Adagrad(
        learning_rate=0.001,initial_accumulator_value=0.1,
        epsilon=1e-07,name="Adagrad",**kwargs
)
```

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\varepsilon I + diag(G_t)}} \cdot g_t, \tag{1}$$

θ is the parameter to be updated, η is the initial learning rate, $\mathcal{E}$ is some small quantity that used to avoid the division of zero, I is the identity matrix, g_τ is the gradient estimate in time-step t that we can get with the equation

$$g_t = \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta \mathcal{L}(x^{(i)}, y^{(i)}, \theta_t), \tag{2}$$

$$G_t = \sum_{\tau=1}^{t} g_\tau g_\tau^\top. \tag{3}$$

Feature:

While the g_t is small, the regularizer is large, so it could amplify the gradient. While the g_t is larger, the regularizer is smaller so it could constrain the gradient.

This Adagrad method suits for dealing with sparse gradients

Drawbacks:

In the later period of training, the speed of decreasing gradient gets faster, which could leads to an early end of training.

learning rate η can not be set large. If it's large, regularizer is very sensitive and affects the precise regulation of gradient.