Problem 1

a. Write a Python function to obtain the update inverse matrix $\boldsymbol{H_n^{-1}}$.

We are given

$$\boldsymbol{H_n^{-1}} = \frac{1}{\gamma_n}\begin{bmatrix} \boldsymbol{\gamma_n H_{n-1}^{-1} + a_n a_n'} & \boldsymbol{-a_n} \\ \boldsymbol{-a_n'} & \boldsymbol{1} \end{bmatrix}.$$

Therefore, the key points here to obtain $\boldsymbol{H_n^{-1}}$ are computing $\boldsymbol{\gamma_n}$ and $\boldsymbol{a_n}$ in an iterative way. The main procedures are : (1) read the data file and extract the target independent variables (x1~x4); (2) define the Gaussian kernel k (xi, xj, sigma) with three inputs (i-th and j-th observations, the width of the Gaussian kernel); (3) define the function Hn_inv (hn_1_inv, data, xn) to update inverse matrix $\boldsymbol{H_n^{-1}}$, where the three inputs of this function denote $\boldsymbol{H_{n-1}^{-1}}$, (n-1) observations and the n-th new observation.

I firstly import some packages we may require as below.

```
import math
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import statsmodels.api as sm
import scipy.stats as ss
from scipy.stats import norm
from numpy.linalg import norm
from numpy.linalg import inv
import random
import time
import csv
```

step (1) reads the data file by pandas and extract the target independent variables (x1~x4) kept in 'datax'.

```
data = pd.read_csv("charlie.csv")
datax=data[['x1','x2','x3','x4']]      #only contain x1,x2,x3,x4
```

step (2) defines the Gaussian kernel k (xi, xj, sigma) with three inputs (i-th and j-th observations, the width of the Gaussian kernel) and an return value named 'kij'.

```
def k(xi,xj,sigma): #xi,xj,sigma are three inputs
    kij=np.exp(-np.power(norm(xi-xj),2)/np.power(sigma,2))
    return(kij)
```

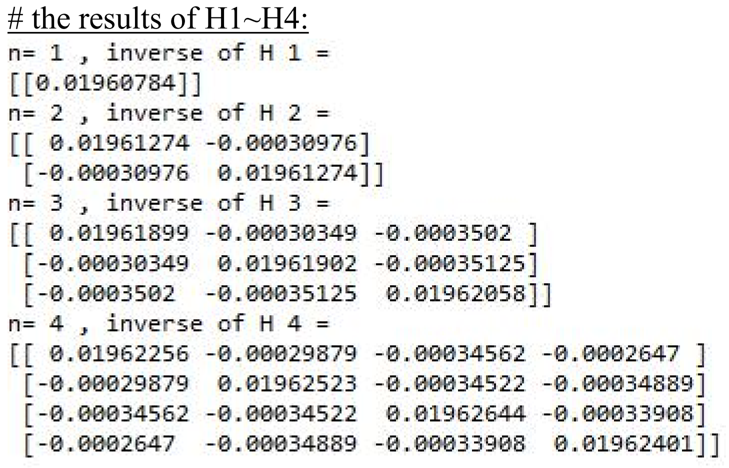
step (3) defines the function Hn_inv (hn_1_inv, data, xn) to update inverse matrix $\boldsymbol{H_n^{-1}}$, where the three inputs of this function denote $\boldsymbol{H_{n-1}^{-1}}$, (n-1) observations and the n-th new observation. Note that $\boldsymbol{\gamma_n}$ and $\boldsymbol{a_n}$ are named as 'rn' and 'an'; $\boldsymbol{\delta_n}$ and $\boldsymbol{\delta_{nn}}$ are shown as 'delta_n' and 'delta_nn'. The return value in function Hn_inv is $\boldsymbol{H_n^{-1}}$ shown as 'hn_inv'.

```
def Hn_inv(hn_1_inv,data,xn): # three inputs
    d=len(data)               # the number of (n-1) observations
    delta_n=np.zeros((d,1))   # column vector delta_n={k(xn,xi)}
    for i in range(d):
        delta_n[i][0]=k(xn,data[i,:],sigma)
    delta_nn=k(xn,xn,sigma)    # the scalar delta_nn
    an=np.array(np.matrix(hn_1_inv)*np.matrix(delta_n))   # (n-1)*1
    rn=delta_nn+(1/(2*C))np.array(np.matrix(delta_n.T)*np.matrix(an))
    hn_inv=(1/rn)*np.block([[rn*hn_1_inv+np.array(np.matrix(an)*np.matrix(an.T)),
                             -an],[-an.T,1]])
    return(hn_inv)
```

Eventually, we can test Hn_inv function and start from the inverse of H1 named as ‘h1_inv’, then use a loop to compute $\boldsymbol{H_n^{-1}}$. Here we only display the part of results from n=1 to n=4.

```
C=0.01      # the penalty factor
sigma=10    # the width of Gaussian Kernel
h1=np.matrix(k(np.array(datax)[0],np.array(datax)[0],sigma)+1/(2*C))
h1_inv=np.array(inv(h1)) # the inverse of H1
hn_1_inv=h1_inv     # the inverse of Hn-1 starts from the inverse of H1
n=len(datax)        # the number of observations
for i in range(n): #a loop for computing the inverse of Hn
    print("n=",(i+1),",","inverse","of","H",(i+1),"=")
    print(hn_1_inv)
    hn_inv=Hn_inv(hn_1_inv,np.array(datax)[0:i+1],np.array(datax)[i+1])
    hn_1_inv=hn_inv
```

The part of results of $\boldsymbol{H_n^{-1}}$, from n=1 to n=4 are attached as below.


the results of H1~H4:

```
n= 1 , inverse of H 1 =
[[0.01960784]]
n= 2 , inverse of H 2 =
[[ 0.01961274 -0.00030976]
 [-0.00030976  0.01961274]]
n= 3 , inverse of H 3 =
[[ 0.01961899 -0.00030349 -0.0003502 ]
 [-0.00030349  0.01961902 -0.00035125]
 [-0.0003502  -0.00035125  0.01962058]]
n= 4 , inverse of H 4 =
[[ 0.01962256 -0.00029879 -0.00034562 -0.0002647 ]
 [-0.00029879  0.01962523 -0.00034522 -0.00034889]
 [-0.00034562 -0.00034522  0.01962644 -0.00033908]
 [-0.0002647  -0.00034889 -0.00033908  0.01962401]]
```

b. Write a Python function to obtain the update the solution α_n.

We define the function alpha1 (hn_inv, data) with two inputs ($\boldsymbol{H_n^{-1}}$ and n observations) and a return a value (the $\boldsymbol{\alpha_n}$ vector shown as ‘alpha’). Note that ‘kj’ means the $\boldsymbol{k_n}$ vector, ‘e’ denotes $\boldsymbol{e_n}$ vector.

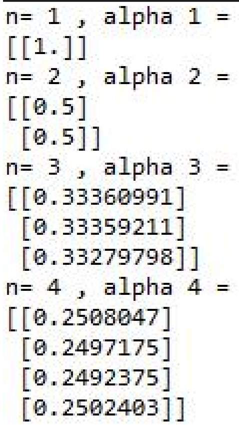
```
def alpha1(hn_inv,data):    # two inputs: the inverse of Hn and n observations
    d=len(data)             # the number of observation
    e=np.ones((d,1))        # n*1 column vector
    kj=np.zeros((d,1))      # kn vector
    for i in range(d):
        kj[i][0]=k(data[i,:],data[i,:],sigma)
    p1=2-np.matrix(e.T)*np.matrix(hn_inv)*np.matrix(kj)
    p2=np.matrix(e.T)*np.matrix(hn_inv)*np.matrix(e)
    alpha=0.5*np.matrix(hn_inv)*(np.matrix(kj)+(p1[0,0]/p2[0,0])*np.matrix(e))
    return (alpha)
```

After defining the function alpha1 (hn_inv, data), we can test it by inputting $\boldsymbol{H}_{\mathbf{1}}^{-\mathbf{1}}$ shown by 'hn_1_inv' at first and compute the $\boldsymbol{\alpha}_{\boldsymbol{n}}$ vector named as 'alpha_n' in a loop as below.

```
hn_1_inv=h1_inv       # the inverse of Hn-1 starts from the inverse of H1
N=len(datax)          # the number of observations
for i in range(N):
    alpha_n=alpha1(hn_1_inv,np.array(datax)[0:i+1])
    hn_inv=Hn_inv(hn_1_inv,np.array(datax)[0:i+1],np.array(datax)[i+1])
    hn_1_inv=hn_inv
    print("n=",(i+1),",","alpha",(i+1),"=")
    print(alpha_n)
```

Similarly, here we only display the part of results of the $\boldsymbol{\alpha}_{\boldsymbol{n}}$ vector from n=1 to n=4 as below.

<u># the results of a1~a4:</u>



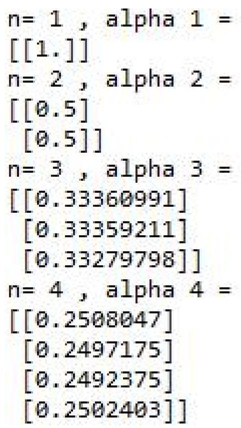
```
n= 1 , alpha 1 =
[[1.]]
n= 2 , alpha 2 =
[[0.5]
 [0.5]]
n= 3 , alpha 3 =
[[0.33360991]
 [0.33359211]
 [0.33279798]]
n= 4 , alpha 4 =
[[0.2508047]
 [0.2497175]
 [0.2492375]
 [0.2502403]]
```

c. Apply your function to the set obtained with the first row and add sequentially the next 6 rows. Check that your code work correctly by comparing the α obtained using the recursive updates (i.e. α2, α3, α4, α5, α6, α7) to the actual α's using the direct approach (α2, α3, α4, α5, α6, α7).

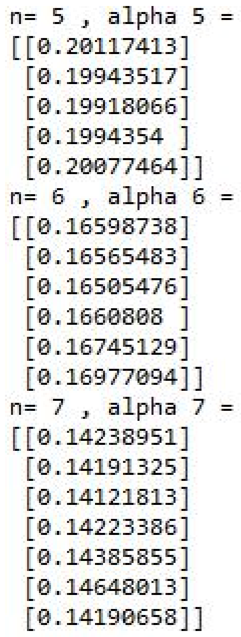
For the iterative method, we use a loop to compute both 'alpha_n' and 'hn_inv' in the iterative fashion. Assume we compute α1~ α7 shown as below.

```
hn_1_inv=h1_inv      # the inverse of Hn-1 starts from the inverse of H1
N=7                           # assume we have 7 observations
for i in range(N):
    alpha_n=alpha1(hn_1_inv,np.array(datax)[0:i+1])
    hn_inv=Hn_inv(hn_1_inv,np.array(datax)[0:i+1],np.array(datax)[i+1])
    hn_1_inv=hn_inv
    print("n=",(i+1),",","alpha",(i+1),"=")
    print(alpha_n)
```

<u>#the results of alpha vector by the iterative method:</u>



```
n= 1 , alpha 1 =
[[1.]]
n= 2 , alpha 2 =
[[0.5]
 [0.5]]
n= 3 , alpha 3 =
[[0.33360991]
 [0.33359211]
 [0.33279798]]
n= 4 , alpha 4 =
[[0.2508047]
 [0.2497175]
 [0.2492375]
 [0.2502403]]
```



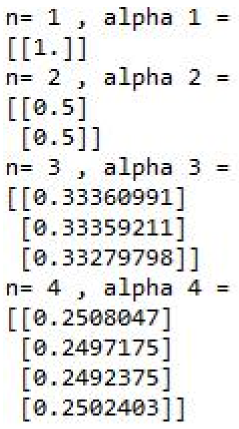
```
n= 5 , alpha 5 =
[[0.20117413]
 [0.19943517]
 [0.19918066]
 [0.1994354 ]
 [0.20077464]]
n= 6 , alpha 6 =
[[0.16598738]
 [0.16565483]
 [0.16505476]
 [0.1660808 ]
 [0.16745129]
 [0.16977094]]
n= 7 , alpha 7 =
[[0.14238951]
 [0.14191325]
 [0.14121813]
 [0.14223386]
 [0.14385855]
 [0.14648013]
 [0.14190658]]
```

For the direct approach, we define a function Hn (data) to compute Hn named as ‘hn’ directly, where the input ‘data’ means n observations. In this function, ‘d’ denotes the number of observations, ‘Km’ means K matrix and ‘I’ denotes the identity matrix. Then using a loop to compute ‘hn’ and its inverse ‘hn_inv, as well as ‘alpha_n’.

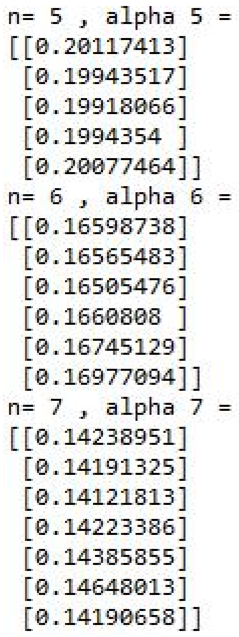
```
def Hn(data):           #define a function of computing Hn directly
    d=len(data)         # the number of observations
    Km=np.zeros((d,d))  # K matrix named as Km
    I=np.eye(d)         # d dimensions identity matrix
    for i in range(d):
        for j in range(d):
            Km[i][j]=k(data[i,:],data[j,:],sigma)
    hn=Km+(1/(2*C))*I    # compute the matrix Hn
    return (hn)

hn_1_inv=h1_inv     # the inverse of Hn-1 starts from the inverse of H1
N=7                 # assume we have 7 observations
for i in range(N): # start the loop
    hn_inv=inv(Hn(np.array(datax)[0:i+1]))
    alpha_n=alpha1(hn_inv,np.array(datax)[0:i+1])
    print("n=",(i+1),",","alpha",(i+1),"=")
    print(alpha_n)
```

#the results of alpha vector by the iterative method:



```
n= 1 , alpha 1 =
[[1.]]
n= 2 , alpha 2 =
[[0.5]
 [0.5]]
n= 3 , alpha 3 =
[[0.33360991]
 [0.33359211]
 [0.33279798]]
n= 4 , alpha 4 =
[[0.2508047]
 [0.2497175]
 [0.2492375]
 [0.2502403]]
```



```
n= 5 , alpha 5 =
[[0.20117413]
 [0.19943517]
 [0.19918066]
 [0.1994354 ]
 [0.20077464]]
n= 6 , alpha 6 =
[[0.16598738]
 [0.16565483]
 [0.16505476]
 [0.1660808 ]
 [0.16745129]
 [0.16977094]]
n= 7 , alpha 7 =
[[0.14238951]
 [0.14191325]
 [0.14121813]
 [0.14223386]
 [0.14385855]
 [0.14648013]
 [0.14190658]]
```

Therefore, we can see the results of the $\boldsymbol{\alpha_n}$ vector of both the iterative and direct methods are same.

d. Use your results from (b) to sequentially update the radius R_n^2 and dz. Check sequentially if the next 7 rows are targets or outliers by comparing d_{xn} to R_{n-1}^2 .

Our steps can be : (1) define the function R2(alpha_n, train) with two inputs (the $\boldsymbol{\alpha_n}$ vector and the training dataset) and an output (the radius R_n^2) named as 'r2'; (2) define the function DZ (alpha_n,train,test) with three inputs (the $\boldsymbol{\alpha_n}$ vector, the training dataset and a new test vector) and an output dz value named as 'dz'; (3) use a loop to check d_{xn} and R_{n-1}^2.

Step (1) defines the function R2(alpha_n, train) with two inputs (the $\boldsymbol{\alpha_n}$ vector and the training dataset) and an output (the radius R_n^2) named as 'r2'.

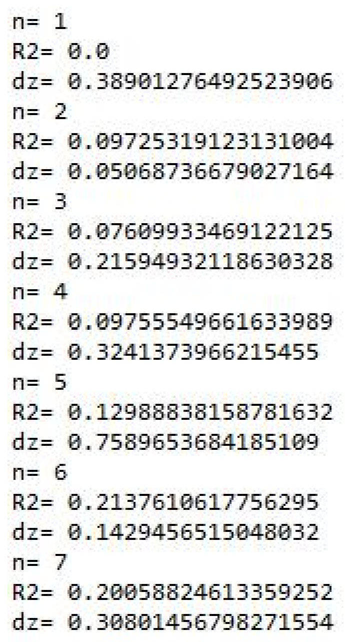
```
def R2(alpha_n,train):  # R2 consists of three average parts: ap1, ap2, ap3
    ap1=0
    ap2=0
    ap3=0
    for i in range(len(train)):
        ap1+=k(train[i,:],train[i,:],sigma) # get the ap1
    for i in range(len(train)):
        for j in range(len(train)):       # get the ap2
            ap2+=2*alpha_n[j,0]*k(train[i,:],train[j,:],sigma)
    for s in range(len(train)):
        for i in range(len(train)):
            for j in range(len(train)):   # get the ap3
                ap3+=alpha_n[i,0]*alpha_n[j,0]*k(train[i,:],train[j,:],sigma)
    r2=(ap1-ap2+ap3)/len(train)  # get the average of ap1, ap2 and ap3
    return(r2)
```

Step (2) defines the function DZ (alpha_n,train,test) with three inputs (the $\boldsymbol{\alpha_n}$ vector, the training dataset and a new test vector) and an output dz value named as 'dz'.

```
def DZ(alpha_n,train,test): # assume dz consists of three parts:p1,p2,p3
    p1=k(test,test,sigma)
    p2=0
    p3=0
    for i in range(len(train)):
        p2+=2*alpha_n[i,0]*k(test,train[i,:],sigma)
    for i in range(len(train)):
        for j in range(len(train)):
            p3+=alpha_n[i,0]*alpha_n[j,0]*k(train[i,:],train[j,:],sigma)
    dz=p1-p2+p3
    return(dz)
```

Step (3) uses a loop to check d_{xn} and R_{n-1}^2 in an iterative fashion. Assume we have a first set with 7 observations and the next new 7 observations.
For the first set, we sequentially update R_n^2 and dz as below.

```
tn1=7               # a first set with 7 observations
hn_1_inv=h1_inv     # the inverse of Hn-1 starts from the inverse of H1
for i in range(tn1):
    alpha_n=alpha1(hn_1_inv,np.array(datax)[0:i+1])
    r2=R2(alpha_n,np.array(datax)[0:i+1])
    dz=DZ(alpha_n,np.array(datax)[0:i+1],np.array(datax)[i+1])
    hn_inv=Hn_inv(hn_1_inv,np.array(datax)[0:i+1],np.array(datax)[i+1])
    hn_1_inv=hn_inv
    print("n=",(i+1))
    print("R2=",r2)
```
# the results of R2 and dz from n=1 to n=7:



```
n= 1
R2= 0.0
dz= 0.38901276492523906
n= 2
R2= 0.09725319123131004
dz= 0.05068736679027164
n= 3
R2= 0.07609933469122125
dz= 0.21594932118630328
n= 4
R2= 0.09755549661633989
dz= 0.3241373966215455
n= 5
R2= 0.12988838158781632
dz= 0.7589653684185109
n= 6
R2= 0.2137610617756295
dz= 0.1429456515048032
n= 7
R2= 0.20058824613359252
dz= 0.30801456798271554
```

For the next 7 new rows, we check them iteratively as below.

```
tn2=14
hn_1_inv=h1_inv    # the inverse of Hn-1 starts from the inverse of H1
for i in range(tn2):
    alpha_n=alpha1(hn_1_inv,np.array(datax)[0:i+1])  # iterative alpha_n
    r2=R2(alpha_n,np.array(datax)[0:i+1])            # iterative R2 (n)
    dz=DZ(alpha_n,np.array(datax)[0:i+1],np.array(datax)[i+1]) # dz of xn
    hn_inv=Hn_inv(hn_1_inv,np.array(datax)[0:i+1],np.array(datax)[i+1])
    hn_1_inv=hn_inv
    if i>6:      # predict the next 7 rows
        print("n=",(i+1))
        print("R2=",r2)
        print("dz=",dz)
        if dz<=r2:
            print("It's a target")
        else:
            print("It's an outlier")
```

# the results of the next 7 rows:

```
n= 8
R2= 0.20928689796795752
dz= 0.03775570765812686
It's a target
n= 9
R2= 0.189695965095094
dz= 0.1545786127931802
It's a target
n= 10
R2= 0.18472528939444272
dz= 0.05800361691682876
It's a target
n= 11
R2= 0.17268184890295968
dz= 0.3579090039802064
It's an outlier
n= 12
```

```
R2= 0.18574779243244022
dz= 0.2863379510678793
It's an outlier
n= 13
R2= 0.19184169626774705
dz= 0.05963809200109371
It's a target
n= 14
R2= 0.18209561908632516
dz= 0.1345539546015615
It's a target
```

e. Compare the training times (in sec) between the 2 algorithms.

The training time mainly relies on the time of computing 'hn_inv' and 'alpha_n' iteratively or directly. Here we use time.time() to record the runtimes of the iterative and direct methods, respectively.

For the iterative method, the start time is denoted by 'start_time1' and the runtime would be 'time.time()-start_time1' and the timing history is kept in the array 't1'.

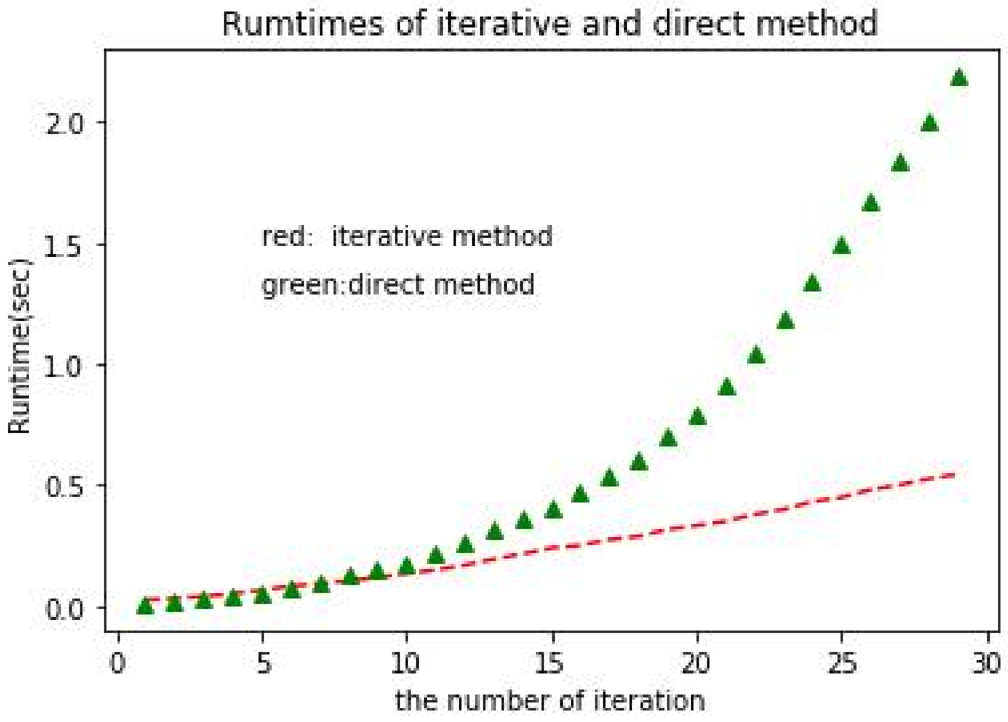
```
start_time1 = time.time()
hn_1_inv=h1_inv    # the inverse of Hn-1 starts from the inverse of H1
N=30  # we have 30 observations
t1=np.zeros(N)     # record the timing as n increases for iterative method
for i in range(N):
    alpha_n=alpha1(hn_1_inv,np.array(datax)[0:i+1])
    hn_inv=Hn_inv(hn_1_inv,np.array(datax)[0:i+1],np.array(datax)[i+1])
    hn_1_inv=hn_inv
    print("n=",(i+1),",","alpha",(i+1),"=")
    print("iterative method timing:",time.time()-start_time1, "seconds")
    t1[i]=time.time()-start_time1
```

For the direct approach, the start time is denoted by 'start_time2' and the runtime would be 'time.time()-start_time2' and the timing history is kept in the array 't2'.

```
start_time2 = time.time()
N=30  # we have 30 observations
t2=np.zeros(N)     # record the timing as n increases for direct method
for i in range(N):
    hn_inv=inv(Hn(np.array(datax)[0:i+1]))
    alpha_n=alpha1(hn_inv,np.array(datax)[0:i+1])
    print("n=",(i+1),",","alpha",(i+1),"=")
    print("direct approach timing:",time.time()-start_time2,"seconds")
    t2[i]=time.time()-start_time2
```

Eventually, we can compare the runtimes of two methods by depicting the time changes as the number of iteration increases shown as the following plot. The plot shows before the number of iteration is 10, the runtimes of both methods are almost same, while the runtime of the direct method increases much more than the iterative method when the number of iteration is more than 10.

```
plt.plot(np.arange(1,30,1), t1, 'r--', np.arange(1,30,1), t2,'g^')
plt.xlabel('the number of iteration')
plt.ylabel('Runtime(sec)')
plt.title('Rumtimes of iterative and direct method')
plt.text(5, 1.5, 'red:  iterative method')
plt.text(5, 1.3, 'green:direct method')
plt.show()
```



Problem 2

a. Write a Python function to obtain the block update inverse matrix $\boldsymbol{H}_{\boldsymbol{N+K}}^{-1}$.

The main procedures are : (1) read the Group I data file and extract the target independent variables (x1~x9); (2) define the Gaussian kernel k (xi, xj, sigma) with three inputs (i-th and j-th observations, the width of the Gaussian kernel); (3) define the function HNK_inv2 (hn_inv, data, newdata) to update inverse matrix $\boldsymbol{H}_{\boldsymbol{N+K}}^{-1}$, where the three inputs of this function denote $\boldsymbol{H}_{\boldsymbol{N}}^{-1}$, the N observations and the K new observation.

I firstly import some packages we may require as below.

```
import math
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import statsmodels.api as sm
import scipy.stats as ss
from scipy.stats import norm
from numpy.linalg import norm
from numpy.linalg import inv
import random
import time
import csv
```

step (1) reads the data file by pandas, kept in 'group1' firstly. Then, split string lines into multiple columns including (x1~x9) and the response variable (positive or negative), kept in 'group1_data'. Next, we transform categorical levels ('x', 'o', 'b') into continuous ones (1, 2, 3), kept in 'group1_data'. Also, we separate the dataset into the 'positive' part in 'data_pos' and the 'negative' part in 'data_neg'. Eventually, we extract the target independent variables (x1~x9) and convert them into 'int' type, kept in 'data_pos_x' and 'data_neg_x'.

```
with open("E:\\Desk\\2019\\STA6160-statistical computing\\tic-tac-toe.txt") as f:
    group1 = f.read().splitlines()
group1_data=np.array(list(csv.reader(group1, delimiter=',')))
```

Transform categorical levels ('x', 'o', 'b') into continuous ones (1, 2, 3) as below. And separate 'group1_data' into positive and negative parts, 'data_pos' and 'data_neg' .

```
for i in range(group1_data.shape[0]):
    for j in range(group1_data.shape[1]):
        if group1_data[i,j]=='x':
            group1_data[i,j]=1
        if group1_data[i,j]=='o':
            group1_data[i,j]=2
        if group1_data[i,j]=='b':
            group1_data[i,j]=3
data_pos=group1_data[group1_data[:,9]=='positive']
data_neg=group1_data[group1_data[:,9]=='negative']
```

we extract the target independent variables (x1~x9) and convert them into 'int' type, kept in 'data_pos_x' and 'data_neg_x'.

```
data_pos_x=np.zeros((data_pos.shape[0],data_pos.shape[1]-1))
```

```
for i in range(data_pos.shape[0]):#only contain independent variables
    for j in range(data_pos.shape[1]-1):
       data_pos_x[i,j]= int(data_pos[i,j])
data_neg_x=np.zeros((data_neg.shape[0],data_neg.shape[1]-1))
for i in range(data_neg.shape[0]):#only contain independent variables
    for j in range(data_neg.shape[1]-1):
       data_neg_x[i,j]= int(data_neg[i,j])
```

step (2) defines the Gaussian kernel k (xi, xj, sigma) with three inputs (i-th and j-th observations, the width of the Gaussian kernel) and an return value named 'kij'.

```
def k(xi,xj,sigma): #xi,xj,sigma are three inputs
    kij=np.exp(-np.power(norm(xi-xj),2)/np.power(sigma,2))
return(kij)
```

step (3) defines the function HNK_inv2 (hn_inv, data, newdata) to update inverse matrix $\boldsymbol{H}_{\boldsymbol{N+K}}^{-1}$, where the three inputs of this function denote $\boldsymbol{H}_{\boldsymbol{N}}^{-1}$, the N observations and the K new observations. Note that $\boldsymbol{B}$ and $\boldsymbol{D}$ matrice are named as 'B' and 'D'; 'w1~w4' are four entries in $\boldsymbol{H}_{\boldsymbol{N+K}}^{-1}$. The return value in function HNK_inv2 is $\boldsymbol{H}_{\boldsymbol{N+K}}^{-1}$ shown as 'hnk_inv'.

```
def HNK_inv2(hn_inv,data,newdata):
    N=len(data)
    K=len(newdata)
    B=np.zeros((K,N))  # matrix B: K*N
    D=np.zeros((K,K))  # matrix D: K*K
    I=np.eye(K)        # K*K identity matrix
    for i in range(K):
        for j in range(N):
            B[i,j]=k(newdata[i,:],data[j,:],sigma)
    for i in range(K):
        for j in range(K):
            D[i,j]=k(newdata[i,:],newdata[j,:],sigma)
    D=D+(1/(2*C))*I
    B=np.matrix(B)
    D=np.matrix(D)
    hn_inv=np.matrix(hn_inv)
    w1=np.array(hn_inv-hn_inv*B.T*inv(-D+B*hn_inv*B.T)*B*hn_inv)
    w2=np.array(hn_inv*B.T*inv(B*hn_inv*B.T-D))
    w3=np.array(inv(B*hn_inv*B.T-D)*B*hn_inv)
    w4=np.array(inv(D-B*hn_inv*B.T))
    hnk_inv=np.block([[w1,w2],[w3,w4]])
    return(hnk_inv)
```

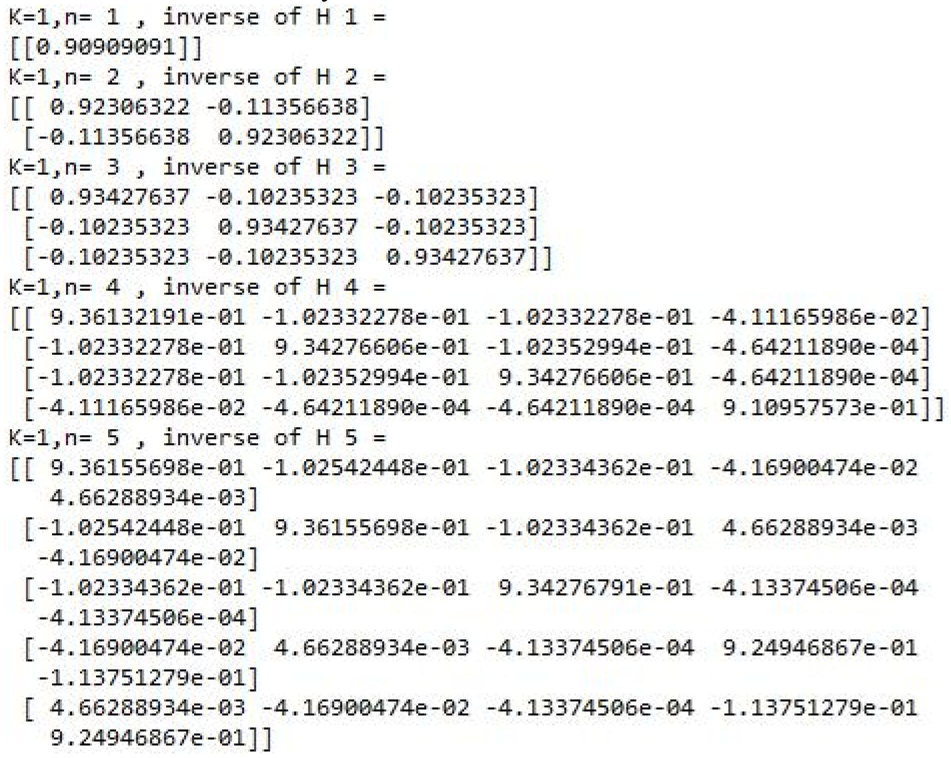
b. Check that the results match the ones using the direct approach.

For the iterative block method, we can test HNK_inv2 function and assume here K=1, C=5, sigma=1. Then we can start from $\boldsymbol{H}_{\boldsymbol{1}}^{-1}$ named as 'h1_inv', then use a loop to compute $\boldsymbol{H}_{\boldsymbol{N+K}}^{-1}$. Here we only display the part of results from n=1 to n=5.

```
C=5                           # the penalty factor
sigma=1                       # the parameter of Gaussian Kernel
h1=np.matrix(k(np.array(data_pos_x)[0],np.array(data_pos_x)[0],sigma)+1/(2*C))
h1_inv=np.array(inv(h1))    # the inverse of H1
```

```
n=5                        # assum the number of observation=5
hn_inv=h1_inv            # the inverse of Hn starts from the inverse of H1
for i in range(n):
    print("K=1,n=",(i+1),",","inverse","of","H",(i+1),"=")
    print(hn_inv)
    hnk_inv=HNK_inv2(hn_inv,np.array(data_pos_x)[0:i+1],
                     np.array(data_pos_x)[i+1:i+2])
    hn_inv=hnk_inv
```

<u># the results of H1~H5 by the iterative block method:</u>

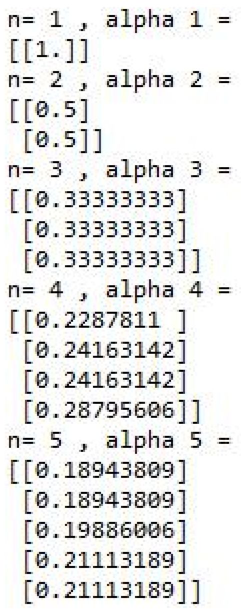


```
K=1,n= 1 , inverse of H 1 =
[[0.90909091]]
K=1,n= 2 , inverse of H 2 =
[[ 0.92306322 -0.11356638]
 [-0.11356638  0.92306322]]
K=1,n= 3 , inverse of H 3 =
[[ 0.93427637 -0.10235323 -0.10235323]
 [-0.10235323  0.93427637 -0.10235323]
 [-0.10235323 -0.10235323  0.93427637]]
K=1,n= 4 , inverse of H 4 =
[[ 9.36132191e-01 -1.02332278e-01 -1.02332278e-01 -4.11165986e-02]
 [-1.02332278e-01  9.34276606e-01 -1.02352994e-01 -4.64211890e-04]
 [-1.02332278e-01 -1.02352994e-01  9.34276606e-01 -4.64211890e-04]
 [-4.11165986e-02 -4.64211890e-04 -4.64211890e-04  9.10957573e-01]]
K=1,n= 5 , inverse of H 5 =
[[ 9.36155698e-01 -1.02542448e-01 -1.02334362e-01 -4.16900474e-02
   4.66288934e-03]
 [-1.02542448e-01  9.36155698e-01 -1.02334362e-01  4.66288934e-03
  -4.16900474e-02]
 [-1.02334362e-01 -1.02334362e-01  9.34276791e-01 -4.13374506e-04
  -4.13374506e-04]
 [-4.16900474e-02  4.66288934e-03 -4.13374506e-04  9.24946867e-01
  -1.13751279e-01]
 [ 4.66288934e-03 -4.16900474e-02 -4.13374506e-04 -1.13751279e-01
   9.24946867e-01]]
```

In the similar way, we can compute ‘alpha_n’ vector from n=1 to n=5 by the iterative block method.

```
# iterative method
hn_inv=h1_inv    # the inverse of Hn starts from the inverse of H1
n=5
for i in range(n):
    alpha_n=alpha1(hn_inv,data_pos_x[0:i+1])
    hnk_inv=HNK_inv2(hn_inv,data_pos_x[0:i+1],data_pos_x[i+1:i+2])
    hn_inv=hnk_inv
    print("n=",(i+1),",","alpha",(i+1),"=")
    print(alpha_n)
```

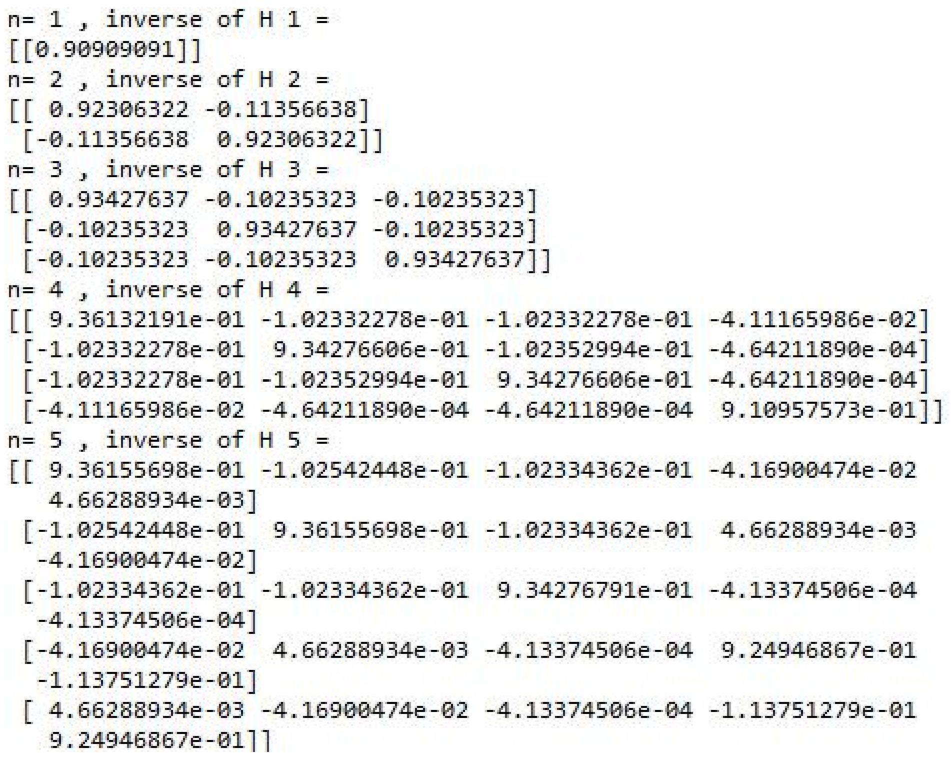
<u># the results of $\boldsymbol{\alpha}_1$~$\boldsymbol{\alpha}_5$ by the iterative block method:</u>


```
n= 1 , alpha 1 =
[[1.]]
n= 2 , alpha 2 =
[[0.5]
 [0.5]]
n= 3 , alpha 3 =
[[0.33333333]
 [0.33333333]
 [0.33333333]]
n= 4 , alpha 4 =
[[0.2287811 ]
 [0.24163142]
 [0.24163142]
 [0.28795606]]
n= 5 , alpha 5 =
[[0.18943809]
 [0.18943809]
 [0.19886006]
 [0.21113189]
 [0.21113189]]
```

For the direct method, assume K=1, C=5, sigma=1, then we compute $\boldsymbol{H}_{\boldsymbol{N+1}}$ based on the matrix K named as 'Km', C and the identity matrix I. we firstly define a function Hn(data) with an input (n observations) and an output (Hn named as 'hn'). Then we use a loop to compute $\boldsymbol{H}_{\boldsymbol{N+1}}^{-\boldsymbol{1}}$ by inv($\boldsymbol{H}_{\boldsymbol{N+1}}$). Here we use 5 observations to compute and correspond to the results through the iterative block method.

```
def Hn(data):  #define a function of computing Hn directly
    d=len(data) # the number of observation
    Km=np.zeros((d,d)) # K matrix named as Km
    I=np.eye(d) # d dimensions identity matrix
    for i in range(d):
        for j in range(d):
            Km[i][j]=k(data[i,:],data[j,:],sigma)
    hn=Km+(1/(2*C))*I # compute the matrix Hn-1
    return (hn)
N=5
for i in range(N):
    hn_inv=inv(Hn(np.array(data_pos_x)[0:i+1]))
    print("n=",(i+1),",","inverse","of","H",(i+1),"=")
    print(hn_inv)
```

<u># the results of H1~H5 through the direct method:</u>

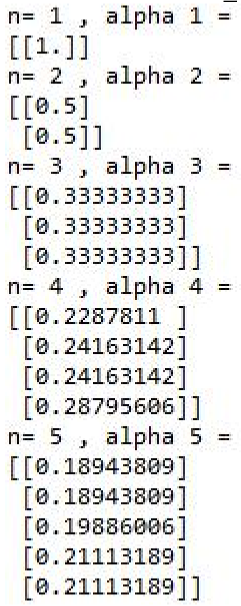


```
n= 1 , inverse of H 1 =
[[0.90909091]]
n= 2 , inverse of H 2 =
[[ 0.92306322 -0.11356638]
 [-0.11356638  0.92306322]]
n= 3 , inverse of H 3 =
[[ 0.93427637 -0.10235323 -0.10235323]
 [-0.10235323  0.93427637 -0.10235323]
 [-0.10235323 -0.10235323  0.93427637]]
n= 4 , inverse of H 4 =
[[ 9.36132191e-01 -1.02332278e-01 -1.02332278e-01 -4.11165986e-02]
 [-1.02332278e-01  9.34276606e-01 -1.02352994e-01 -4.64211890e-04]
 [-1.02332278e-01 -1.02352994e-01  9.34276606e-01 -4.64211890e-04]
 [-4.11165986e-02 -4.64211890e-04 -4.64211890e-04  9.10957573e-01]]
n= 5 , inverse of H 5 =
[[ 9.36155698e-01 -1.02542448e-01 -1.02334362e-01 -4.16900474e-02
   4.66288934e-03]
 [-1.02542448e-01  9.36155698e-01 -1.02334362e-01  4.66288934e-03
  -4.16900474e-02]
 [-1.02334362e-01 -1.02334362e-01  9.34276791e-01 -4.13374506e-04
  -4.13374506e-04]
 [-4.16900474e-02  4.66288934e-03 -4.13374506e-04  9.24946867e-01
  -1.13751279e-01]
 [ 4.66288934e-03 -4.16900474e-02 -4.13374506e-04 -1.13751279e-01
   9.24946867e-01]]
```

In the similar way, we can compute ‘alpha_n’ vector from n=1 to n=5 by the direct method.

```
for i in range(n): # compute alpha_n by the direct method
    hnk_inv=inv(Hn(data_pos_x[0:i+1]))
    alpha_n=alpha1(hnk_inv,data_pos_x[0:i+1])
    print("n=",(i+1),",","alpha",(i+1),"=")
    print(alpha_n)
```

<u># the results of $\boldsymbol{\alpha}_1$~$\boldsymbol{\alpha}_5$ by the direct method:</u>



```
n= 1 , alpha 1 =
[[1.]]
n= 2 , alpha 2 =
[[0.5]
 [0.5]]
n= 3 , alpha 3 =
[[0.33333333]
 [0.33333333]
 [0.33333333]]
n= 4 , alpha 4 =
[[0.2287811 ]
 [0.24163142]
 [0.24163142]
 [0.28795606]]
n= 5 , alpha 5 =
[[0.18943809]
 [0.18943809]
 [0.19886006]
 [0.21113189]
 [0.21113189]]
```

Therefore, comparing the results of Hn and alpha_n through the iterative block method and the direct method indicates that both methods obtain the same results.

c. We will test the speed and effectiveness of the presented algorithm by applying LS-SVDD, LS-SVDD update (K = 1) from problem 1, and LS-SVDD block update (K = 15) to the following data sets available from the UCI Machine Learning Repository. We will use a Gaussian kernel with σ = 1. Also, we will set C to 5 and N to 1. Your results should include "Training accuracy (%)", "Testing accuracy (%)" and "Training Time (s)".

For K=1, Our steps are: (1) redefine the radius and dz functions named as R2(alpha_n,train,ap3) and DZ(alpha_n,train,test,ap3) for accelerating the later computations, where the R2() function has three inputs (the alpha_n vector, the training set, and the constant only related to the training set); the DZ() function has four inputs (the alpha_n vector, the training set, the test vector, and the constant only related to the training set); (2) split datasets into 90% 'data_pos_x'for training and 10% 'data_pos_x' for test, so we get the 'train' and 'test' datasets; (3) initialize the parameters for training; (4) start the training process and record 'Training time'; (5) compute the 'training accuracy' and the 'testing accuracy'.

Step (1) redefines the radius and dz functions named as R2(alpha_n,train,ap3) for accelerating the later computations, where the R2() function has three inputs (the alpha_n vector, the training set, and the constant only related to the training set) and its return value is the radius value named as 'r2'.

```
def R2(alpha_n,train,ap3):        # R2 consists of three average parts: ap1,ap2,ap3
    p1=np.zeros((len(train),1))
    p2=np.zeros((len(train),len(train)))
    for i in range(len(train)):
        p1[i,0]=k(train[i,:],train[i,:],sigma)
    ap1=sum(p1)/len(train)
    for i in range(len(train)):
        for j in range(len(train)):
            p2[i,j]=k(train[i,:],train[j,:],sigma)
    ap2=np.array(sum(np.matrix(p2)*np.matrix(alpha_n)))/ len(train)
    r2=ap1-2*ap2+ap3
    return(r2)
```

Redefine DZ(alpha_n,train,test,ap3) for accelerating the later computations, where the DZ() function has four inputs (the alpha_n vector, the training set, the test vector, and the constant only related to the training set) and its return value is the dz value named as 'dz'.

```
def DZ(alpha_n,train,test,ap3):     # dz consists of three parts: ap1, ap2, ap3
    ap1=k(test,test,sigma)
    p2=np.zeros((len(train),1))
    for i in range(len(train)):
        p2[i,0]=k(test,train[i,:],sigma)
    ap2=np.array(np.matrix(alpha_n.T)*np.matrix(p2))
    dz=ap1-2*ap2+ap3
    return(dz)
```

For the input 'ap3' as a constant only related to the training set, we display the vertorization way to compute it instead of using 'for loop' in problem 1, which aims to save the execution time in the training step.

```
# the third term in dz and R2 is a constant
p3=np.zeros((len(train),len(train)))
for i in range(len(train)):
    for j in range(len(train)):
        p3[i,j]=k(train[i,:],train[j,:],sigma)
ap3=sum(sum(np.outer(alpha_n,alpha_n.T)*p3))
```

Step (2) splits datasets into 90% ‘data_pos_x’for training and 10% ‘data_pos_x’ for test, so we get the ‘train’ and ‘test’ datasets.

```
# split datasets into training and test
indice1=list(range(len(data_pos_x)))
indice2=list(range(len(data_neg_x)))
random.seed(6)
random.shuffle(indice1)
random.shuffle(indice2)
train= data_pos_x[indice1[:571],:]  # 90% for training
test= data_neg_x[indice2[:31],:]   # 10% for test
```

Step (3) initializes the parameters for training by giving H1 and its inverse ‘h1_inv’, setting K=1 by ‘kk=1’ and C=5, sigma=1.

```
h1=np.matrix(k(train[0],train[0],sigma)+1/(2*C))
h1_inv=np.array(inv(h1))  # the inverse of H1
hn_inv=h1_inv             # the inverse of Hn starts from the inverse of H1
kk=1
tn2=int((len(train)-1)/kk) # the number of training set
C=5                       # the penalty factor
sigma=1                 # the width of Gaussian Kernel
```

Step (4) starts the training process and record ‘Training time’. Here we will compute the $\boldsymbol{H_{N+K}^{-1}}$ in a iterative fashion indicated by ‘hnk_incv’, the ‘alpha_n’ vector, the constant input ap3 in functions R2() and DZ(), the radius ‘r2’ and the ‘Training time’ indicated by ‘time.time()-start_time6’.

```
start_time6=time.time() #training starts at K=1
for i in range(tn2):
    hnk_inv=HNK_inv2(hn_inv,train[0:len(hn_inv)],
                     train[len(hn_inv):len(hn_inv)+int(kk*tn2)])
    hn_inv=hnk_inv
    print('the number of iteration=',i+1)
alpha_n=alpha1(hn_inv,train) # iterative alpha_n

p3=np.zeros((len(train),len(train))) # the third term in dz/R2 is a constant
for i in range(len(train)):
    for j in range(len(train)):
        p3[i,j]=k(train[i,:],train[j,:],sigma)
ap3=sum(sum(np.outer(alpha_n,alpha_n.T)*p3)) # the third term in dz/R2 is a constant

r2=(R2(alpha_n,train,ap3))           # iterative R2 (n)

print("R2=",r2[0,0])        # output the trained radius R2
print("the sum of alpha",sum(alpha_n)[0,0]) # output the trained alpha_n
print("K=1, Training time (s):",time.time()-start_time6)
```

<u># the results of training process:</u>

```
the number of iteration= 570
R2= 0.9946269684214775
the sum of alpha: 1.0000000000000044
K=1, Training time (s): 209.32197260856628
```

Step (5) computes the training accuracy denoted by ‘train_accuracy’ (the number of targets ‘ta’ / the number of training observations ‘len(train)’).

```
ta=0
for i in range(len(train)): #compute the training accuracy when K=1
    dz=DZ(alpha_n,train,train[i,:],ap3)
    if dz[0,0]<=r2[0,0]:
        ta+=1
        print("n=",i,"It's a target")
    else:
        print("n=",i,"It's an outlier")
train_accuracy=ta/len(train)
print("K=1, Training accuracy(%):",train_accuracy*100)
```

<u># the result of training accuracy:</u>

```
K=1, Training accuracy(%): 45.008756567425564
```

Similarly, the testing accuracy denoted by ‘test_accuracy’ (the number of outliers ‘len(test)-tt’ / the number of training observations ‘len(test)’).

```
tt=0
for i in range(len(test)): # compute the test accuracy when K=1
    dz=DZ(alpha_n,train,test[i,:],ap3)
    if dz[0,0]<=r2[0,0]:
        tt+=1
        print("n=",i+1,"It's a target")
    else:
        print("n=",i+1,"It's an outlier")
test_accuracy=(len(test)-tt)/len(test)
print("K=1, Testing accuracy(%):",test_accuracy*100)
```

<u># the result of testing accuracy:</u>

```
K=1, Testing accuracy(%): 100.0
```

For K=15, we have the almost same steps and codes like we do for K=1 case above, except now ‘kk=15’. Comparing the results when K=1 and K=15, we can observe that both cases have the same radius R2 and training/testing accuracy, while the training time at K=15 is less than K=1 because of its less iterations than K=1.

<u># the results of training and testing accuracy:</u>

```
the number of iteration= 38
R2= 0.9946269684214775
the sum of alpha 1.0000000000000044
K=15, Training time (s): 151.62067222595215
K=15, Training accuracy(%): 45.008756567425564
K=15, Testing accuracy(%): 100.0
```