Name: Zicheng Qu zID: z5092597

Question 1.

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

$L(\beta,\beta_0) = L(\beta,\beta_0 \mid y_i=0) + L(\beta,\beta_0 \mid y_i=1)$

$= \sum_{i=1}^{a} \ln\left(\frac{1}{1-s(\beta_0+\beta^T x_i)}\right) + \sum_{i=1}^{b} \ln\left(\frac{1}{s(\beta_0+\beta^T x_i)}\right)$, where a is the number of $y_i=0$, b is the one of $y_i=1$, $a+b=n$

$s(z) = (1+e^{-z})^{-1}$

Hence $s(\beta_0+\beta^T x_i) = \left(1+\exp(-1\cdot(\beta_0+\beta^T x_i))\right)^{-1}$

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

$\frac{1}{1-s(\beta_0+\beta^T x_i)} = \left(1-s(\beta_0+\beta^T x_i)\right)^{-1} =$

$= \left(1-\frac{1}{1+\exp(-1\cdot(\beta^T x_i+\beta_0))}\right)^{-1}$

$= \left(\frac{\exp(-1\cdot(\beta^T x_i+\beta_0))}{1+\exp(-1\cdot(\beta^T x_i+\beta_0))}\right)^{-1}$

$= \frac{1+\exp(-1\cdot(\beta^T x_i+\beta_0))}{\exp(-1\cdot(\beta^T x_i+\beta_0))}$

$= 1+\left(\exp(-1\cdot(\beta^T x_i+\beta_0))\right)^{-1}$

$= 1+\exp(1\cdot(\beta^T x_i+\beta_0))$

$\ln\left(\frac{1}{1-s(\beta_0+\beta^T x_i)}\right) = \ln\left(1+\exp(1\cdot(\beta^T x_i+\beta_0))\right)$

Hence, $L(\beta,\beta_0) = L(\beta,\beta_0 \mid y_i=0) + L(\beta,\beta_0 \mid y_i=1)$

$= \sum_{i=1}^{a} \ln\left(\frac{1}{1-s(\beta_0+\beta^T x_i)}\right) + \sum_{i=1}^{b} \ln\left(\frac{1}{s(\beta_0+\beta^T x_i)}\right)$

$= \sum_{i=1}^{a} \ln\left(1+\exp(1\cdot(\beta^T x_i+\beta_0))\right) + \sum_{i=1}^{b} \ln\left(1+\exp(-1(\beta^T x_i+\beta_0))\right)$

From Question 1 (a), I know that $\hat{\beta}_0 = \hat{c}$, $\hat{\beta} = \hat{w}$, also $\text{penalty}(\beta) = \|\beta\|_1$

Hence $\text{penalty}(\beta) = \|w\|_1$, $\beta^T x_i + \beta_0 = w^T x_i + c$

When $y_i = 0$ and $\tilde{y}_i = -1$:

$\log\left(1+\exp(-\tilde{y}_i(w^T x+c))\right) = L(\beta,\beta_0 \mid y_i=0) = \ln\left(1+\exp(1\cdot(\beta^T x_i+\beta_0))\right)$

When $y_i = 1$ and $\tilde{y}_i = 1$:

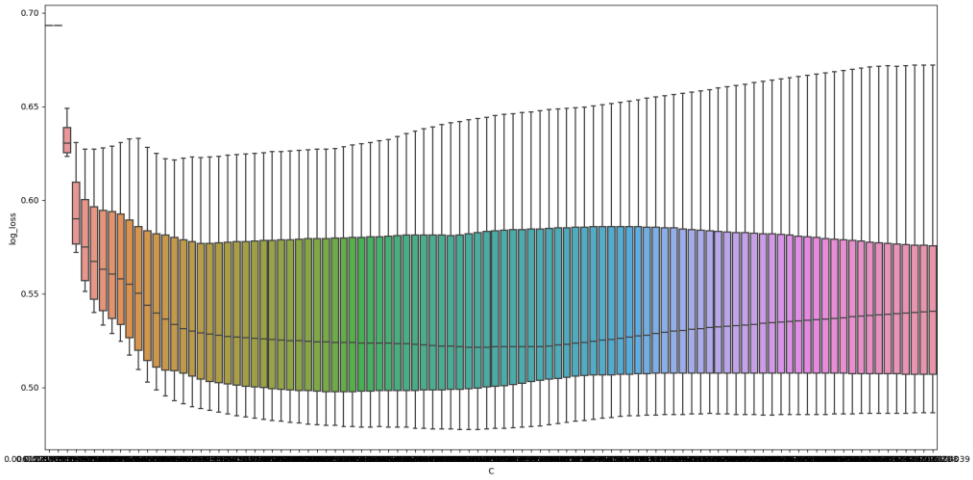
$$\log\left(1+\exp(-\tilde{y}_i(w^T x_i + c))\right) = L(\beta, \beta_0 \mid y_i = 0) = \ln\left(1 + \exp(-1 \cdot (\beta^T x_i + \beta_0))\right)$$

Hence $$CL(\beta_0, \beta) = C \sum_{i=1}^{n} \log\left(1+\exp(-\tilde{y}_i(w^T x_i + c))\right)$$

Hence, the two objectives (1) and (2) are identical.

As for the role of C, I think it is very similar to the standard LASSO parameter λ in some way to balance the accuracy and generalization of the model. However, the larger the λ is, the more generalization ability the model will focus on. The larger the C is, the higher accuracy the model is for the current train or test set. Hence, C is like an inversion of regularization parameter λ.

(b)



The train accuracy of my final model is 0.752, and the test accuracy is 0.74.

```python
######################################################################
# Question 1 (b)
######################################################################
data = pd.read_csv("Q1.csv")

train = data[:500]
test = data[500:]

train_x = train.iloc[:,:45]
train_y = train.iloc[:,-1]

test_x = test.iloc[:,:45]
test_y = test.iloc[:,-1]

C_grid = np.linspace(0.0001, 0.6, 100)
record = list() # Record log-loss of the kFold for the 100 C values
for c in C_grid:
    kFold = 10
    sub_record = list()
    for i in range(kFold):
        # define model
        classifier = LogisticRegression(C = c, penalty = "l1", solver= "liblinear", random_state = 0)

        # process dataset
        test_start = 50 * i
        test_end = test_start + 50

        train_grid = train[0: test_start].append(train[test_end:])
        test_grid = train[test_start: test_end]

        train_grid_x = train_grid.iloc[:,:45]
        train_grid_y = train_grid.iloc[:,-1]

        test_grid_x = test_grid.iloc[:,:45]
        test_grid_y = test_grid.iloc[:,-1]

        # fit  and predict
        classifier.fit(train_grid_x, train_grid_y)
        predicted_grid_y = classifier.predict_proba(test_grid_x)

        # calculate log_loss in every loop of kFold
        logloss = log_loss(test_grid_y, predicted_grid_y)

        sub_record.append(logloss)

    record.append(sub_record)


record_mean = [sum(i) / len(i) for i in record] # calculate the mean value for each C
choosen_C = C_grid[argmin(record_mean)] # choose c with the min(mean)
print(choosen_C) # 0.18794747474747472

log_loss_result = pd.DataFrame(columns=['C', 'log_loss'])

for index, sub_record in enumerate(record):
    for loss in sub_record:
        log_loss_result.loc[log_loss_result.shape[0] - 1]=[C_grid[index], loss]

sns.boxplot(x="C", y="log_loss", data=log_loss_result)
plt.savefig("Question 1(b).png")
plt.clf()

# Re-fit the model with this chosen C, and get the accuracy for both train and test via this model.
classifier = LogisticRegression(C = choosen_C, penalty = "l1", solver= "liblinear", random_state = 0)
classifier.fit(train_x, train_y)
print(classifier.score(train_x, train_y)) # 0.752
print(classifier.score(test_x, test_y)) # 0.74
```

(c)

Two parameters are different from the cross validation used in 1(b) from the document.

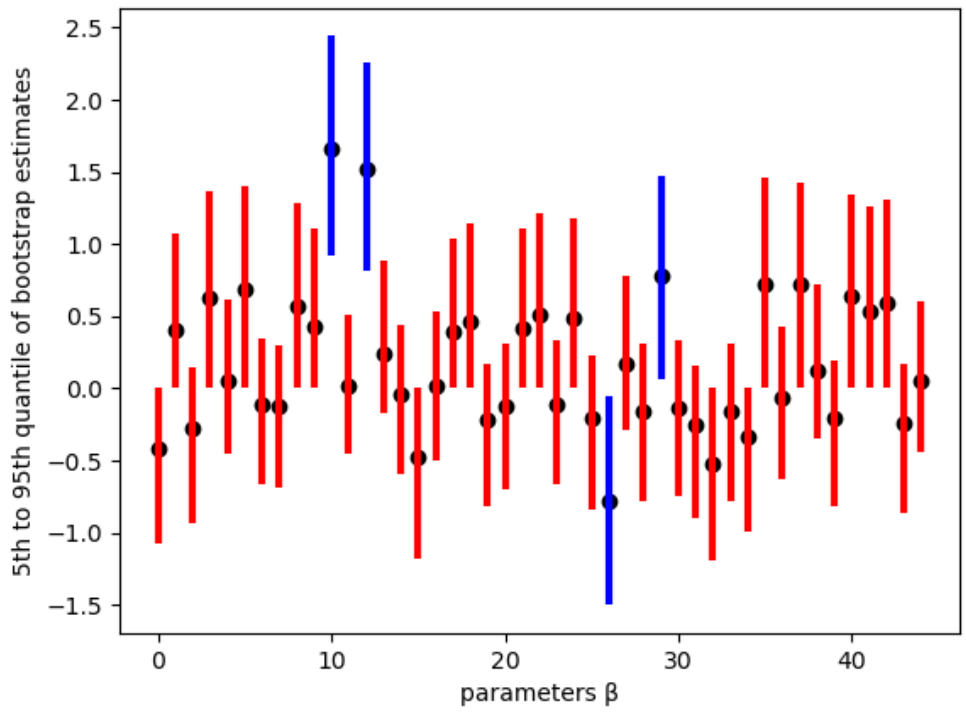
The first one is ‘scoring’, the default value for this parameter is ‘None’, and if I want to use a single score like log_loss used in 1(b), I have to define ‘scoring’ = ‘neg_log_loss’, which will provide the same predict_proc and the same log_loss function.

The second one is ‘cv’, which said if the estimator is a classifier, and the input is integer or none, the ‘StratifiedKFold’ will be used by default. The StratifiedKFold’ is a variation of KFold, but not exactly the same as KFold, it focuses on preserving the percentage. Hence, I modify the GridSearchCV to make it use the original KFold.

Therefore, they are the two reasons why cause this case.

```
86  ######################################################################
87  # Question 1 (c)
88  ######################################################################
89  C_grid = np.linspace(0.0001, 0.6, 100)
90  grid_lr = GridSearchCV(estimator = LogisticRegression(penalty='l1',solver='liblinear'),
91                         cv = KFold(n_splits=10),
92                         scoring = 'neg_log_loss',
93                         param_grid = {'C': C_grid})
94
95  grid_lr.fit(train_x, train_y)
96
97  print(grid_lr.best_params_) # original: {'C': 0.012219191919191918}, after modified: {'C': 0.18794747474747472}
```

(d)



```
######################################################################
# Question 1 (d)
######################################################################
np.random.seed(12)
c = 1
coefs = pd.DataFrame(columns=['X' + str(m) for m in range(train.shape[1] - 1)])
for i in range(10000):
    if i % 50 == 0:
        print("Process: " + str(i / 100) + "%")
    row = [np.random.choice(500) for j in range(500)] # 500 random index from [0, 499]
    train_bootstrap = train.loc[row] # train set for bootstrap
    train_bootstrap_x = train_bootstrap.iloc[:,:45]
    train_bootstrap_y = train_bootstrap.iloc[:,-1]

    classifier = LogisticRegression(C = c, penalty = "l1", solver= "liblinear", random_state = 0)
    classifier.fit(train_bootstrap_x, train_bootstrap_y)
    coefs.loc[coefs.shape[0] - 1] = np.squeeze(classifier.coef_, axis=0) # Append coef to coefs

quantile_5 = list()
quantile_95 = list()
avg = list()
for index, col in coefs.iteritems():
    quantile_5.append(np.percentile(sorted(col), 5))
    quantile_95.append(np.percentile(sorted(col), 95))
    avg.append(sum(col) / len(col))

x_lable = [i for i in range(45)]
colors_list = ['red' if quantile_5[i] * quantile_95[i] <= 0 else 'blue' for i in range(45)]
plt.vlines(x = x_lable, ymin = quantile_5, ymax = quantile_95, lw = 3, colors = colors_list, linestyles = '-')
plt.scatter(x_lable, avg, c= "black")
plt.xlabel("parameters β")
plt.ylabel("5th to 95th quantile of bootstrap estimates")
plt.savefig("Question 1(d).png")
plt.clf()
```

(e)

Most of the bars are red, and only four of them are blue. The confidence intervals tell me that which one is necessary for the model. If the majority of confidence intervals involve 0 for any β, it seems that this β should be discarded.

The parameter C in logistic regression is kind of an inversion of regularization.

Hence, if most bars are red, it means most β should be penalized. The regularization is necessary, and C also must be decreased to shrink the scope of confidence intervals.

If most bars are blue, it means most β should be included. The regularization is unnecessary and should decrease. The C should increase to enlarge the scope of most confidence intervals.

Question 2

(a)

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

$$\nabla f(x) = \frac{1}{2} * [(Ax - b)^{T\prime} \cdot (Ax - b) + (Ax - b)^T \cdot (Ax - b)']$$

$$= \frac{1}{2} * [(Ax - b)^{T\prime} \cdot (Ax - b) + (Ax - b)^{T\prime} \cdot (Ax - b)]$$

$$= \frac{1}{2} * 2 * (Ax - b)^{T\prime} \cdot (Ax - b) = A^T \cdot (Ax - b)$$

$x^{(1)} = x^{(0)} - 0.1 * A^T \cdot (Ax^{(0)} - b) = [[1.\ 0.5\ 0.\ 1.5]]^T$

$x^{(2)} = x^{(1)} - 0.1 * A^T \cdot (Ax^{(1)} - b) = [[\ 1.2\ \ 0.25\ -0.25\ 1.45]]^T$

$x^{(k+1)} = x^{(k)} - 0.1 * A^T \cdot (Ax^{(k)} - b)$

To make it more intuitive, I print it out in the form of transpose. I suppose the initial x [1,1,1,1] is not in the iteration, so I did not involve it in.

```
[[1.   0.5 0.   1.5]]
[[ 1.2    0.25 -0.25   1.45]]
[[ 1.345  0.125 -0.36   1.44 ]]
[[ 1.4565   0.0625 -0.4075   1.459 ]]
[[ 1.5499    0.03125 -0.4242    1.49205]]
[[ 3.99699850e+00 -2.59623079e-16 -5.61531549e-04  2.99812156e+00]]
[[ 3.99709142e+00 -2.15214158e-16 -5.44147417e-04  2.99817971e+00]]
[[ 3.99718146e+00 -2.59623079e-16 -5.27301471e-04  2.99823607e+00]]
[[ 3.99726872e+00 -2.15214158e-16 -5.10977048e-04  2.99829068e+00]]
[[ 3.99735328e+00 -3.04032000e-16 -4.95158004e-04  2.99834359e+00]]
```

```python
######################################################################
# Question 2 (a)
######################################################################
A = np.array([[1, 0, 1, -1],
              [-1, 1, 0, 2],
              [0, -1, -2, 1]])
b = np.array([[1],
              [2],
              [3]])
x = np.array([[1],
              [1],
              [1],
              [1]])
lr = 0.1

f = 0.5 * np.dot((np.dot(A, x) - b).T, (np.dot(A, x) - b)) # shape = (1, 1)
derivation_f = np.dot(A.T, (np.dot(A, x) - b)) # f's derivation

x_list = list() # to record the x for each iteration
# x_list.append(x) # I suppose the initial one is not in the iteration, so I did not involve it in.
while True:
    if np.linalg.norm(derivation_f) < 0.001: # 2 norm
        break
    x = x - lr * derivation_f # Update x
    derivation_f = np.dot(A.T, (np.dot(A, x) - b)) # Update derivation
    x_list.append(x)

for i in range(5): # the first 5 values of x_k
    print(x_list[i].T) # To make it more intuitive, I print it out in the form of transpose

for i in range(-5, 0): # the last 5 values of x_k
    print(x_list[i].T) # To make it more intuitive, I print it out in the form of transpose
```

(b)

$$
\begin{aligned}
&f(x-\alpha\nabla f(x)) \\
&= \tfrac{1}{2}\left(A(x-\alpha\nabla f(x))-b\right)^T\left(A(x-\alpha\nabla f(x))-b\right) \\
&= \tfrac{1}{2}\left(Ax-A\nabla f(x)\alpha-b\right)^T\left(Ax-A\nabla f(x)\alpha-b\right) \\
&= \tfrac{1}{2}\Big[(Ax)^T(Ax)-(Ax)^T(A\nabla f(x))\alpha-Axb-(A\nabla f(x))^T(Ax)\alpha+(A\nabla f(x))^T(A\nabla f(x))\alpha^2 \\
&\quad +(A\nabla f(x))^Tb\alpha-b^TAx+b^TA\nabla f(x)\alpha+b^Tb\Big] \\
&= \tfrac{1}{2}\Big[(A\nabla f(x))^T(A\nabla f(x))\alpha^2-(Ax)^T(A\nabla f(x))\alpha-(A\nabla f(x))^T(Ax)\alpha \\
&\quad +(A\nabla f(x))^Tb\alpha+b^TA\nabla f(x)\alpha \\
&\quad +(Ax)^T(Ax)-Axb-b^TAx+b^Tb\Big]
\end{aligned}
$$

From Question 2, I know that $A.shape=(m,n)$, $b.shape=(m,1)$, $x.shape=(n,1)$

Hence, $(Ax).shape=(m,1)$, $\nabla f(x).shape=(n,1)$, $(A\nabla f(x)).shape=(m,1)$

$(Ax)^T(A\nabla f(x)).shape=(A\nabla f(x))^T(Ax).shape=(1,1)$

Also $(A\nabla f(x))^Tb.shape=b^TA\nabla f(x).shape=(1,1)$

$$
\begin{aligned}
\text{Hence, } f(x-\alpha\nabla f(x)) = \tfrac{1}{2}\Big[&(A\nabla f(x))^T(A\nabla f(x))\alpha^2-2(Ax)^T(A\nabla f(x))\alpha+2(A\nabla f(x))^Tb\alpha \\
&+(Ax)^T(Ax)-Axb-b^TAx+b^Tb\Big]
\end{aligned}
$$

Hence $\alpha_k=\underset{\alpha\geq 0}{\operatorname{argmin}} f(x^{(k)}-\alpha\nabla f(x^{(k)}))$ can be transfered into: $\frac{\partial}{\partial\alpha}f(x^{(k)}-\alpha\nabla f(x^{(k)}))$

$=(A\nabla f(x^{(k)}))^T(A\nabla f(x^{(k)}))\alpha-(Ax^{(k)})^T(A\nabla f(x^{(k)}))+(A\nabla f(x^{(k)}))^Tb$, where $(A\nabla f(x^{(k)}))^T(A\nabla f(x^{(k)})).shape=(1,1)$

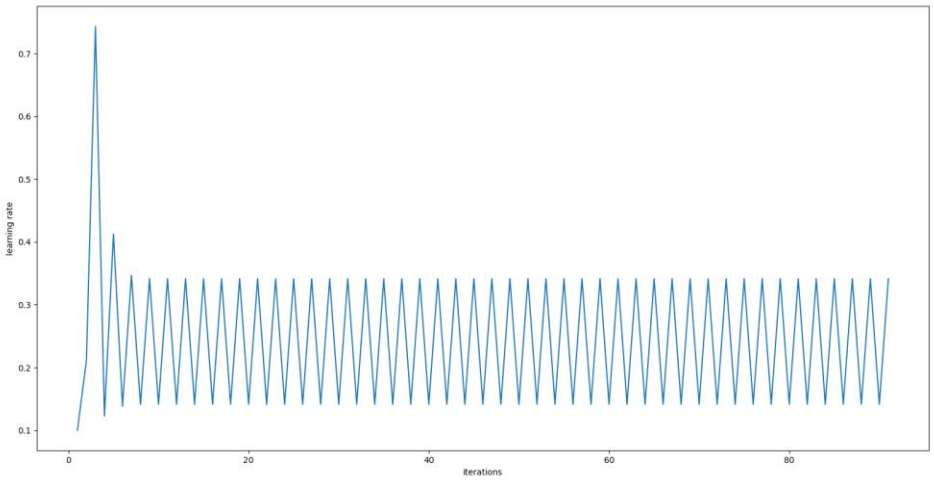
Let $\frac{\partial}{\partial\alpha}f(x^{(k)}-\alpha_k\nabla f(x^{(k)}))=0$

$$
\alpha_k=\frac{(Ax^{(k)})^T(A\nabla f(x^{(k)}))-(A\nabla f(x^{(k)}))^Tb}{(A\nabla f(x^{(k)}))^T(A\nabla f(x^{(k)}))}, \text{ I used this one in the code.}
$$

$$
=\frac{(Ax^{(k)})^T\left(A\cdot A^T(Ax^{(k)}-b)\right)-\left(A\cdot A^T(Ax^{(k)}-b)\right)^Tb}{\left(A\cdot A^T(Ax^{(k)}-b)\right)^T\left(A\cdot A^T(Ax^{(k)}-b)\right)}
$$

To make it more intuitive, I print it out in the form of transpose. I suppose the initial x [1,1,1,1] is not in the iteration, so I did not involve it in.

```
[[1.  0.5 0.  1.5]]
[[ 1.42271293 -0.02839117 -0.52839117  1.39432177]]
[[ 2.04509976  0.07709813 -0.18730912  1.65101238]]
[[ 2.06071834  0.02978136 -0.34806892  1.84620026]]
[[ 2.38888907 -0.03168527 -0.28257453  1.85898232]]
[[ 3.99692548e+00 -2.29737197e-16 -6.10498584e-04  2.99814648e+00]]
[[ 3.99733476e+00 -2.29737197e-16 -4.17136083e-04  2.99816903e+00]]
[[ 3.99737079e+00 -1.66928959e-16 -5.22075184e-04  2.99841494e+00]]
[[ 3.99772079e+00 -4.70119129e-16 -3.56718923e-04  2.99843423e+00]]
[[ 3.99775160e+00 -3.44502652e-16 -4.46458854e-04  2.99864452e+00]]
```




```python
#####################################################################
# Question 2 (b)
#####################################################################
A = np.array([[1, 0, 1, -1],
              [-1, 1, 0, 2],
              [0, -1, -2, 1]])
b = np.array([[1],
              [2],
              [3]])
x = np.array([[1],
              [1],
              [1],
              [1]])
lr = 0.1

f = 0.5 * np.dot((np.dot(A, x) - b).T, (np.dot(A, x) - b)) # shape = (1, 1)
derivation_f = np.dot(A.T, (np.dot(A, x) - b)) # f's derivation

x_list = list() # to record the x for each iteration
lr_list = list() # to record the lr for each iteration
# x_list.append(x) # I suppose the initial one is not in the iteration, so I did not involve it in.
lr_list.append(lr)
while True:
    if np.linalg.norm(derivation_f) < 0.001: # 2 norm
        break
    x = x - lr * derivation_f # Update x
    derivation_f = np.dot(A.T, (np.dot(A, x) - b)) # Update derivation
    lr = (np.dot(np.dot(A, x).T, np.dot(A, derivation_f)) - np.dot(np.dot(A, derivation_f).T, b)) / np.dot(np.dot(A, derivation_f).T, np.dot(A, derivation_f))
    x_list.append(x)
    lr_list.append(lr[0][0])

```

```
197
198 for i in range(5): # the first 5 values of x_k
199     print(x_list[i].T) # To make it more intuitive, I print it out in the form of transpose
200
201 for i in range(-5, 0): # the last 5 values of x_k
202     print(x_list[i].T) # To make it more intuitive, I print it out in the form of transpose
203
204 plt.plot(range(1, len(lr_list) + 1), lr_list)
205 plt.xlabel("iterations")
206 plt.ylabel("learning rate")
207 plt.savefig("Question 2(b).png")
208 plt.clf()
```

(c)

Steepest descent has a faster convergence rate. Gradient Descent needs more than 200 iterations to reach the termination condition. However, the steepest descent only needs less than 100 iterations to reach the condition.

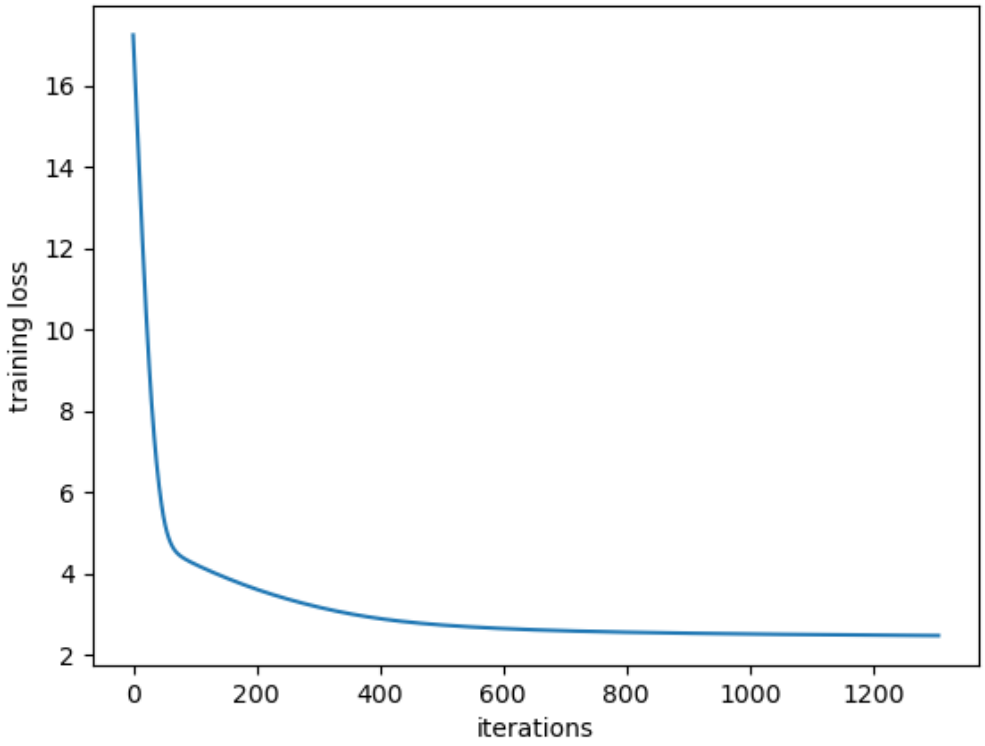
Two different methods, resulting in different learning rate for the two. Steepest descent can select the appropriate learning rate based on the current situation, rather than keeping the same learning rate as in gradient descent. Therefore, the learning rate of steepest descent tends to be larger in the early stages, increasing the training speed. In the later stage, it decreases to reduce the oscillation and convergence as soon as possible.

The gradient of the termination condition is small, so its gradient can be approximately considered to approach 0. Through the figure plotted, I can find that in the later period of training, learning rate is almost always oscillating, which means that the optimal or partial optimal solution has been reached, so it should terminate at the condition.

(d)

Nothing required to be put in this pdf.

(e)



From the forum, the coordinator tells that to choose k if abs(loss_k - loss_(k-1)) < condition.

Hence, there are 1307 iterations in total, and the final weight vector is [37.05697, -12.684172, -22.38835, 22.195482].

The train loss of my final model is 2.4737415, and the test is 2.6956608.

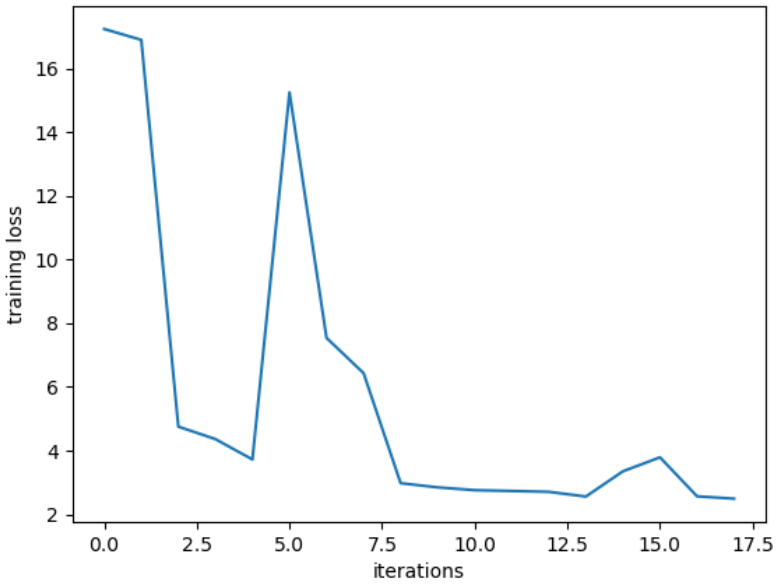
```
226 ######################################################################
227 # Question 2 (e)
228 ######################################################################
229 X_train_temp = np.insert(X_train, 0, values=1, axis=1) # Add a new first column with all the value 1.
230
231 def loss(w):
232     loss = jnp.sum((jnp.sqrt(0.25 * jnp.square(jnp.array(Y_train) - jnp.dot(X_train_temp, w.T).reshape(-1, 1)) + 1) - 1)) / Y_train.shape[0]
233     return loss
234
235 w = jnp.array([1.0,1.0,1.0,1.0])
236
237 iteration_counter = 0
238 lr = 1 # learning rate, step size
239 training_loss_list = list()
240 while True:
241     cur_loss = loss(w)
242     training_loss_list.append(cur_loss)
243     pre_loss = training_loss_list[len(training_loss_list) - 2] # when index < 0, take the first one, no exception occur.
244     W_grad = grad(loss)(w)
245     w = w - lr * W_grad # from the forum, Omar tells that to choose k if abs(loss_k - loss_(k-1)) < condition
246     iteration_counter += 1
247     if len(training_loss_list) > 1 and abs(cur_loss - pre_loss) < 0.0001: # the len(list) must be larger than 1
248         break
249 print(iteration_counter) # len(training_loss_list) = iteration_counter = 1037
250 print(w) # [37.05697, -12.684172, -22.38835, 22.195482]
251 print("Train loss: ", training_loss_list[-1]) # 2.4737415
252
253 X_test_temp = np.insert(X_test, 0, values=1, axis=1) # Add a new first column with all the value 1.
```

```python
X_test_temp = np.insert(X_test, 0, values=1, axis=1) # Add a new first column with all the value 1.

def test_loss(w):
    loss = jnp.sum((jnp.sqrt(0.25 * jnp.square(jnp.array(Y_test) - jnp.dot(X_test_temp, w.T).reshape(-1, 1)) + 1) - 1)) / Y_test.shape[0]
    return loss
print("Test loss", test_loss(w)) # 2.6956608

plt.plot(range(len(training_loss_list)), training_loss_list)
plt.xlabel("iterations")
plt.ylabel("training loss")
plt.savefig("Question 2(e).png")
plt.clf()
```

(f)



There are 18 iterations in total, and the final weight vector is [37.36413, -13.3802595, -20.870495, 21.694393]

The train loss of my final model is 2.4907434, and the test is 2.7091722.

```python
######################################################################
# Question 2 (f)
######################################################################
X_train_temp = np.insert(X_train, 0, values=1, axis=1) # Add a new first column with all the value 1.

def loss(w):
    loss = jnp.sum((jnp.sqrt(0.25 * jnp.square(jnp.array(Y_train) - jnp.dot(X_train_temp, w.T).reshape(-1, 1)) + 1) - 1)) / Y_train.shape[0]
    return loss

def lr_func(args):
    old_w, W_grad = args
    # argmin L(w_k - lr * W_grad), L => loss function
    v = lambda lr: (sum((np.sqrt(0.25 * np.square(np.array(Y_train) - np.dot(X_train_temp, (old_w - lr * W_grad).T).reshape(-1, 1)) + 1) - 1)))[0] / Y_train.shape[0]
    return v

w = jnp.array([1.0,1.0,1.0,1.0])

iteration_counter = 0
lr = 1 # learning rate, step size
training_loss_list = list()
while True:
    w_k = np.array(w) # lr need w_k, hence w_k should be stored before updated to w_(k+1)
    cur_loss = loss(w) # the current loss
    training_loss_list.append(cur_loss)
    W_grad = grad(loss)(w)
    w = w - lr * W_grad # Update the w_k to w_(k+1)

    args = (w_k, np.array(W_grad))
    lr_start = np.asarray((lr))
    optimizer = minimize(lr_func(args), lr_start, method='BFGS')
    lr = optimizer.x
```

```
    lr = optimizer.x

    iteration_counter += 1
    if cur_loss < 2.5: # the len(list) must be larger than 1
        break

print(iteration_counter) # len(training_loss_list) = iteration_counter = 18
print(w) # [37.36413, -13.3802595, -20.870495, 21.694393]
print("Train loss: ", training_loss_list[-1]) # 2.4907434

X_test_temp = np.insert(X_test, 0, values=1, axis=1) # Add a new first column with all the value 1.

def test_loss(w):
    loss = jnp.sum((jnp.sqrt(0.25 * jnp.square(jnp.array(Y_test) - jnp.dot(X_test_temp, w.T).reshape(-1, 1)) + 1) - 1)) / Y_test.shape[0]
    return loss
print("Test loss", test_loss(w)) # 2.7091722

plt.plot(range(len(training_loss_list)), training_loss_list)
plt.xlabel("iterations")
plt.ylabel("training loss")
plt.savefig("Question 2(f).png")
plt.clf()
```

(g)

I choose Momentum, which is an optimization strategy based on gradient descent.

Momentum is different from traditional gradient descent mainly because it introduces the concept of momentum. In the training process of traditional gradient descent, the initialization of the learning rate is generally small. In addition, due to batch learning and other conditions, gradient descent will have oscillation. Therefore, the training cost is greatly increased. Momentum is created mainly to reduce the vibration in the traditional gradient descent training process and try to ensure its movement trend is relatively stable.

$$v_w = \beta v_w + \frac{\partial C}{\partial w}$$

$v_b = \beta v_b + \frac{\partial C}{\partial b}$, where β is the coefficient of friction in momentum.

Therefore, Momentum introduces the friction coefficient and uses the exponential weighted average method to average the gradient in periods of time to reduce the oscillation and keep the main movement trend stable in the gradient descent process.

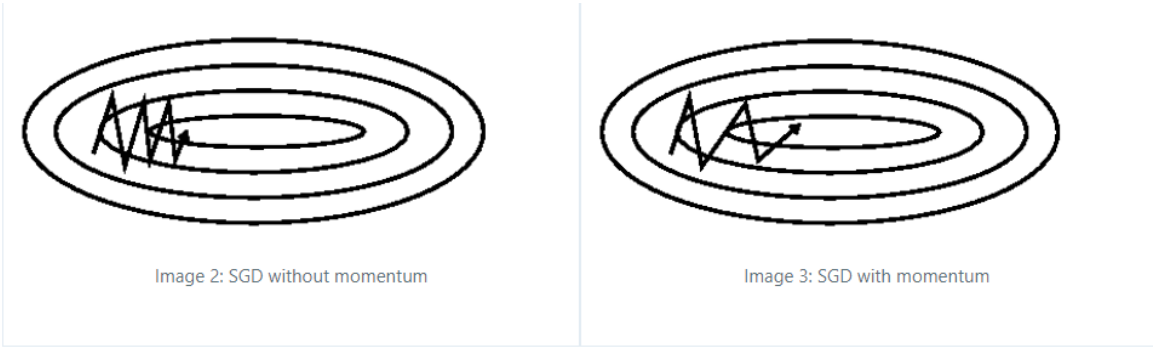


Image 2: SGD without momentum

Image 3: SGD with momentum

(Ruder, 2017)

The sourced used here are listed below:

Ruder, S., 2017. *SGD*. [image] Available at: <https://ruder.io/optimizing-gradient-descent/index.html#momentum> [Accessed 17 July 2021].

Ruder, S., 2017. *An overview of gradient descent optimization algorithms*. [online] ruder. Available at: <https://ruder.io/optimizing-gradient-descent/> [Accessed 17 July 2021].

Jun, P., 2020. *Momentum Algorithm in Deep Learning*. [online] Blog.csdn.net. Available at: <https://blog.csdn.net/gaoxueyi551/article/details/105238182> [Accessed 17 July 2021].