9417 homework 2

Name: Dong AO
ZID: z5305320

Q1:

(a).

Since the possible values of $\widetilde{y_k}$ are -1 and 1. We apply the sigmoid function to the new dataset for $\widetilde{y_k}\epsilon\{-1,1\}$.

We can write the form of probability as following (we suppose $z_k = w^T x_k + c$):

$$P(\ \widetilde{y_k} = 1|z_k) = \sigma(z_k) = \frac{1}{1+e^{-(z_k)}}\ \ and\ \ P(\ \widetilde{y_k} = -1|z_k) = 1 - \sigma(z_k) = \frac{1}{1+e^{(z_k)}}.$$

But we can write those more compactly as:

$$P(\ \widetilde{y_k} = 1|z_k) = \sigma(\widetilde{y_k}z_k) = \sigma(z_k)\ \ and\ \ P(\ \widetilde{y_k} = -1|x_k) = \sigma(\widetilde{y_k}z_k) = \sigma(-z_k)$$

because $\sigma(z_k) + \ \sigma(-z_k) = \frac{1}{1+e^{-z_k}} + \frac{1}{1+e^{z_k}} . = \frac{e^{z_k}}{1+e^{z_k}} + \frac{1}{1+e^{z_k}} = 1.$

Thus, we got: $P(\ \widetilde{y_k}|z_k) = \ \sigma(\widetilde{y_k}z_k) = \frac{1}{1+e^{-(\widetilde{y_k}z_k)}} = \frac{1}{1+e^{-\widetilde{y_k}(w^T x_k+c)}}.$

Similarly from how we got formular (1), apply the log loss here to get:

$L(\tilde{w},\tilde{c}) = -\log(\prod_{k=1}^{n} P(\ \widetilde{y_k}|z_k)) = -\sum_{k=1}^{n}\log\big(P(\ \widetilde{y_k}|z_k)\big) = -\sum_{k=1}^{n}\log\left(\frac{1}{1+e^{-\widetilde{y_k}(w^T x_k+c)}}\right)$

$= \sum_{k=1}^{n}\log\big(1+e^{-\widetilde{y_k}(w^T x_k+c)}\big).$

Then we apply the $l1$ regularization to the loss function, and we want to minimize the negative log loss with the penalty. Thus, we got:

$$(\tilde{w},\tilde{c}) = \underset{w,c}{\operatorname{argmin}}\{CL(\tilde{w},\tilde{c}) + penalty(w)\} = \underset{w,c}{\operatorname{argmin}}\left\{C\sum_{k=1}^{n}\log\big(1+e^{-\widetilde{y_k}(w^T x_k+c)}\big) + \big|\big|w\big|\big|_1\right\}(2).$$

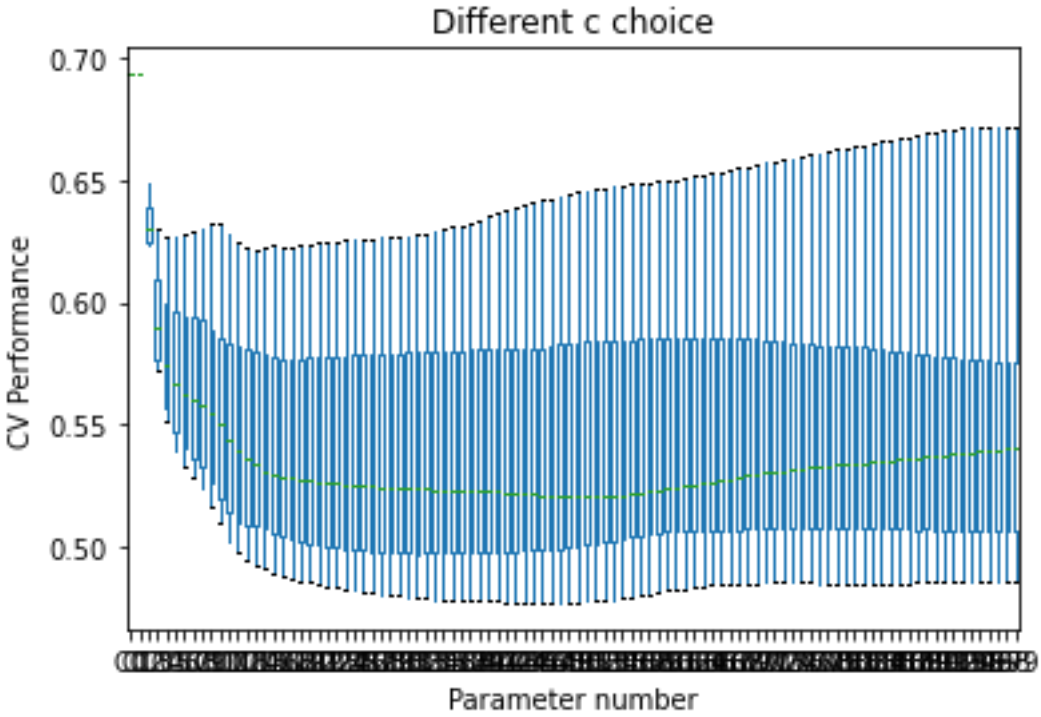
We got exactly the same objective as (2) in the question.

The difference between the two objectives (1) and (2) in this question is: for $y_k\epsilon\{0,1\}$, $P(y|z) = \sigma(z)^y\big(1-\sigma(z)\big)^{1-y}$, but for $\widetilde{y_k}\epsilon\{-1,1\}$, $P(y|z) = \sigma(\tilde{y}z)$. Except that, how we maximize the total probability, how we apply the negative log-loss and what the penalty is, are all the same. It is just the difference between the expression of probability, the essence after it is the same, we are counting the probability of two classes. It is just because the two classes' values which we are classifying are different. That is why the two objectives are identical.

In terms of the objective of C:

Since we are using 1-Norm penalty function, we need a coefficient C to adjust the influence of the penalty function. When C is small, then the influence of penalty will be huge, otherwise it will be small. For the parameter λ in LASSO, it is also the regularization parameter. They are both set to avoid overfitting. The only difference is the positions of them are different and when we are adjusting them, the influence of regularization in logistic regression becomes larger as C increases but becomes smaller when λ increases.

(b).




```
my choice of C: 0.18794747474747472
Test score: 7.519999999999999  Train_score:7.400000000000001
```

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

from sklearn.model_selection import train_test_split
# load dataset
data = pd.read_csv('Q1.csv')
data.head()
numerics = ['int16', 'int32', 'int64', 'float16', 'float32', 'float64']
numerical_vars = list(data.select_dtypes(include=numerics).columns)
data = data[numerical_vars]
data.shape
data_train , data_test = train_test_split(
    data,
    #data.drop(labels=['Y'], axis=1),
    #data['Y'],
    train_size=500,
    shuffle=False)
```

```
C = np.linspace(0.0001, 0.6, 100)
```

```
import sklearn.model_selection as cv
#train = cv.KFold(n_splits=5)

cross_train = []
cross_test = []

for i in range(10):
    datatmp = data_train
    for rows in range(0 + 50*i, 50+50*i):
        datatmp = datatmp.drop(rows,axis=0)
    cross_test.append(data_train.loc[0 + 50 * i: 49 + 50 * i])
    cross_train.append(datatmp)
```

```
from sklearn.linear_model import LogisticRegression
from sklearn import metrics
scores = []
losses = []
mean_scores = []
best_c = 0
```

```
for c in C:
    score = []
    classifier = LogisticRegression(solver='liblinear', C=c, penalty='l1')
    loss = []
    values = 0
    for i in range(10):
        train_x = cross_train[i].drop(labels=['Y'], axis=1)
        train_y = cross_train[i]['Y']
        test_x = cross_test[i].drop(labels=['Y'], axis=1)
        test_y = cross_test[i]['Y']
        classifier.fit(train_x, train_y)

        pre_y = classifier.predict(test_x)
        values += pre_y
        pro_y = classifier.predict_proba(test_x)
        #print(pre_y)
        loss.append(metrics.log_loss(test_y, pro_y))
    losses.append(loss)
    #score.append(classifier.score(test_x, test_y))
    mean_score = np.mean(loss)
    mean_scores.append(mean_score)
```

```
#print(mean_scores)
cmax = mean_scores.index(min(mean_scores))
print("my choice of C:", C[cmax])
score_test = 0
score_train = 0
data_test_x = data_test.drop(labels=['Y'], axis=1)
data_test_y = data_test['Y']
for i in range(10):
    classifier = LogisticRegression(solver='liblinear', C=C[cmax], penalty='l1')

    train_x = cross_train[i].drop(labels=['Y'], axis=1)
    train_y = cross_train[i]['Y']
    test_x = cross_test[i].drop(labels=['Y'], axis=1)
    test_y = cross_test[i]['Y']
    classifier.fit(train_x, train_y)
    score_test += classifier.score(test_x, test_y)
    score_train += classifier.score(data_test_x, data_test_y)

print(f"Test score: {score_test}  Train_score:{score_train}")
```

```
import matplotlib.pyplot as plt
#scores = pd.DataFrame(scores)
#print(scores.values)
#print(clabels)
#print(losses)
mydata = {}
for i in range(0,100):
    mydata['C' + str(i)] = losses[i]
    #plt.boxplot(scores.values[i], labels= string)

df = pd.DataFrame(mydata)
df.plot.box(title="Different c choice")
plt.xlabel("Parameter number")
plt.ylabel("CV Performance")
plt.show()
```

(c).

The outcomes are different because:

1. GridSearchCV uses stratified as cv by default when the estimator is classifier, which is different from our choice of data: every 50 rows as one part. StratifiedKFold method will choose random 50 data from 500, which causes the difference.
2. The score method that GridSearchCV uses by default is different from ours, since we are using neg_log_loss method in (b). There are a lot of score method we can choose.

We can solve this problem and make the outcomes consistent by setting the parameter of GridSearchCV:

1. Set scoring = 'neg_log_loss'.
2. Use KFold(n_splits=10) instead of the StratifiedKFold by default.

```
Test score: 0.74
Train score: 0.752
best parameter:  {'C': 0.012219191919191918}
```

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

from sklearn.model_selection import train_test_split
# load dataset
data = pd.read_csv('Q1.csv')
data.head()
numerics = ['int16', 'int32', 'int64', 'float16', 'float32', 'float64']
numerical_vars = list(data.select_dtypes(include=numerics).columns)
data = data[numerical_vars]
data.shape
data_train, data_test = train_test_split(
    data,
    #data.drop(labels=['Y'], axis=1),
    #data['Y'],
    train_size=500,
    shuffle=False)

x_train = data_train.drop(labels=['Y'], axis=1)
y_train = data_train['Y']
x_test = data_test.drop(labels=['Y'], axis=1)
y_test = data_test['Y']
```

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV

param_grid = {'C': np.linspace(0.0001,0.6,100)}

grid_lr = GridSearchCV(estimator= LogisticRegression(penalty='l1', solver='liblinear'),
                       param_grid=param_grid, cv=10)
grid_lr.fit(x_train, y_train)
print()
```

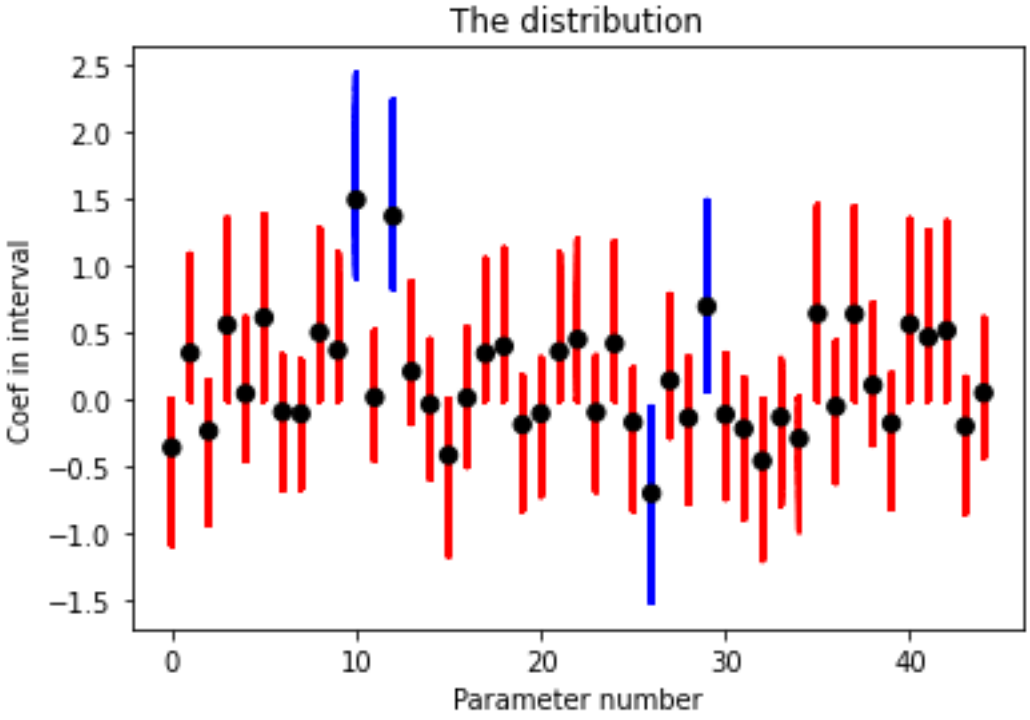
```
print(f"Test score: {grid_lr.score(x_test, y_test)}")
print(f"Train score: {grid_lr.score(x_train, y_train)}")
print("best parameter: ", grid_lr.best_params_)
```

```
import sklearn.model_selection as cv
#skf = ShuffleSplit(n_splits=3, random_state=0)
skf = cv.KFold(n_splits=10)
grid_lr = GridSearchCV(estimator= LogisticRegression(penalty='l1', solver='liblinear'),
                       param_grid=param_grid, cv=skf, scoring='neg_log_loss')
grid_lr.fit(x_train, y_train)
print()
print("best parameter: ", grid_lr.best_params_)
```

```
best parameter:  {'C': 0.18794747474747472}
```

```
# from sklearn import metrics
# best_model = grid_lr.best_estimator_
# predict_y=best_model.predict(x_train)
classifier = LogisticRegression(solver='liblinear', C=grid_lr.best_params_['C'], penalty='l1')
classifier.fit(x_train, y_train)
print(f"Test score: {classifier.score(x_test, y_test)}")
print(f"Train score: {classifier.score(x_train, y_train)}")
```

(d).




```
coefs = []
for i in range(45):
    coefs.append([])

B = 10000
np.random.seed(12)
for l in range(10000):
    data_boot = data_train.iloc[np.random.randint(0,500,500),]
    #print(data_boot)
    #data_boot = data_train.iloc[boot,]
    x_boot = data_boot.drop(labels=['Y'], axis=1)
    y_boot = data_boot['Y']
    classifier = LogisticRegression(solver='liblinear', C=1, penalty='l1')
    classifier.fit(x_boot, y_boot)
    for i in range(len(classifier.coef_[0])):
        coefs[i].append(classifier.coef_[0][i])
#print(coefs[0])
```

```
coef_mean = []
coef_std = []
for i in range(45):
    coef_mean.append(np.mean(coefs[i]))
    coef_std.append(np.std(coefs[i],ddof=1))
```

```
import matplotlib.pyplot as plt
from scipy import stats
alpha = 0.95
B = 10000
#print(coefs[i])
dots_y = []
dots_x = []
red_cols = []
mean_cols = []
for i in range(45):
    #conf_intveral = stats.norm.interval(0.95, loc = coef_mean[i], scale=coef_std[i])
    some_coef = coefs[i]
    some_coef = np.sort(coefs[i])
    lower = some_coef[int(B * 0.05) - 1]
    higher = some_coef[int(B * 0.95) - 1]
    #print(lower, higher)
    #lower = max(0,np.percentile(coefs[i], 10))
```

(d): code part1

```
    lower = some_coef[int(B * 0.05) - 1]
    higher = some_coef[int(B * 0.95) - 1]
    #print(lower, higher)
    #lower = max(0,np.percentile(coefs[i], 10))
    if (lower <= 0 and higher >= 0):
        red_cols.append(i)
    mean = 0
    for l in range(len(some_coef)):
        if (some_coef[l] >= lower and some_coef[l] <= higher):
            mean += some_coef[l]
            dots_x.append(i)
            dots_y.append(some_coef[l])
    mean = mean / len(some_coef)
    #print(mean)
    dots_x.append(i)
    dots_y.append(mean)
    mean_cols.append(len(dots_x))
    #lower = max(0,np.percentile(coefs[i], 10))
    #upper = min(np.percentile(coefs[i], 90), 1)
    #print('%.1f confidence interval %.1f%% and %.1f%%' % (alpha*100, lower*100, upper*100))
    #stats.append(abs(lower))
#plt.bar(range(45), stats)
#plt.show()
colors = []
def getColor():
    for i in range(len(dots_x)):
        if (i + 1) in mean_cols:
            colors.append("k")
        elif dots_x[i] in red_cols :
            colors.append("r")
        else:
            colors.append("b")
getColor()
plt.title("The distribution")
plt.xlabel("Parameter number")
plt.ylabel("Value")

plt.scatter(dots_x, dots_y,s=[40 if (i + 1) in mean_cols else 1 for i in range(len(dots_x))],
            c=colors)
plt.show()
```

(d): code part2

(e).

The parameters we got from the distribution of coefficients are mainly around 1 to -1. And for most of the features, their computed 90% interval contains 0, which means except the four features with blue bar, all the features are excluded from our model. When we choose a C smaller than 1, the effects of regularization should become larger, which means more features should be picked up to increase the weight of penalty because we want to minimize $(\hat{\beta}_0, \hat{\beta})$. And when we choose a C bigger than 1, the weight of loss function should become larger, which means less features should be chosen to minimize $(\hat{\beta}_0, \hat{\beta})$. For this dataset, we found that most features are not selected, which means they are useless to the model. We need $l1$ regularization to help with us select the features that we really need and then we can build the logistic regression model correctly. Thus, it is very necessary to use regularisation on this data.

Q2:

(a):

Gradient steps: $x^{(k+1)} = x^{(k)} - \alpha_k\left(A^T(Ax-b)\right) = x^{(k)} - \frac{\left(A^T(Ax-b)\right)}{10}$.

```
import numpy as np
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]])
alpha = 0.1

k = 0
all_xs = []
while 1 == 1:
    diff_x = np.dot(A.T, (np.dot(A, x) - b))
    x = x - alpha * diff_x
    x_norm = np.linalg.norm(diff_x, ord=2, axis=None)
    if (x_norm < 0.001):
        break
    all_xs.append(x)
    k += 1


for i in range(5):
    print(f"k={i},  x({i})={all_xs[i].reshape(4)} ")
for i in range(k - 5, k):
    print(f"k={i},  x({i})={all_xs[i].reshape(4)} ")
```

```
k=0,   x(0)=[1.    0.5  0.    1.5]
k=1,   x(1)=[ 1.2    0.25 -0.25   1.45]
k=2,   x(2)=[ 1.345   0.125 -0.36    1.44 ]
k=3,   x(3)=[ 1.4565   0.0625 -0.4075   1.459 ]
k=4,   x(4)=[ 1.5499    0.03125 -0.4242    1.49205]
k=217,   x(217)=[ 3.99699850e+00 -2.59623079e-16 -5.61531549e-04  2.99812156e+00]
k=218,   x(218)=[ 3.99709142e+00 -2.15214158e-16 -5.44147417e-04  2.99817971e+00]
k=219,   x(219)=[ 3.99718146e+00 -2.59623079e-16 -5.27301471e-04  2.99823607e+00]
k=220,   x(220)=[ 3.99726872e+00 -2.15214158e-16 -5.10977048e-04  2.99829068e+00]
k=221,   x(221)=[ 3.99735328e+00 -3.04032000e-16 -4.95158004e-04  2.99834359e+00]
```

(b):

We know that $f(x) = \frac{1}{2}||Ax-b||_2^2$.

And $f(x) = \frac{1}{2}(Ax-b)^T(Ax-b) = \frac{1}{2}(x^TA^TAx - 2b^TAx + b^Tb)$.

Thus $f'(x) = \nabla f(x) = \frac{1}{2}(A^TA + A^TA)x - \frac{1}{2} * 2A^Tb + 0 = A^TAx - A^Tb = A^T(Ax-b)$.

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

$$= \frac{1}{2}\left|\left|A\left(x^{(k)} - \alpha A^T A x^{(k)} + \alpha A^T b\right) - b\right|\right|_2^2 = \frac{1}{2}\left|\left|Ax^{(k)} - \alpha AA^T Ax^{(k)} + \alpha AA^T b - b\right|\right|_2^2.$$

Thus,

$$f\left(x^{(k)} - \alpha \nabla f\left(x^{(k)}\right)\right) = \frac{1}{2}\left|\left|-\alpha AA^T Ax^{(k)} + \alpha AA^T b - b + Ax^{(k)}\right|\right|_2^2$$

$$= \frac{1}{2}\left|\left|\left(AA^T b - AA^T Ax^{(k)}\right)\alpha + Ax^{(k)} - b\right|\right|_2^2.$$

We suppose $C = \left(AA^T b - AA^T Ax^{(k)}\right)$ and $D = -Ax^{(k)} + b$ $.(C, D$ are all in $R^4)$

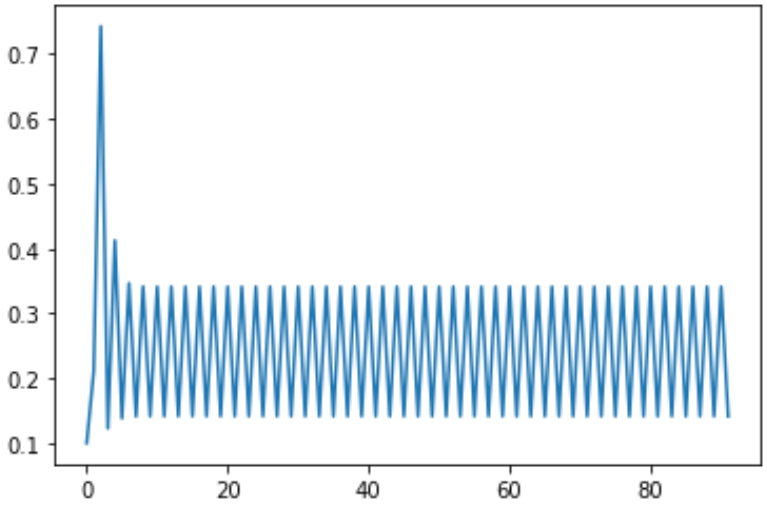
Thus, we have $f\left(x^{(k)} - \alpha \nabla f\left(x^{(k)}\right)\right) = \frac{1}{2}\left|\left|C\alpha - D\right|\right|_2^2$.

We are calculating $\alpha_k = arg\min_{\alpha \geq 0} f\left(x^{(k)} - \alpha \nabla f\left(x^{(k)}\right)\right)$. We have:

$$\alpha_k = arg\min_{\alpha \geq 0}(f\left(x^{(k)} - \alpha \nabla f\left(x^{(k)}\right)\right)) = arg\min_{\alpha \geq 0}(\frac{1}{2}\left|\left|C\alpha - D\right|\right|_2^2)$$

We take the derivative of $f\left(x^{(k)} - \alpha \nabla f\left(x^{(k)}\right)\right)$, similarly we got $f'\left(x^{(k)} - \alpha \nabla f\left(x^{(k)}\right)\right) = C^T C\alpha - C^T D$.

Thus, the minimal $\alpha = \frac{C^T D}{C^T C}$, where $C = \left(AA^T b - AA^T Ax^{(k)}\right)$ and $D = b - Ax^{(k)}$.




```
k=0,   x(0)=[1 1 1 1]
k=1,   x(1)=[1.   0.5 0.   1.5]
k=2,   x(2)=[ 1.42271293 -0.02839117 -0.52839117  1.39432177]
k=3,   x(3)=[ 2.04509976  0.07709813 -0.18730912  1.65101238]
k=4,   x(4)=[ 2.06071834  0.02978136 -0.34806892  1.84620026]
k=86,   x(86)=[ 3.99692548e+00 -7.04138187e-17 -6.10498584e-04  2.99814648e+00]
k=87,   x(87)=[ 3.99733476e+00 -7.04138187e-17 -4.17136083e-04  2.99816903e+00]
k=88,   x(88)=[ 3.99737079e+00 -7.04138187e-17 -5.22075184e-04  2.99841494e+00]
k=89,   x(89)=[ 3.99772079e+00  8.11812664e-17 -3.56718923e-04  2.99843423e+00]
k=90,   x(90)=[ 3.99775160e+00  1.83730279e-17 -4.46458854e-04  2.99864452e+00]
```

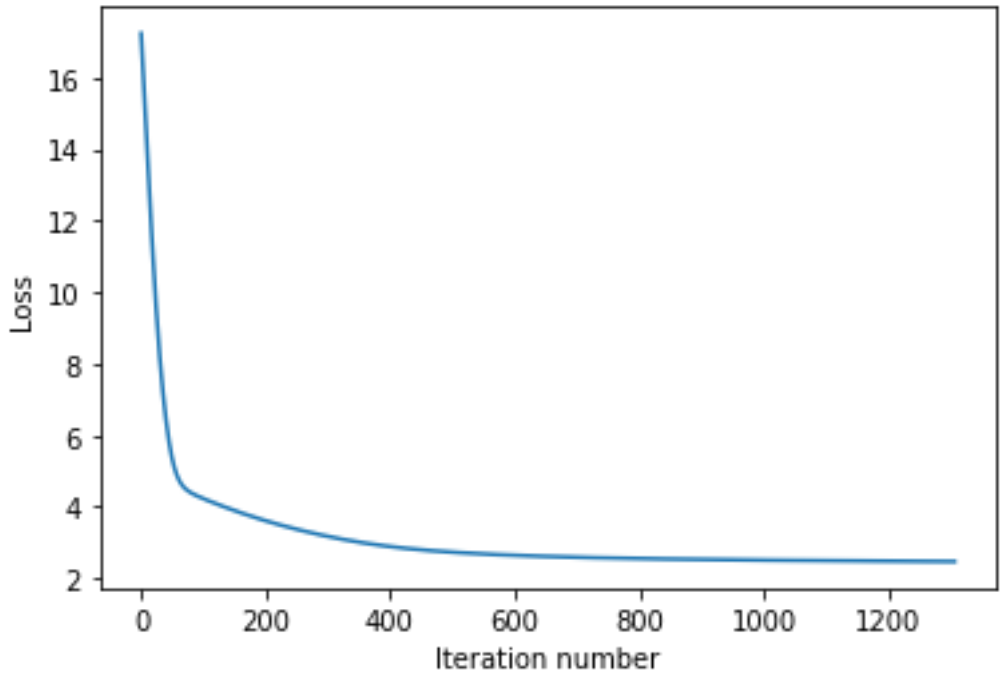
```python
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

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]])
alpha = 0.1

k  = 0
all_xs = []
alphas = []
alphas.append(0.1)
all_xs.append(x)
while 1 == 1:
    diff_x = np.dot(A.T,(np.dot(A, x) - b))
    x = x - alpha * diff_x
    x_norm = np.linalg.norm(diff_x, ord=2, axis=None)
    all_xs.append(x)
    C = np.dot(np.dot(A,A.T),b) -  np.dot(np.dot(np.dot(A,A.T),A), x)
    D = b - np.dot(A,x)
    alpha = np.dot(C.T, D)/np.dot(C.T, C)
    alphas.append(alpha[0][0])
    if (x_norm < 0.001):
        break
    k += 1


print(x_norm)
for i in range(5):
    print(f"k={i},  x({i})={all_xs[i].reshape(4)} ")
for i in range(k - 4, k + 1):
    print(f"k={i},  x({i})={all_xs[i].reshape(4)} ")

plt.plot(alphas)
plt.show()
```

(c):
The gradient descent takes much more iterations than the steepest descent, which is about 2 times larger than that of steepest descent. It means it will cost much more time and resources to calculate, that is why we prefer steepest descent here instead of gradient descent.

By looking at the plot of alpha in steepest descent, I found that alpha oscillates bet ween two values after the 5th iteration and it changes sharply from the last value. When it finds the convergent value, its result is smaller than the result of gradient d escent ($\left|\left|\nabla f\left(x^{(k)}\right)\right|\right|_2 = 0.0009709409289596629$ for steepest descent and 0.0009786977622281967 for gradient descent). When I need to have a more accurate way to find the result, I prefer gradient descent.

This is a reasonable condition because by looking at the last 5 values of x, we can see the changes of x are already tiny, which means if we set a less terminate condition, it will take a huge amount of iteration to converge and meet the terminate. Meanwhile, if we set a larger terminate condition, it the accuracy of our descent will decrease since we have only taken 221 and 89 times of iterations.

(e):



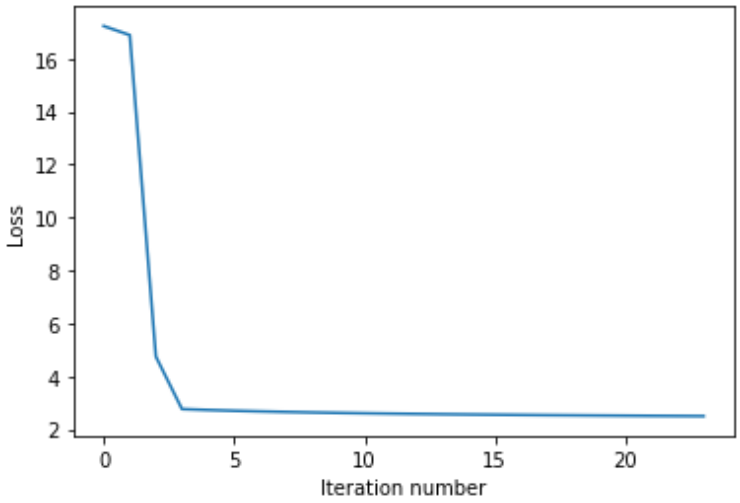

```
The final weight vector is[ 37.056965 -12.684172 -22.388344  22.195488]
The train loss is 2.4736413955688477
The test loss is 2.6956610679626465
```

```
import jax.numpy as jnp
from jax import grad
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn import preprocessing
%matplotlib inline
from sklearn.model_selection import train_test_split


data = pd.read_csv('Q2.csv')
data = data[~np.isnan(data).any(axis=1)]
data_x = data.drop(labels=['transactiondate','latitude','longitude','price'],axis=1)
data_y = data['price']

min_max_scaler = preprocessing.MinMaxScaler()
data_x = min_max_scaler.fit_transform(data_x)
data_x = pd.DataFrame(data_x)
train_x,test_x,train_y,test_y = train_test_split(
    data_x,
    data_y,
    train_size = 0.5,
    shuffle=False)
```

```
w = np.array([1,1,1,1], dtype=float)
def lossFunc(w_t, x, y):
  return np.sum( (jnp.sqrt(((y - jnp.dot(w_t, x.T))**2)/4 + 1) - 1) ) / 204


length = train_x.shape[0]
w = np.array([1,1,1,1], dtype=float)
iteration = 0

one_test_x = test_x.values
one_test_x = np.insert(one_test_x, 0, 1.0, axis=1)
one_train_x = train_x.values
one_train_x = np.insert(one_train_x, 0, 1.0, axis=1)

x_k = []
x_k.append(w)
loss_k = []
while 1 == 1:
  W_grad = grad(lossFunc, argnums=0, allow_int=True)(w.T, one_train_x, train_y.values)
  loss_k.append(lossFunc(w.T, one_train_x, train_y.values))

  w = w - W_grad
  x_k.append(w)
  iteration += 1
  if (iteration > 1):
    if (np.abs(loss_k[-1] - loss_k[-2]) < 0.0001):
      break
print(iteration)
```

```
print(f"The total iteration:{iteration}")
print(f"The final weight vector is{x_k[-1]}")
print(f"The train loss is {lossFunc(x_k[-1].T, one_train_x, train_y.values)}\nThe test loss is \
{lossFunc(x_k[-1].T, one_test_x, test_y.values)}")
import matplotlib.pyplot as plt
%matplotlib inline
plt.plot(loss_k)
plt.xlabel("Iteration number")
plt.ylabel("Loss")
plt.show()
```

(f):




```
The final weight vector is[ 36.262196 -13.278603 -20.989244  23.271177]
The train loss is 2.4960882663726807
The test loss is 2.7308456897735596
```

```
import jax.numpy as jnp
from jax import grad
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn import preprocessing
from scipy.optimize import minimize,minimize_scalar
%matplotlib inline
from sklearn.model_selection import train_test_split


data = pd.read_csv('Q2.csv')
data = data[~np.isnan(data).any(axis=1)]
data_x = data.drop(labels=['transactiondate','latitude','longitude','price'],axis=1)
data_y = data['price']

min_max_scaler = preprocessing.MinMaxScaler()
data_x = min_max_scaler.fit_transform(data_x)
data_x = pd.DataFrame(data_x)
train_x,test_x,train_y,test_y = train_test_split(
    data_x,
    data_y,
    train_size = 0.5,
    shuffle=False)
```

```
def lossFunc(w_t, x, y):
  return (np.sum( (    (  (  (y - (w_t @ x.T) )**2 ) /4 + 1)**(1/2) - 1) ) / 204).astype(float)

def find_least_alpha(alpha, w_t, x, y):
  grad_loss = grad(lossFunc, argnums=0, allow_int=True)(w_t, one_train_x, train_y.values)
  new_w = w_t - alpha * grad_loss
  return float(np.sum(((((y - jnp.dot(new_w, x.T))**2)/4 + 1)**(1/2) - 1) ) / 204)

one_test_x = test_x.values
one_test_x = np.insert(one_test_x, 0, 1.0, axis=1)
one_train_x = train_x.values
one_train_x = np.insert(one_train_x, 0, 1.0, axis=1)
w = np.array([1,1,1,1], dtype=float)


loss_k = []

def optimize(x_k):

  w = np.array([1,1,1,1], dtype=float)
  x_k.append(w)

  alpha = 1
  iteration = 0
  while 1 == 1:
    W_grad = grad(lossFunc, argnums=0, allow_int=True)(w.T, one_train_x, train_y.values)
    loss_k.append(lossFunc(w.T, one_train_x, train_y.values))

    w = w - alpha * W_grad
    x_k.append(w)
    print(loss_k[-1], "  ", alpha, "  ", w)
    # mini = minimize(find_least_alpha, x0=alpha, method="BFGS", args=(x_k[-1].T, one_train_x, train_y.values))
    mini = minimize_scalar(find_least_alpha , args=(x_k[-1].T, one_train_x, train_y.values))
    alpha = mini.x
    iteration += 1
    if (iteration > 1):
      if (loss_k[-1] < 2.5):
        return iteration
x_k = []
iteration = optimize(x_k)
print(iteration)
```

```
print(f"The total iteration:{iteration}")
print(f"The final weight vector is{x_k[-2]}")
print(f"The train loss is {lossFunc(x_k[-2].T, one_train_x, train_y.values)}\nThe test loss is \
{lossFunc(x_k[-1].T, one_test_x, test_y.values)}")
import matplotlib.pyplot as plt
%matplotlib inline
plt.plot(loss_k)
plt.xlabel("Iteration number")
plt.ylabel("Loss")
plt.show()
```

(g):

I am going to talk about mini-batch gradient descent method. This method is trying to achieve a balance between the methods of batch gradient descent (BGD) and stochastic gradient descent(SGD). The method split the training set into a lot of batches and we will only use parts of them to calculate the errors and update the parameters. If every batch size is 1, this is exactly the same as what we just did: stochastic gradient descent. There is only one data item been trained to update the parameters. If batch size is the total length of the dataset, then it is the same as batch gradient descent.

The advantages of the method are:

1. It updates the parameters much more frequently than BGD, so it is easier to achieve a more robust convergence. Thus, at the same time, it can relatively avoid to reaching the local minima.

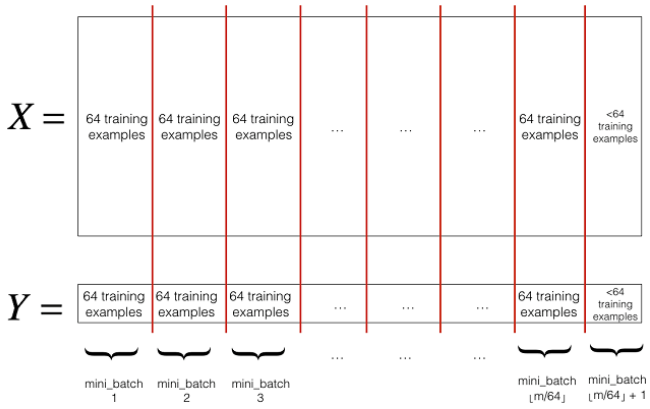
2. The update of algorithm provides a much more efficient process then the SGD.
3. The algorithm does not need to read all the data into the RAM, which can save a lot of memory.

The disadvantages:

1. There is one more hyperparameter “mini-batch size” in the algorithm, which means we have to adjust one more hyperparameters.
2. We have to add up all the errors from the previous mini-batches of training examples.

Then, I will just talk about how to configure the algorithm. First, we need to choose a starter value of batch-size, which is normally set to 32 because most computing hardware has memory of 32 or its multiplication. For example, Windows has two version of 32-bit and 64-bit operating system. It is convenient for the GPU or CPU to accelerate the computation.

After we choose an appropriate batch size, we divide the dataset into batches.



We use the batches to update the parameters and the cost.

```
seed = 0
for i in range(0, num_iterations):
    # Define the random minibatches. We increment the seed to reshuffle differently the dataset after each epoch
    seed = seed + 1
    minibatches = random_mini_batches(X, Y, mini_batch_size, seed)
    for minibatch in minibatches:
        # Select a minibatch
        (minibatch_X, minibatch_Y) = minibatch
        # Forward propagation
        AL, caches = forward_propagation(minibatch_X, parameters)
        # Compute cost
        cost = compute_cost(AL, minibatch_Y)
        # Backward propagation
        grads = backward_propagation(AL, minibatch_Y, caches)
        parameters = update_parameters(parameters, grads, learning_rate)
```

Figure 1: Mini-batch Gradient Descent Code [1]

From the plots, we can see that the results correspond with our analysis: mini-batch Gradient Descent is relatively more stable than SGD. SGD makes a little progress vertically but not horizontally. MBGD optimize the process, reduce the variants of the updated parameters and make the updates more stable.

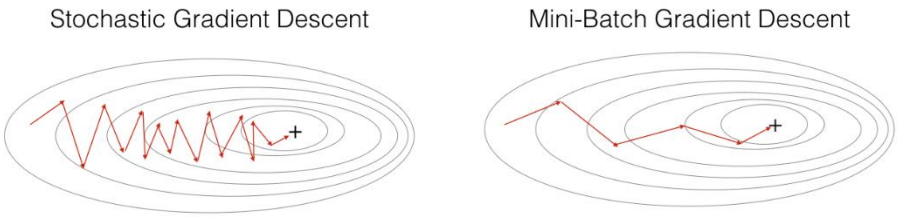


Figure 2: Comparison between Mini-batch Gradient Descent and SGD [2]

Reference:

[1]. https://blog.csdn.net/u012328159/article/details/80252012

[2]. https://zhuanlan.zhihu.com/p/42479917