EE608 Exam 2

Barry Xiong 10471474

Q1.

Steepest Descent algorithm:

As we discussed in class

Step 1. Choose an starting point x_0= $[1\ 1\ 1]^T$ and and a stopping threshold $\epsilon = 10^{-6}$.

Step 2. Calculate the gradient of the objective function as:

$$\frac{df}{dx_1} = 2\left(2x_1x_2^2 + 4x_1x_3^2 + x_1 + 5\right)$$

$$\frac{df}{dx_2} = 2\left(2x_1^2x_2 + x_2 + 8\right)$$

$$\frac{df}{dx_3} = 2\left(4x_1^2 + x_3 + 7\right)$$

And set the search direct to be the negative of the gradient i.e. $d_k = -\frac{df}{dx}$

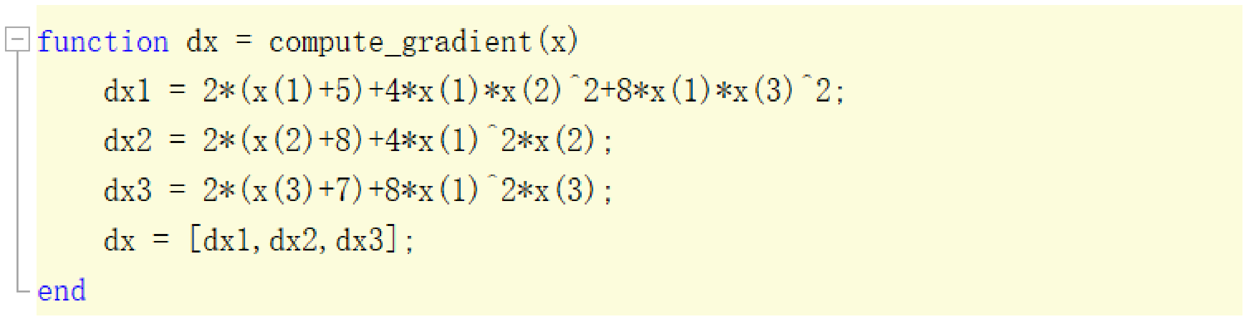
Step 3. Pick $\alpha = 0.001$ The value is chosen based on trials and error of the result from the algorithm.

Step 4. Update $x^{(k+1)} = x^{k} + \alpha * d_k$

Step 5. Continue the algorithm until $\left|\alpha * d_k\right| < \epsilon$

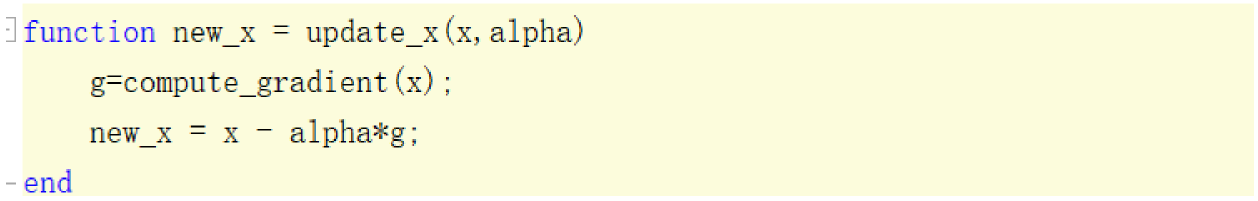
Below are the implementations of this algorithm:

1. **Compute gradient**



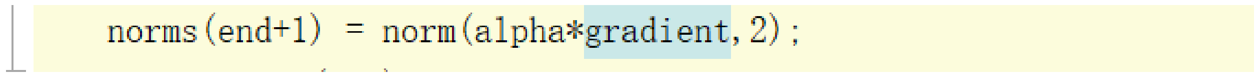
```
function dx = compute_gradient(x)
    dx1 = 2*(x(1)+5)+4*x(1)*x(2)^2+8*x(1)*x(3)^2;
    dx2 = 2*(x(2)+8)+4*x(1)^2*x(2);
    dx3 = 2*(x(3)+7)+8*x(1)^2*x(3);
    dx = [dx1,dx2,dx3];
end
```

2. Update x



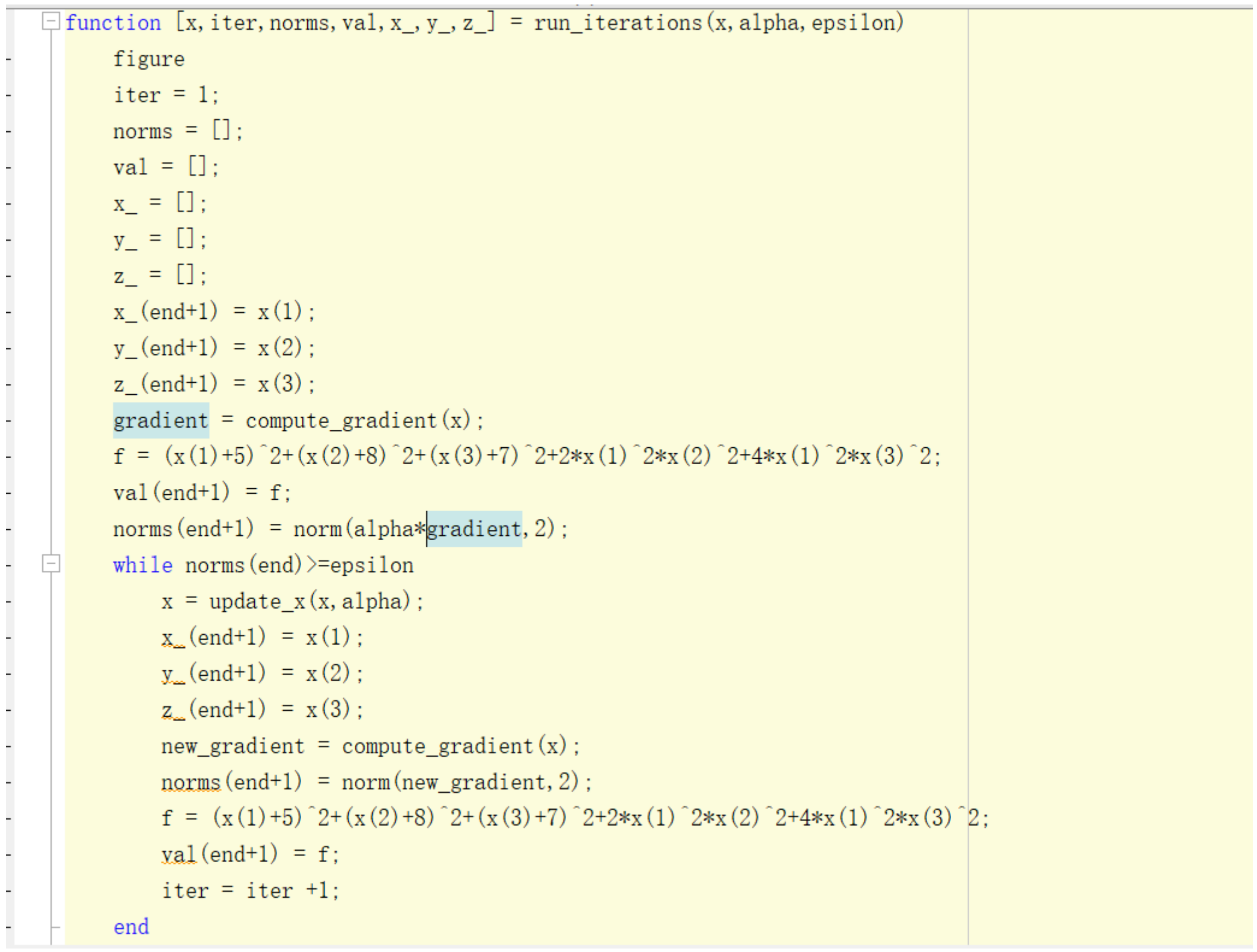
```
function new_x = update_x(x,alpha)
    g=compute_gradient(x);
    new_x = x - alpha*g;
end
```

3. Finding the $\left|\alpha * d_k\right|$



```
    norms(end+1) = norm(alpha*gradient,2);
```

4. The whole iterations to find the best x.



```
function [x,iter,norms,val,x_,y_,z_] = run_iterations(x,alpha,epsilon)
    figure
    iter = 1;
    norms = [];
    val = [];
    x_ = [];
    y_ = [];
    z_ = [];
    x_(end+1) = x(1);
    y_(end+1) = x(2);
    z_(end+1) = x(3);
    gradient = compute_gradient(x);
    f = (x(1)+5)^2+(x(2)+8)^2+(x(3)+7)^2+2*x(1)^2*x(2)^2+4*x(1)^2*x(3)^2;
    val(end+1) = f;
    norms(end+1) = norm(alpha*gradient,2);
    while norms(end)>=epsilon
        x = update_x(x,alpha);
        x_(end+1) = x(1);
        y_(end+1) = x(2);
        z_(end+1) = x(3);
        new_gradient = compute_gradient(x);
        norms(end+1) = norm(new_gradient,2);
        f = (x(1)+5)^2+(x(2)+8)^2+(x(3)+7)^2+2*x(1)^2*x(2)^2+4*x(1)^2*x(3)^2;
        val(end+1) = f;
        iter = iter +1;
    end
```

Result for x:

```
final_x =

   -0.0154   -7.9962   -6.9934
```

a) In order to evaluate the second order necessary condition, we could test if the Hessian matrix is positive semi-definite. The way to test is that obtain the eigen value of the Hessian matrix, if all the eigen values are nonnegatives. However, this method only works if the Hessian matrix is symmetrical. Thus, below are the procedure of the code and the result.

$$H = \begin{bmatrix} 4x_2^2 + 8x_3^2 & 8x_1x_2 & 16x_1x_3 \\ 8x_1\boldsymbol{x}_2 & 4x_1^2 + 2 & 0 \\ 16x_1x_3 & 0 & 8x_1^2 + 2 \end{bmatrix}$$

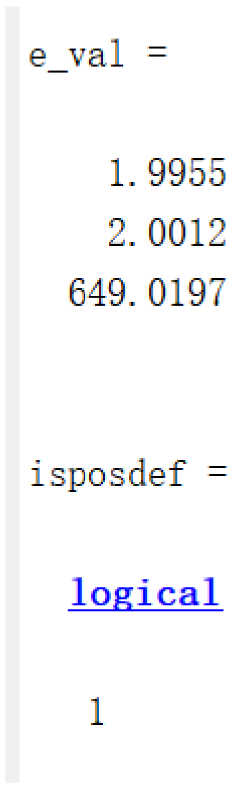
Check if H is symmetric

```
is_sym = issymmetric(H)
```

```
is_sym =

  logical

   1
```

Check if H is positive semi-definite

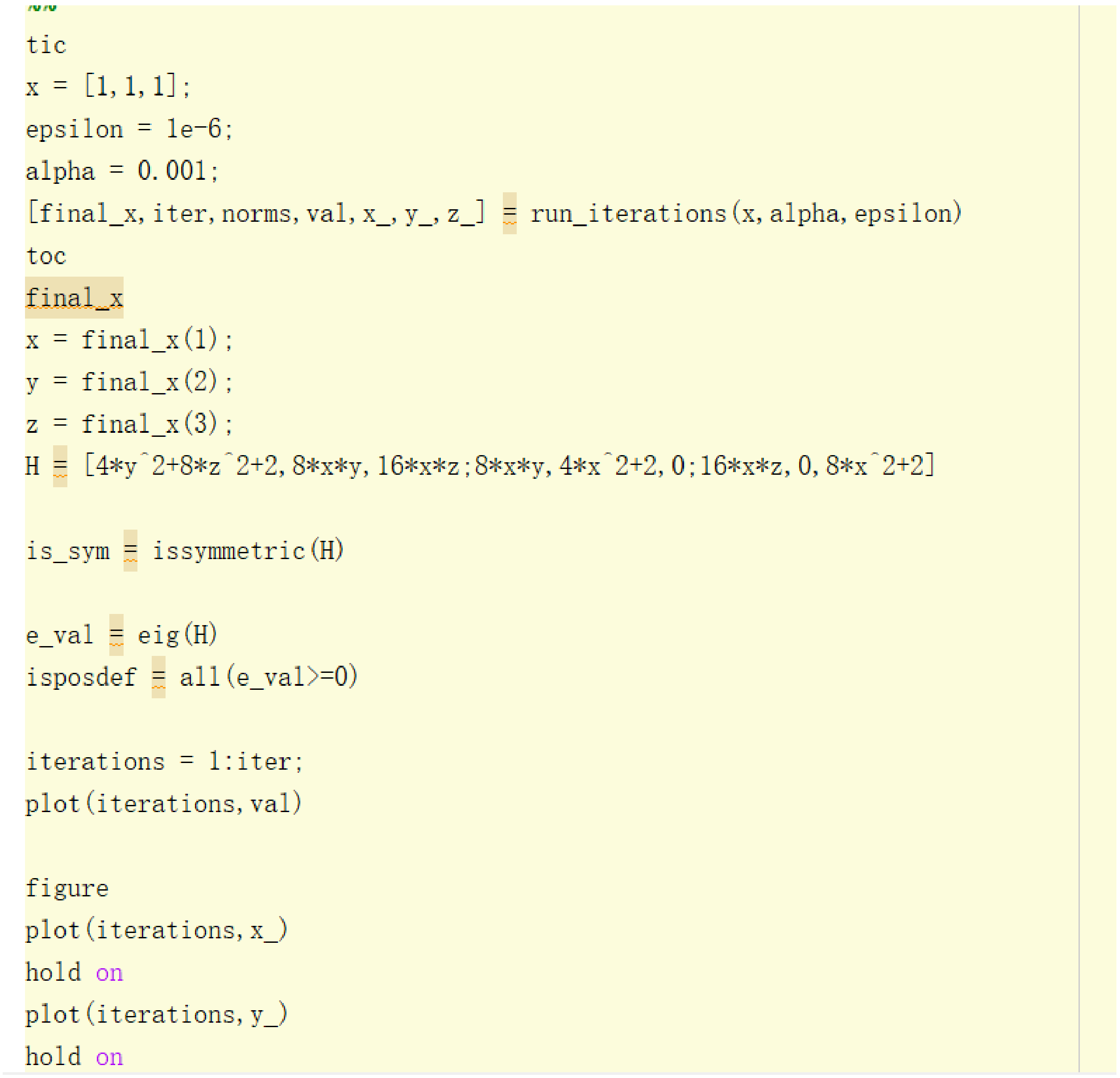
```
e_val = eig(H)
isposdef = all(e_val>=0)
```



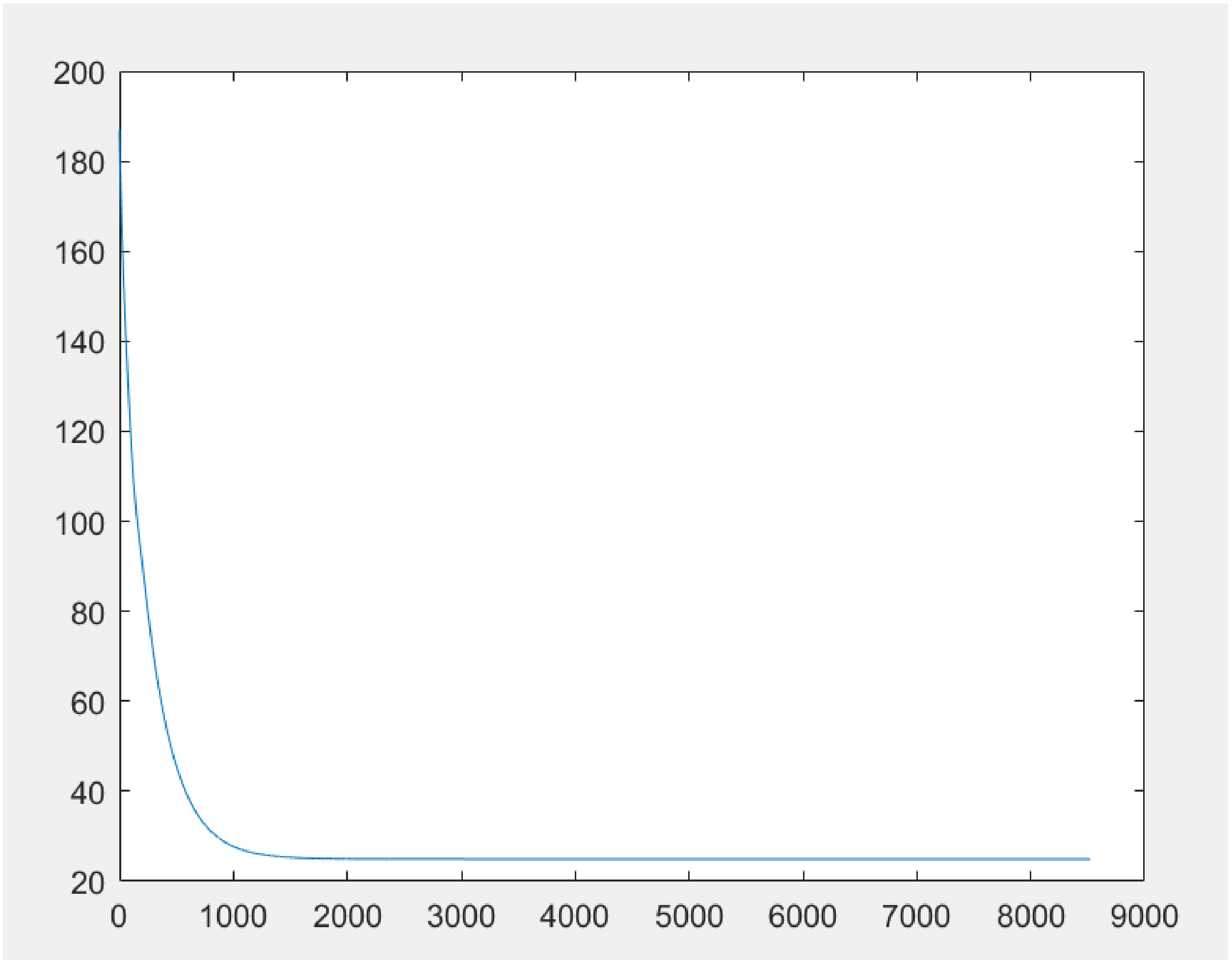
```
e_val =

    1.9955
    2.0012
  649.0197


isposdef =

  logical

   1
```

Hence the x that we find satisfies the second order necessary condition.

b) In each iteration, whenever we update x, we then calculate the objective function and store in the matrix. Below is the implementations and results:



```
%%
tic
x = [1,1,1];
epsilon = 1e-6;
alpha = 0.001;
[final_x,iter,norms,val,x_,y_,z_] = run_iterations(x,alpha,epsilon)
toc
final_x
x = final_x(1);
y = final_x(2);
z = final_x(3);
H = [4*y^2+8*z^2+2,8*x*y,16*x*z;8*x*y,4*x^2+2,0;16*x*z,0,8*x^2+2]

is_sym = issymmetric(H)

e_val = eig(H)
isposdef = all(e_val>=0)

iterations = 1:iter;
plot(iterations,val)

figure
plot(iterations,x_)
hold on
plot(iterations,y_)
hold on
```

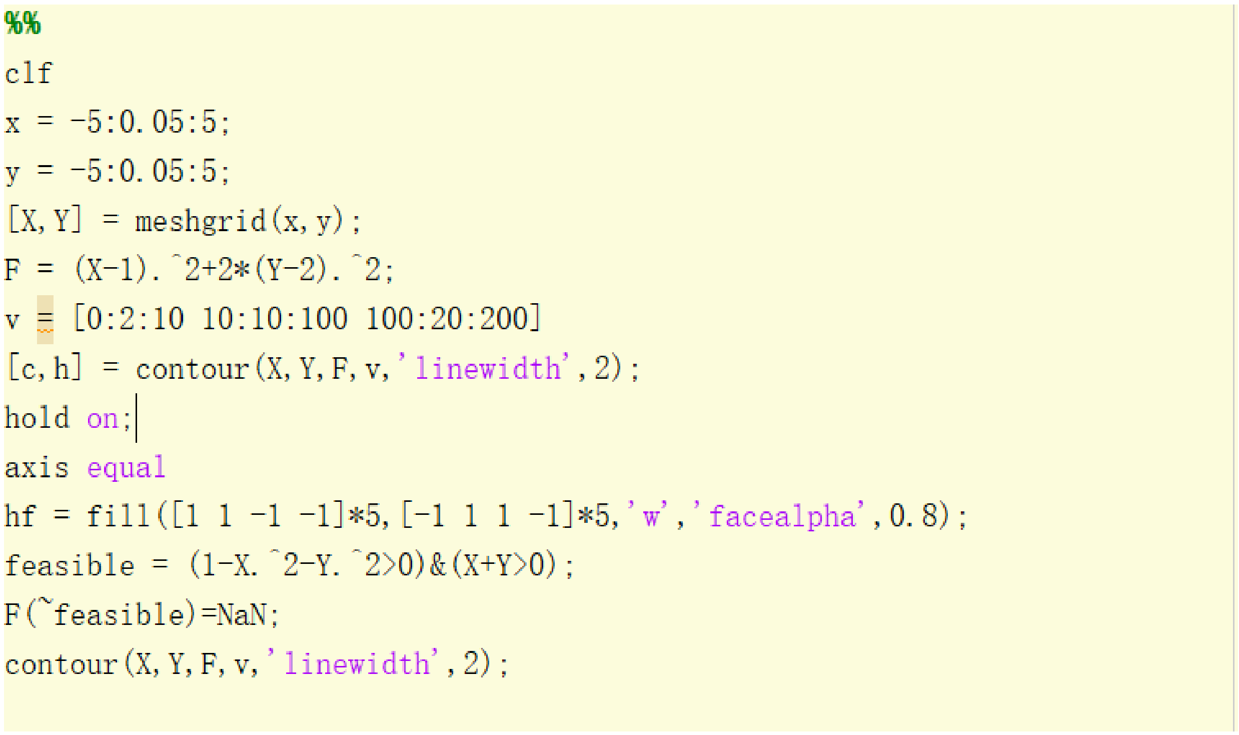


c) Rate of convergence. Another aspect of looking at the convergence rate is to look at the eigenvalues of the Hessian matrix. The minimum eigenvalue is 1.9955, whereas the largest eigenvalue is 649.0197. Thus,
$r = \frac{smallest\ eigenvalue}{largest\ eigenvalue} = 0.0031$, the convergence ratio
$\beta = (\frac{1-r}{1+r})^2 = 0.9878$. Thus, by looking at the eigenvalue difference, we can conclude that the rate of convergence is low because all the eigenvalues are distributed sparsely. I also ran a timer to count the time the program takes to finish the algorithm

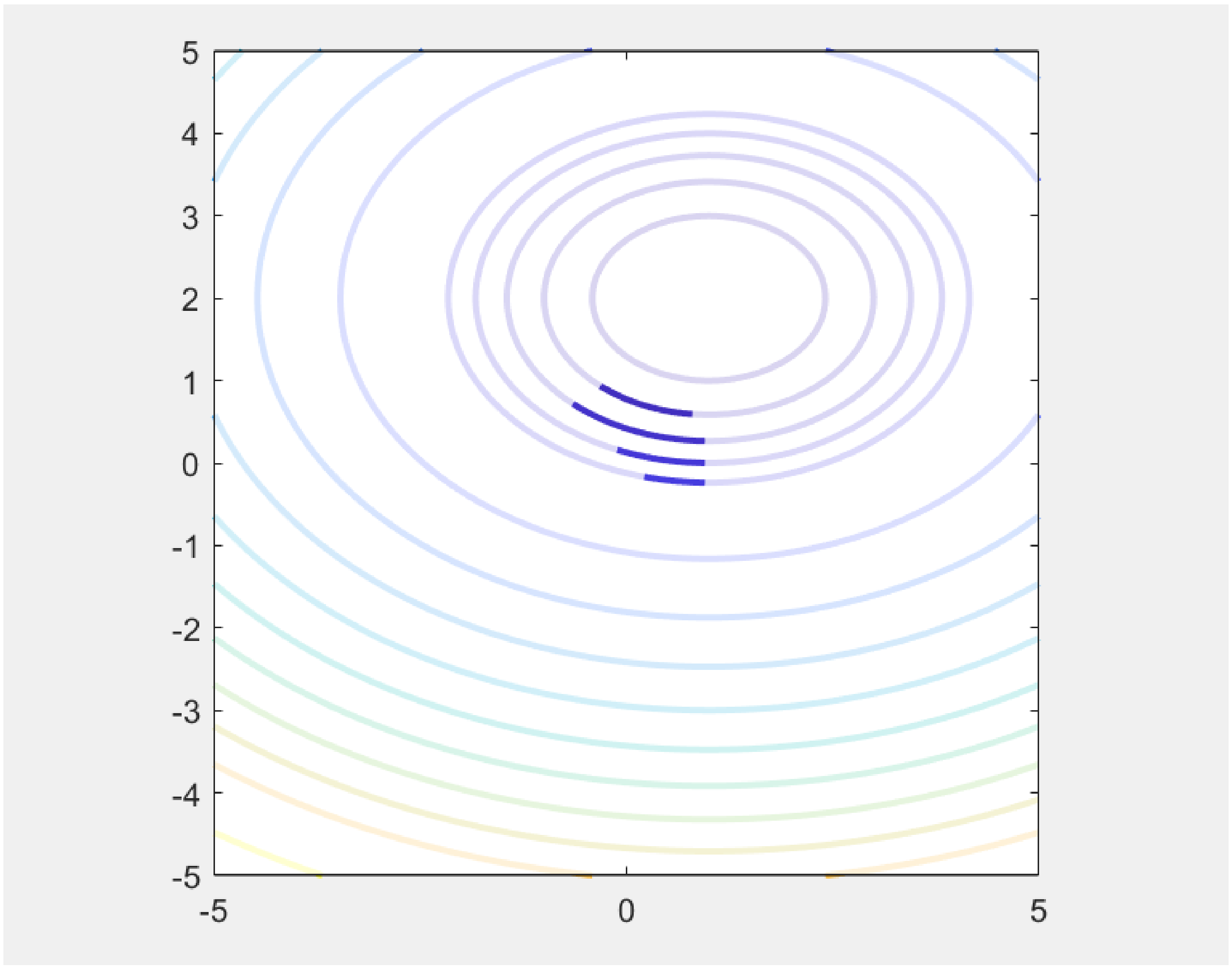
Results:
0.098815 seconds

Q2.

a)



```
%%
clf
x = -5:0.05:5;
y = -5:0.05:5;
[X,Y] = meshgrid(x,y);
F = (X-1).^2+2*(Y-2).^2;
v = [0:2:10 10:10:100 100:20:200]
[c,h] = contour(X,Y,F,v,'linewidth',2);
hold on;
axis equal
hf = fill([1 1 -1 -1]*5,[-1 1 1 -1]*5,'w','facealpha',0.8);
feasible = (1-X.^2-Y.^2>0)&(X+Y>0);
F(~feasible)=NaN;
contour(X,Y,F,v,'linewidth',2);
```



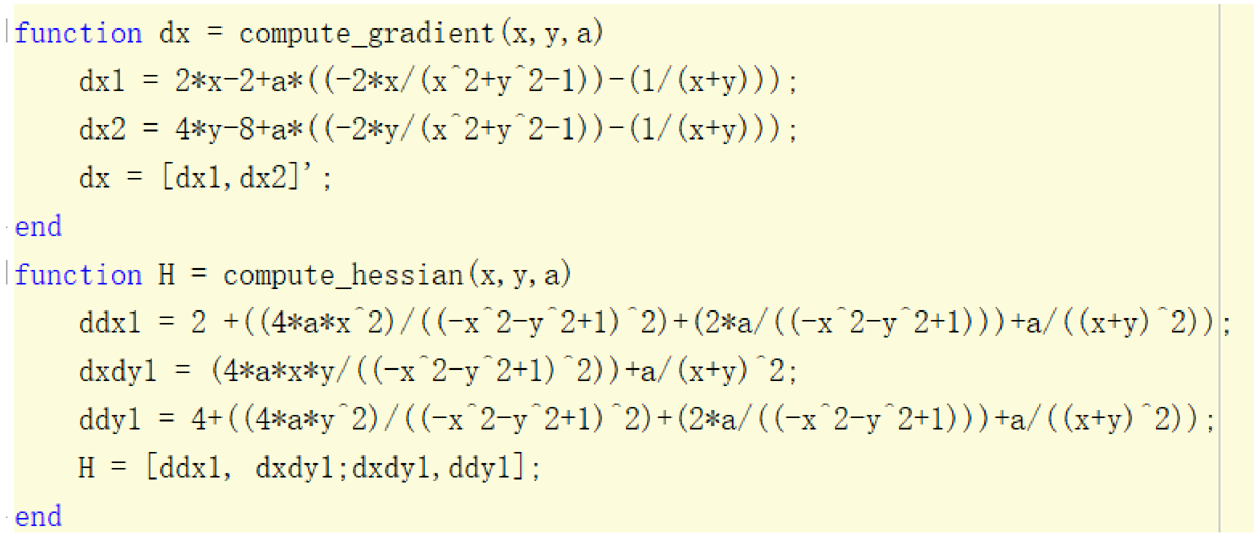
The dark blue regions are the feasible sets regarding this problem.

b) The algorithm of solving this problem is as follow:

Step 1. Choose an starting point $x_0 = [0.5\ 0.5]^T$ and and a stopping threshold $\epsilon = 0.002$. The penalty term is a = 1 as stated in the question.

```
clc
clear
a = 1;
tol = 0.002;
iter =1;
x(1) = 0.5;
y(1) = 0.5;
h1(1) = compute_h1(x(1),y(1));
h2(1) = compute_h2(x(1),y(1));
val(1) = compute_val(x(1),y(1));
```

Step 2. Compose a barrier function and compute the gradient and the hessian matrix as the following:



```
function dx = compute_gradient(x,y,a)
    dx1 = 2*x-2+a*((-2*x/(x^2+y^2-1))-(1/(x+y)));
    dx2 = 4*y-8+a*((-2*y/(x^2+y^2-1))-(1/(x+y)));
    dx = [dx1,dx2]';
end
function H = compute_hessian(x,y,a)
    ddx1 = 2 +((4*a*x^2)/((-x^2-y^2+1)^2)+(2*a/((-x^2-y^2+1)))+a/((x+y)^2));
    dxdy1 = (4*a*x*y/((-x^2-y^2+1)^2))+a/(x+y)^2;
    ddy1 = 4+((4*a*y^2)/((-x^2-y^2+1)^2)+(2*a/((-x^2-y^2+1)))+a/((x+y)^2));
    H = [ddx1, dxdy1;dxdy1,ddy1];
end
```

Step 3. Choose the search direction as $d_k = H^{-1}$

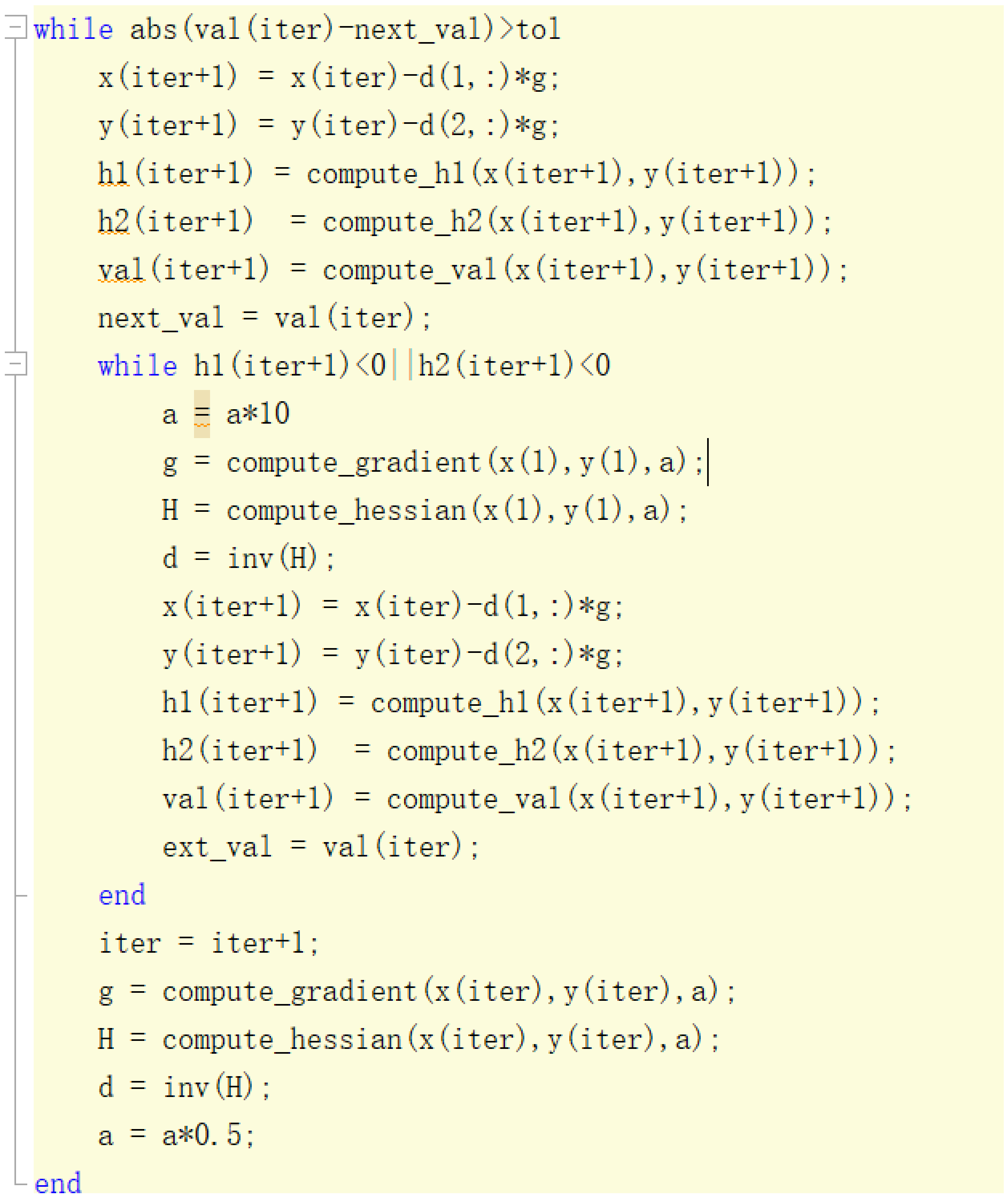
Step 4. Start the iteration by first computing the new x variables by using $x^{(k+1)} = x^{k} + H^{-1} * g_k$, where g_k is the gradient of objective function.

```
x(iter+1) = x(iter)-d(1,:)*g;
y(iter+1) = y(iter)-d(2,:)*g;
h1(iter+1) = compute_h1(x(iter+1),y(iter+1));
h2(iter+1)  = compute_h2(x(iter+1),y(iter+1));
val(iter+1) = compute_val(x(iter+1),y(iter+1));
next_val = val(iter);
```

Step 5. In the meantime check if the new x is in the feasible set. If not, then we first increase the barrier coefficient a by 10 and re-pick a new x from starting point and compute its gradient and hessian, then do update again to see if the x is in the feasible set. (iterative unless x is in the feasible set)

```
while h1(iter+1)<0||h2(iter+1)<0
    a = a*10
    g = compute_gradient(x(1),y(1),a);
    H = compute_hessian(x(1),y(1),a);
    d = inv(H);
    x(iter+1) = x(iter)-d(1,:)*g;
    y(iter+1) = y(iter)-d(2,:)*g;
    h1(iter+1) = compute_h1(x(iter+1),y(iter+1));
    h2(iter+1)  = compute_h2(x(iter+1),y(iter+1));
    val(iter+1) = compute_val(x(iter+1),y(iter+1));
    ext_val = val(iter);
end
```

Step 6. Then update gradient and hessian and stop when $\left|f(x^{k-1}) - f(x^{k})\right| < \epsilon$

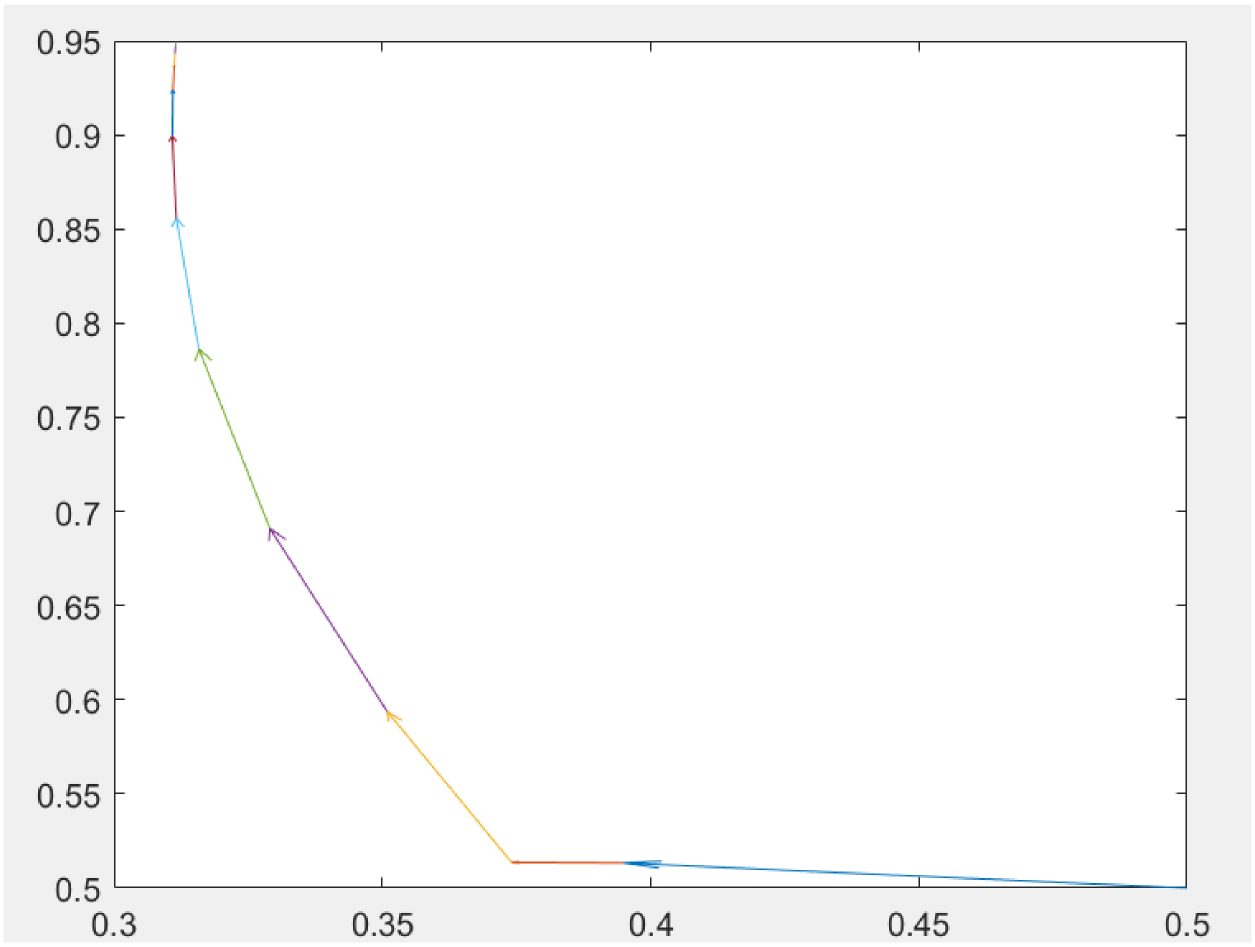


```
while abs(val(iter)-next_val)>tol
    x(iter+1) = x(iter)-d(1,:)*g;
    y(iter+1) = y(iter)-d(2,:)*g;
    h1(iter+1) = compute_h1(x(iter+1),y(iter+1));
    h2(iter+1)  = compute_h2(x(iter+1),y(iter+1));
    val(iter+1) = compute_val(x(iter+1),y(iter+1));
    next_val = val(iter);
    while h1(iter+1)<0||h2(iter+1)<0
        a = a*10
        g = compute_gradient(x(1),y(1),a);
        H = compute_hessian(x(1),y(1),a);
        d = inv(H);
        x(iter+1) = x(iter)-d(1,:)*g;
        y(iter+1) = y(iter)-d(2,:)*g;
        h1(iter+1) = compute_h1(x(iter+1),y(iter+1));
        h2(iter+1)  = compute_h2(x(iter+1),y(iter+1));
        val(iter+1) = compute_val(x(iter+1),y(iter+1));
        ext_val = val(iter);
    end
    iter = iter+1;
    g = compute_gradient(x(iter),y(iter),a);
    H = compute_hessian(x(iter),y(iter),a);
    d = inv(H);
    a = a*0.5;
end
```

```
X =

    0.3115    0.9497


H1 =

   9.1738e-04


H2 =

    1.2613


Val =

    2.6800
```

Result:

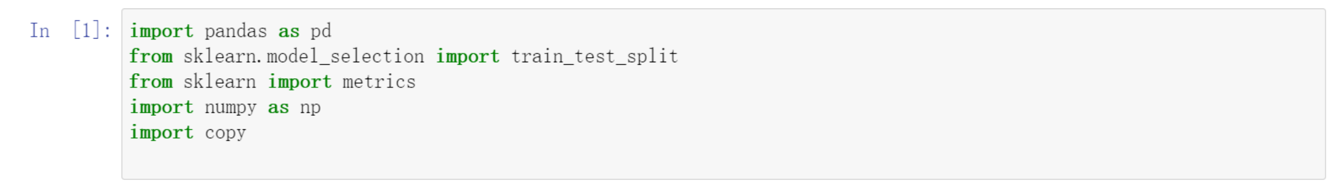
c)


0.95
0.9
0.85
0.8
0.75
0.7
0.65
0.6
0.55
0.5
0.3
0.35
0.4
0.45
0.5

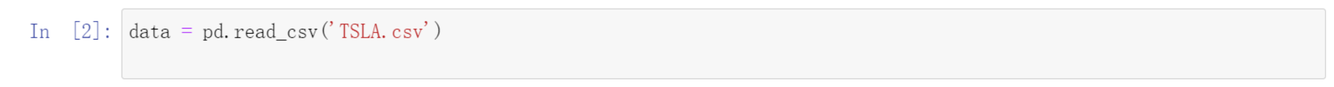
Q3

a)

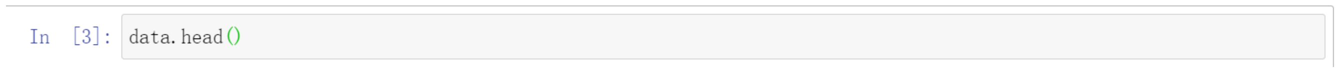
Step 1. Load Data from .csv file



```
In [1]: import pandas as pd
        from sklearn.model_selection import train_test_split
        from sklearn import metrics
        import numpy as np
        import copy
```

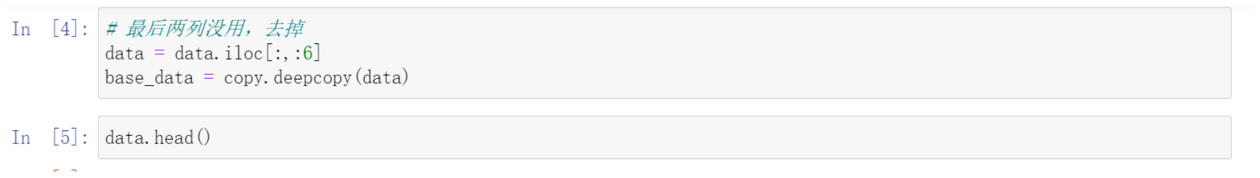


```
In [2]: data = pd.read_csv('TSLA.csv')
```



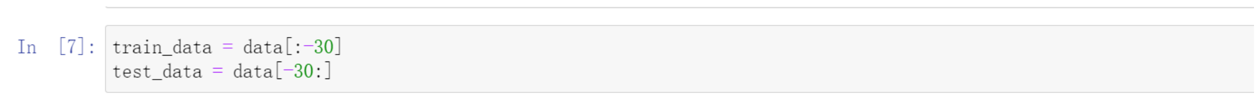
```
In [3]: data.head()
```

Step 2. Pre-processing the data (delete columns that are not useful)



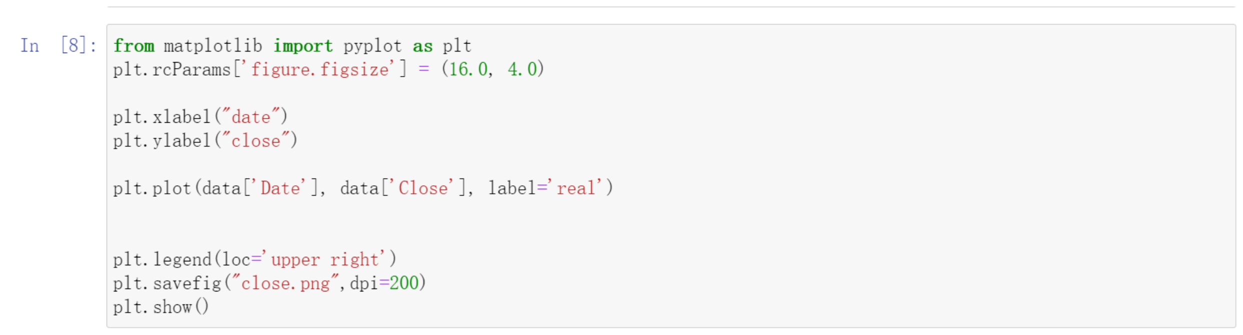
```
In [4]: # 最后两列没用，去掉
        data = data.iloc[:,:6]
        base_data = copy.deepcopy(data)

In [5]: data.head()
```

Step 3. Select the past 30 days date from the “Date” column



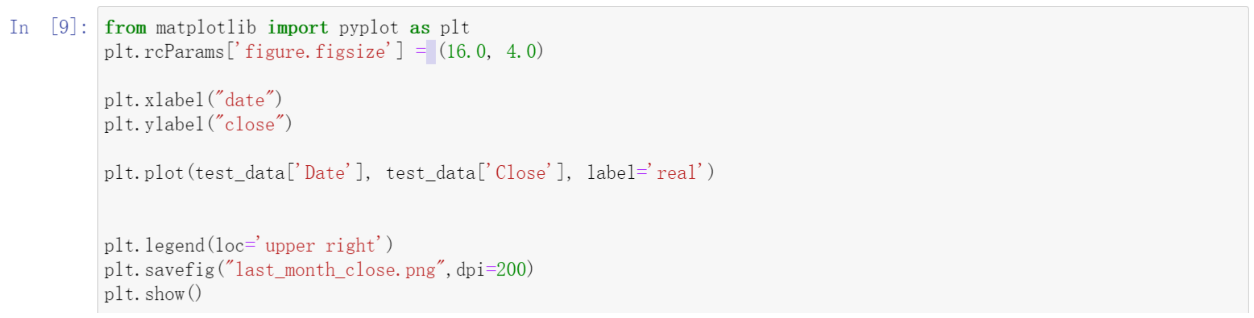
```
In [7]: train_data = data[:-30]
        test_data = data[-30:]
```

Step 4. Select “Date” column of data and “Close” value for the stock data and plot them out for the whole dataset.



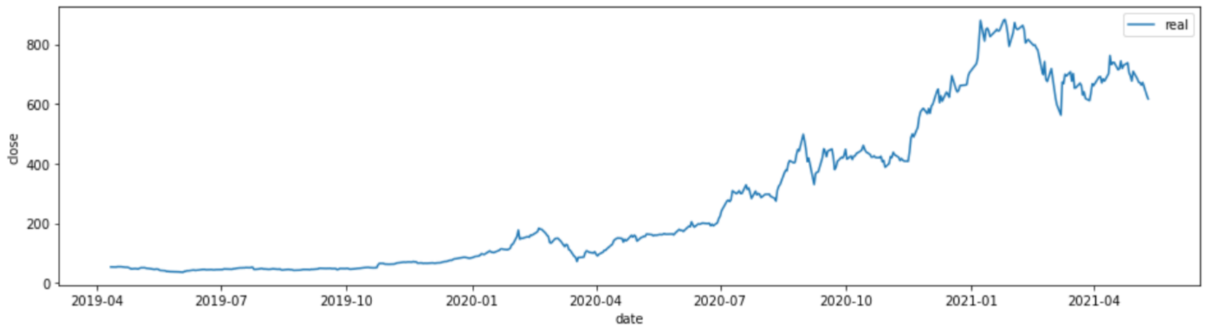
```
In [8]: from matplotlib import pyplot as plt
        plt.rcParams['figure.figsize'] = (16.0, 4.0)

        plt.xlabel("date")
        plt.ylabel("close")

        plt.plot(data['Date'], data['Close'], label='real')


        plt.legend(loc='upper right')
        plt.savefig("close.png",dpi=200)
        plt.show()
```

Step 5. Select the “Date” column of data and “Close” value for the stock data and plot them out for the past 30 days.

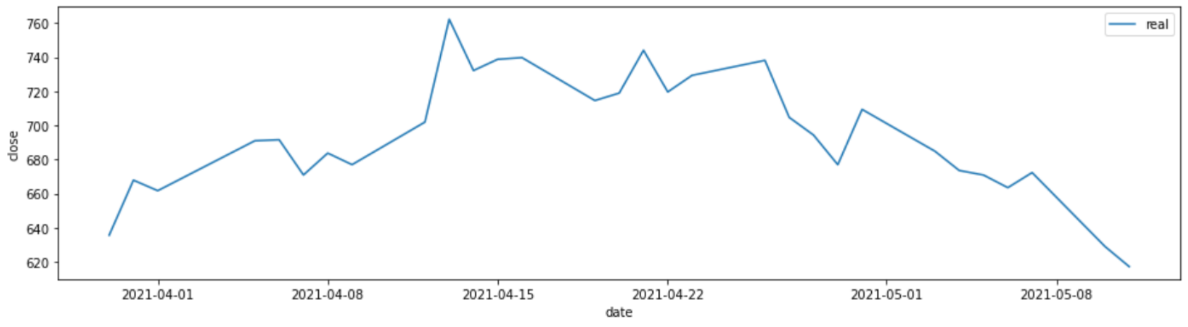


```
In  [9]: from matplotlib import pyplot as plt
         plt.rcParams['figure.figsize'] = (16.0, 4.0)

         plt.xlabel("date")
         plt.ylabel("close")

         plt.plot(test_data['Date'], test_data['Close'], label='real')


         plt.legend(loc='upper right')
         plt.savefig("last_month_close.png",dpi=200)
         plt.show()
```

Results:



This is for a 3 years period.



This is the last 30 days.

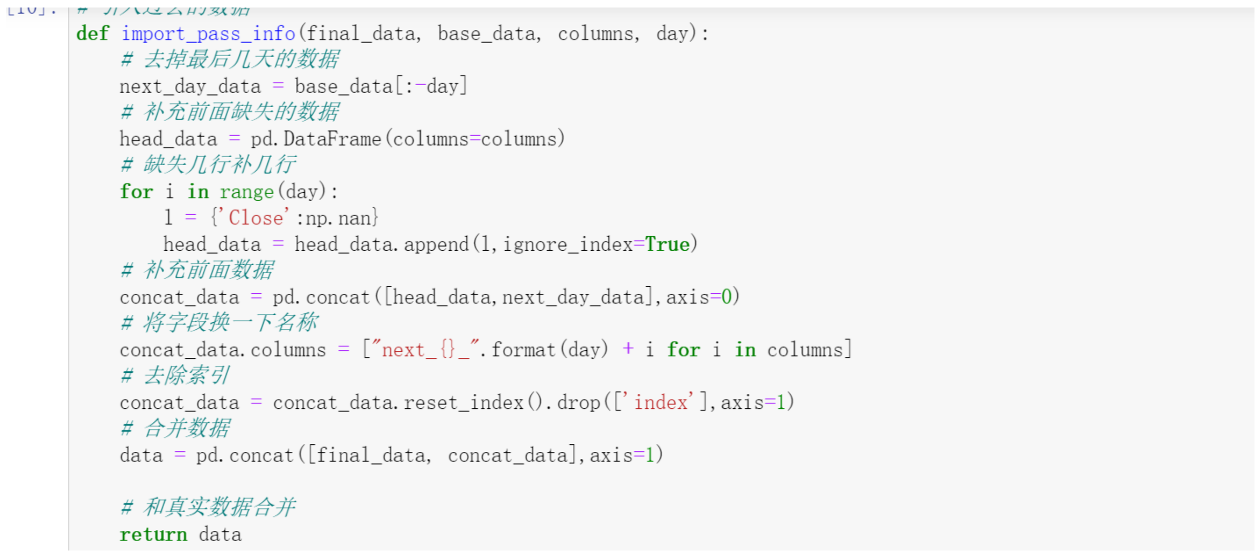
b)
Before using the stochastic gradient descent, there are two ways to approach this problem. The first is that from the dataset, we know the attributes ['Date',

'Open', 'High', 'Low', 'Close', 'Volume'], the one we want to predict is the ‘Close’ attribute. Thus, we could build a model using ‘Open’, ‘High’, ‘Low’, ‘Volume’ as the inputs x and the corresponding ‘Close’ attribute as the target. The inputs x will result in a predicted ‘Close’ value, which then compares with the target value and sets up a Mean Square Error (MSE) as the optimization model in order to tune the parameters of the model. Another way to do this problem is to only look at the ‘Close’ attributes, and use the past close values to predict today's close value and compare with the actual today’s close value.

Both of the methods use past 5 days data as training and testing sets. The only difference between these two approaches is the data preprocessing.

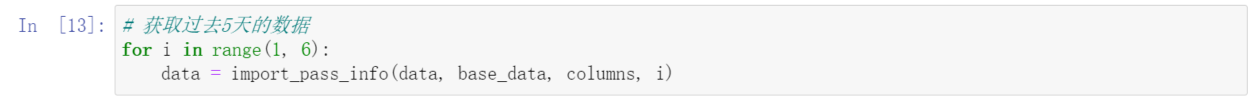
<u>**Train and test with Open, High, Low, Volume attributes and predict close:**</u>

Step 1. Assemble the attribute so that the next day will have the previous day's attribute values.

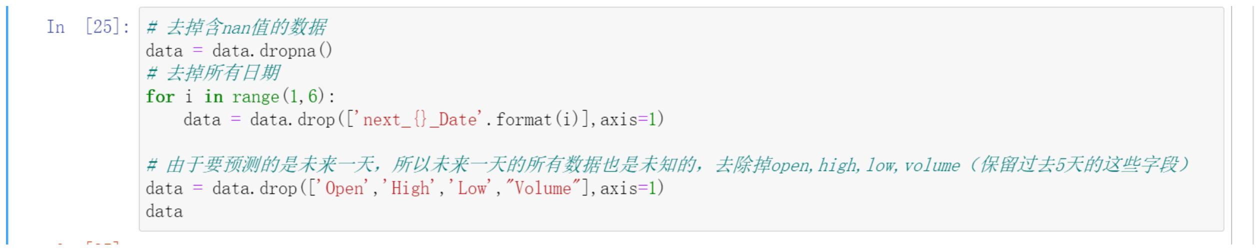


```
[10]: # 引入过去的数据
def import_pass_info(final_data, base_data, columns, day):
    # 去掉最后几天的数据
    next_day_data = base_data[:-day]
    # 补充前面缺失的数据
    head_data = pd.DataFrame(columns=columns)
    # 缺失几行补几行
    for i in range(day):
        l = {'Close':np.nan}
        head_data = head_data.append(l,ignore_index=True)
    # 补充前面数据
    concat_data = pd.concat([head_data,next_day_data],axis=0)
    # 将字段换一下名称
    concat_data.columns = ["next_{}_".format(day) + i for i in columns]
    # 去除索引
    concat_data = concat_data.reset_index().drop(['index'],axis=1)
    # 合并数据
    data = pd.concat([final_data, concat_data],axis=1)

    # 和真实数据合并
    return data
```

Step 2. Gather last 5 days data.

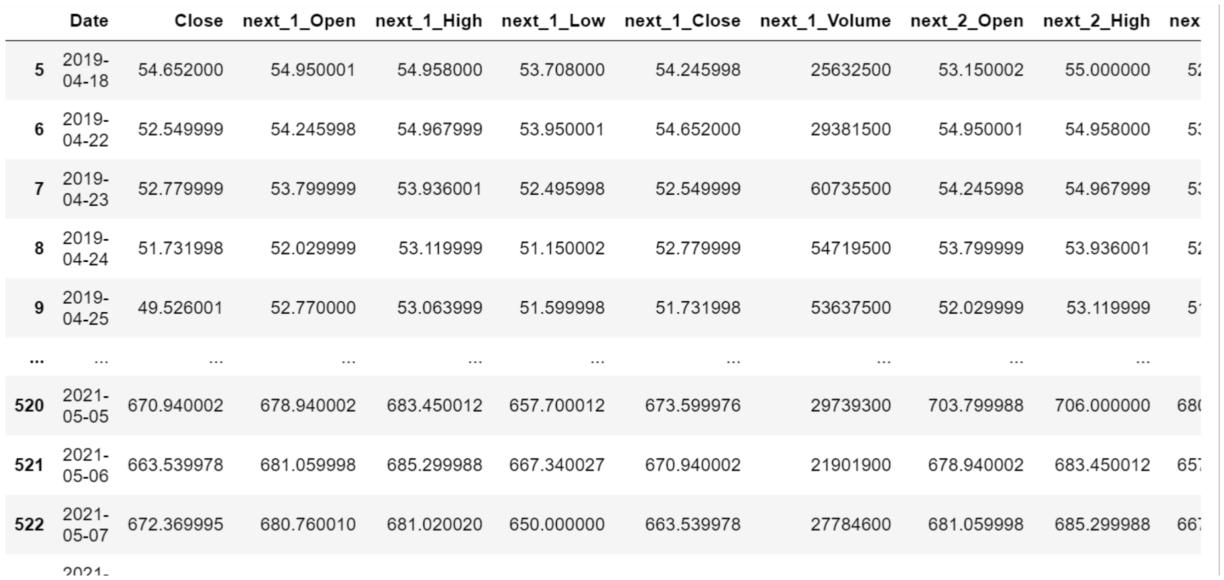


```
In [13]: # 获取过去5天的数据
         for i in range(1, 6):
             data = import_pass_info(data, base_data, columns, i)
```



```
In [25]: # 去掉含nan值的数据
         data = data.dropna()
         # 去掉所有日期
         for i in range(1,6):
             data = data.drop(['next_{}_Date'.format(i)],axis=1)

         # 由于要预测的是未来一天，所以未来一天的所有数据也是未知的，去除掉open,high,low,volume（保留过去5天的这些字段）
         data = data.drop(['Open','High','Low',"Volume"],axis=1)
         data
```

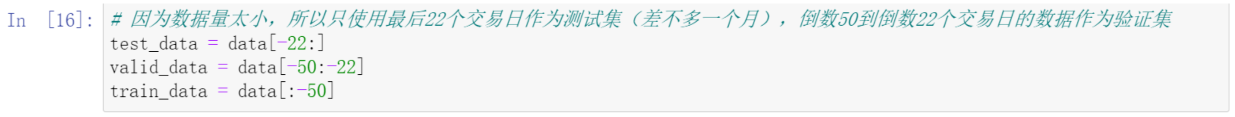
The dataset now looks like this:



	Date	Close	next_1_Open	next_1_High	next_1_Low	next_1_Close	next_1_Volume	next_2_Open	next_2_High	nex
5	2019-04-18	54.652000	54.950001	54.958000	53.708000	54.245998	25632500	53.150002	55.000000	52
6	2019-04-22	52.549999	54.245998	54.967999	53.950001	54.652000	29381500	54.950001	54.958000	53
7	2019-04-23	52.779999	53.799999	53.936001	52.495998	52.549999	60735500	54.245998	54.967999	53
8	2019-04-24	51.731998	52.029999	53.119999	51.150002	52.779999	54719500	53.799999	53.936001	52
9	2019-04-25	49.526001	52.770000	53.063999	51.599998	51.731998	53637500	52.029999	53.119999	51
...	...	...	...	...	...	...	...	...	...	
520	2021-05-05	670.940002	678.940002	683.450012	657.700012	673.599976	29739300	703.799988	706.000000	680
521	2021-05-06	663.539978	681.059998	685.299988	667.340027	670.940002	21901900	678.940002	683.450012	65
522	2021-05-07	672.369995	680.760010	681.020020	650.000000	663.539978	27784600	681.059998	685.299988	66
	2021-									

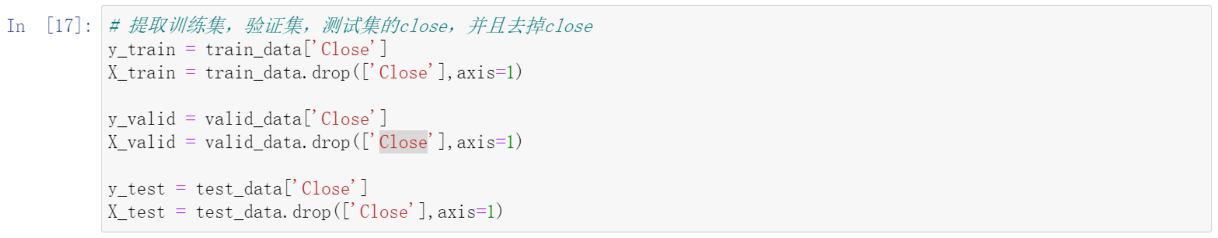
Now we can see that the dataset has the current close value as the target follow by the previous 5 days' attributes values.

Step 3. Split this data set into train, test and validation



```
In  [16]: # 因为数据量太小，所以只使用最后22个交易日作为测试集（差不多一个月），倒数50到倒数22个交易日的数据作为验证集
          test_data = data[-22:]
          valid_data = data[-50:-22]
          train_data = data[:-50]
```

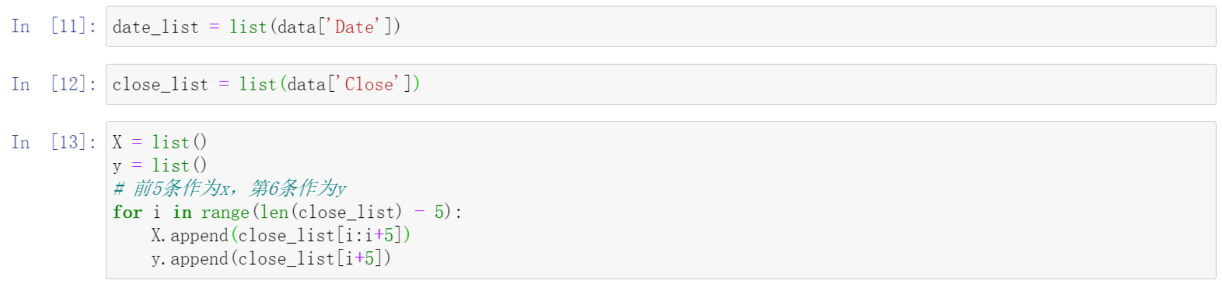
Step 4. Set Close attribute as target label. Drop Close attribute in both train , test and validation sets.



```
In  [17]: # 提取训练集，验证集，测试集的close，并且去掉close
          y_train = train_data['Close']
          X_train = train_data.drop(['Close'],axis=1)

          y_valid = valid_data['Close']
          X_valid = valid_data.drop(['Close'],axis=1)

          y_test = test_data['Close']
          X_test = test_data.drop(['Close'],axis=1)
```

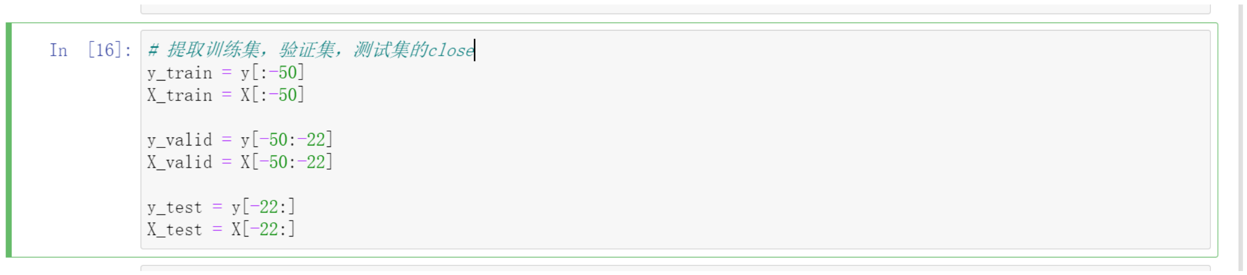
<u>**Train and test with only close attribute and predict close:**</u>

Step 1. Pick only Close attributes from the whole dataset and use the first 5 as inputs and the 6 close as the target.



```
In  [11]: date_list = list(data['Date'])

In  [12]: close_list = list(data['Close'])

In  [13]: X = list()
          y = list()
          # 前5条作为x，第6条作为y
          for i in range(len(close_list) - 5):
              X.append(close_list[i:i+5])
              y.append(close_list[i+5])
```

Step 2. Split this data set into train, test and validation



```
In  [16]: # 提取训练集，验证集，测试集的close
          y_train = y[:-50]
          X_train = X[:-50]

          y_valid = y[-50:-22]
          X_valid = X[-50:-22]

          y_test = y[-22:]
          X_test = X[-22:]
```

Now we have obtained each of the training sets and testing sets. We can apply to Stochastic Gradient Descent on the linear regression model.

Similar to Steepest gradient descent procedure, the update of x is $x^{(k+1)} = x^{k} + \alpha * d_k$, where the $d_k = -\frac{df}{dx}$. However, the question is what is the objective function now?

We define the objective function as the following:

$$f(\theta) = \frac{1}{n}\sum_{i=1}^{n} Loss(y^{(i)} - \hat{y}^{(i)})$$

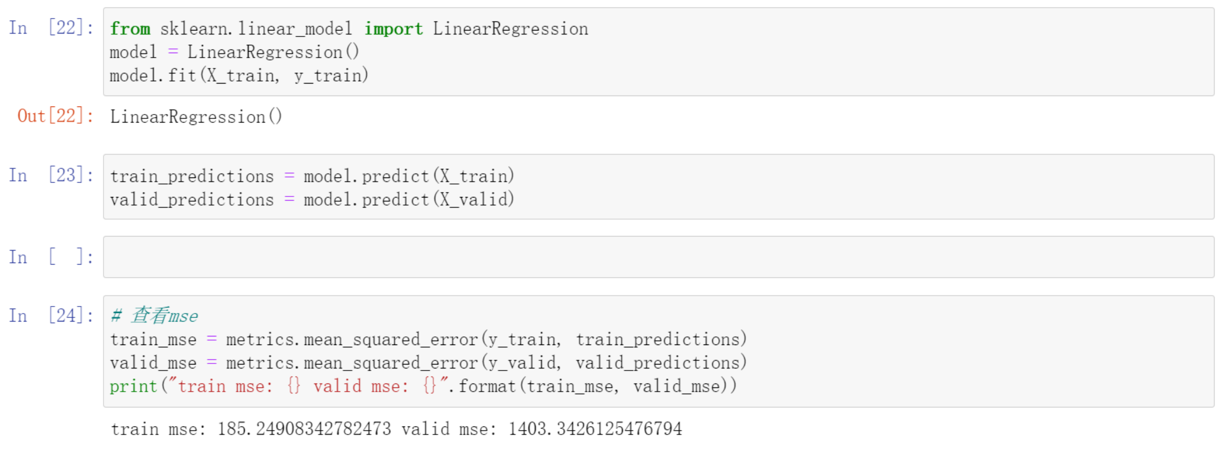
$$Where\ Loss(y^{(i)} - \hat{y}^{(i)}) = \frac{1}{2}(y^{(i)} - \hat{y}^{(i)})^2$$

We define $y^{(i)} - \hat{y}^{(i)}$ as the difference between the actual target and the predicted target of the closing price.

Below are the implementations:

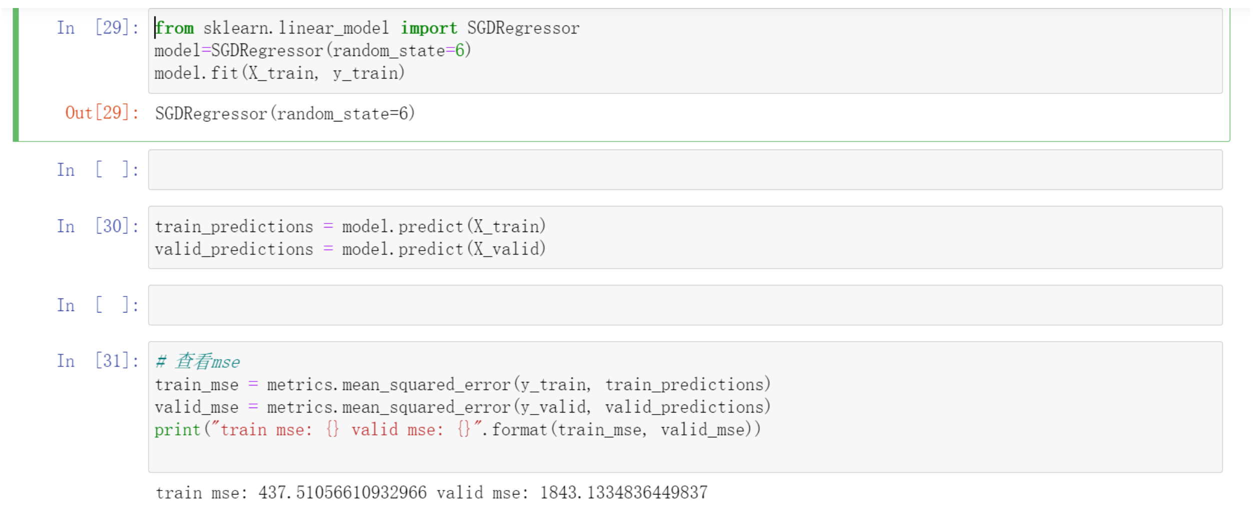
<u>**Linear Regression Model:**</u>

linear regression



```
In [22]: from sklearn.linear_model import LinearRegression
         model = LinearRegression()
         model.fit(X_train, y_train)

Out[22]: LinearRegression()

In [23]: train_predictions = model.predict(X_train)
         valid_predictions = model.predict(X_valid)

In [ ]:

In [24]: # 查看mse
         train_mse = metrics.mean_squared_error(y_train, train_predictions)
         valid_mse = metrics.mean_squared_error(y_valid, valid_predictions)
         print("train mse: {} valid mse: {}".format(train_mse, valid_mse))

         train mse: 185.24908342782473 valid mse: 1403.3426125476794
```

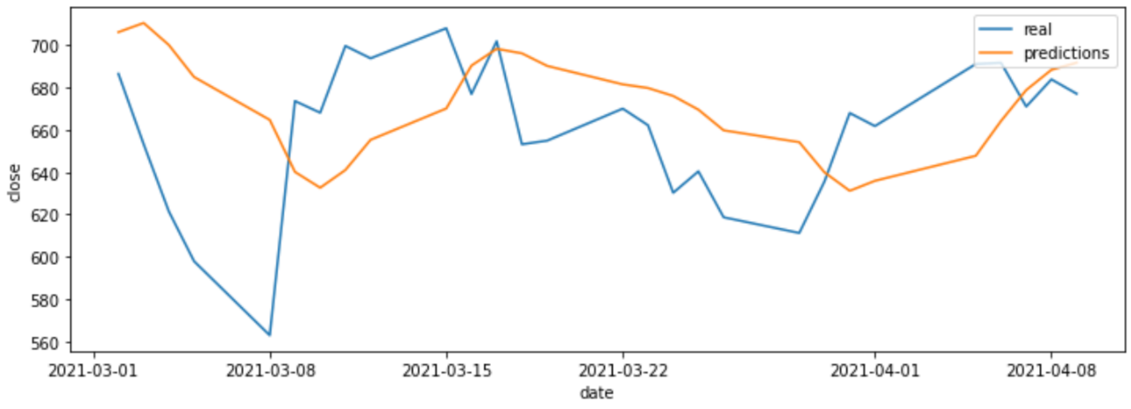
Stochastic Gradient Descent on Linear Regression Model:



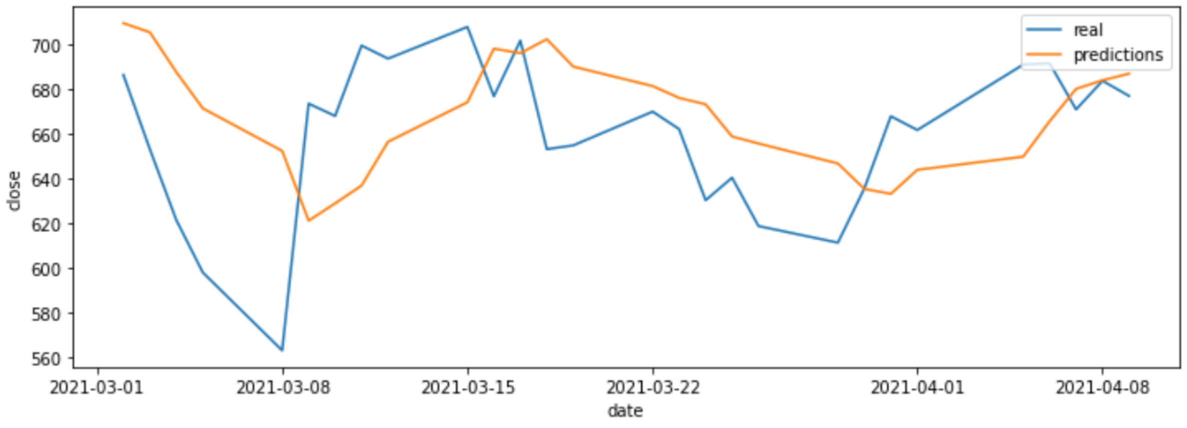
```
In [29]: from sklearn.linear_model import SGDRegressor
         model=SGDRegressor(random_state=6)
         model.fit(X_train, y_train)

Out[29]: SGDRegressor(random_state=6)

In [ ]:

In [30]: train_predictions = model.predict(X_train)
         valid_predictions = model.predict(X_valid)

In [ ]:

In [31]: # 查看mse
         train_mse = metrics.mean_squared_error(y_train, train_predictions)
         valid_mse = metrics.mean_squared_error(y_valid, valid_predictions)
         print("train mse: {} valid mse: {}".format(train_mse, valid_mse))

         train mse: 437.51056610932966 valid mse: 1843.1334836449837
```

c)

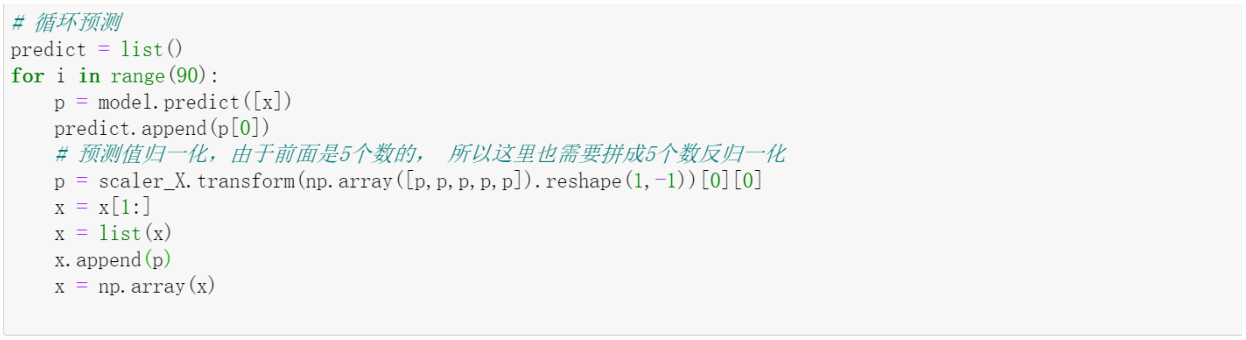
Plots that use train and test sets based on Open, High, Low, Volume attributes and predict close:



<u>Plots that use train and test sets based on only close attributes and predict close:</u>

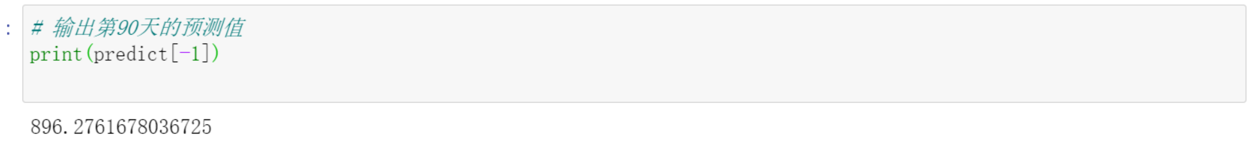


d)
In order to simplify the process of predicting the close stock price after 90 days, I did not use the first model to make a prediction. This is because the first model needs more than one attribute as inputs; it is impossible to obtain this data after 90 days. I will have to build individual models to estimate individual attributes of inputs after 90 days, and then predict the closing price after 90 days. However, the error will be significantly high since each prediction with the inputs will cause an error and the error will stack up as it goes. Thus, I use the second model and use only the previous closing price to predict today's closing price. Then iterate this method 90 times to obtain the result.



```
# 循环预测
predict = list()
for i in range(90):
    p = model.predict([x])
    predict.append(p[0])
    # 预测值归一化，由于前面是5个数的， 所以这里也需要拼成5个数反归一化
    p = scaler_X.transform(np.array([p,p,p,p,p]).reshape(1,-1))[0][0]
    x = x[1:]
    x = list(x)
    x.append(p)
    x = np.array(x)
```

The result is:



```
: # 输出第90天的预测值
  print(predict[-1])

  896.2761678036725
```

$ 896.28 after 90 days.

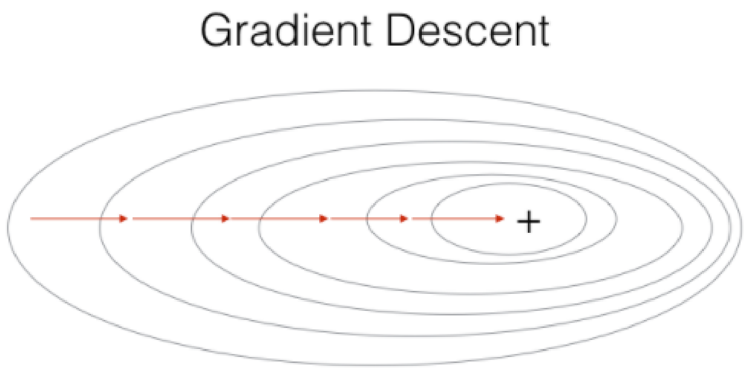
Discussion:

In general, this model does not consider any other realistic factors. The model is only based on the closing price; therefore, the prediction will not be as accurate as the actual value.

e)

<u>Pros of Batch Gradient Descent:</u>

- Uses all the sample sets to update gradient and weights. This guarantees the algorithm will find the global optimizations



- Figure shows above indicates that the search direction of Batch Gradient Descent is always smooth and points to the same direction every time

<u>Cons of Batch Gradient Descent:</u>

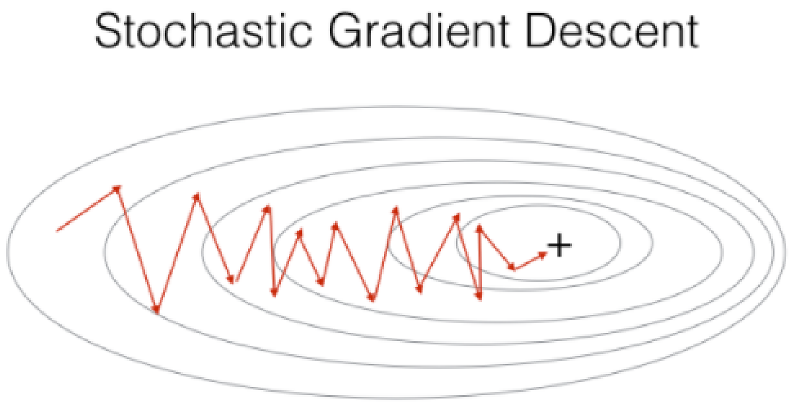
- If the dataset is large, then there will be no space to put all the datasets in the memory.
- The running time will grow as the dataset size increases.
- With small learning rate and not enough iterations, the Batch Gradient Descent will be trapped in the local minima without finding the most optimal solutions.

<u>Pros of Stochastic Gradient Descent:</u>

- At each iteration, choose random samples to update gradients and weights.
- Faster Converging rate than Batch Gradient Descent.
- Require less memory spaces to store the datasets.
- Since SGD may start at a new place each time, it is more likely to jump out of the local minima.

Cons of Stochastic Gradient Descent:

- Since SGD chooses random sets at each iteration, then it is likely to choose the same sets everytime and optimize only on one set, which is not enough to optimize the objective function. Thus, the shuffle function needs to be implemented at the start of each iteration.



- Since SGD starts with different sample sets, the direction at each iteration will also be different. This will cause the problem that when the SGD algorithm gets closer to the solutions, it will be more likely to jump back and forth or even miss the most optimal solutions. Therefore, in order to overcome this issue. A dynamic learning rate is considered. At the start of the process the learning rate could be large, so that the step of SGD will converge faster to the solution. However, when it approaches solutions, the learning rate will have to be small in order to reduce the oscillation and get the correct solutions.