به نام خدا

دانشگاه تهران

پردیس دانشکده‌های فنی

دانشکده برق و کامپیوتر

درس سیستم‌های هوشمند

تمرین شماره یکم

نام و نام خانوادگی:

اشکان جعفری

شماره دانشجویی:

۸۱۰۱۹۷۴۸۳

مهر ۱۴۰۰

فهرست سوالات

سوال ۱

الف:

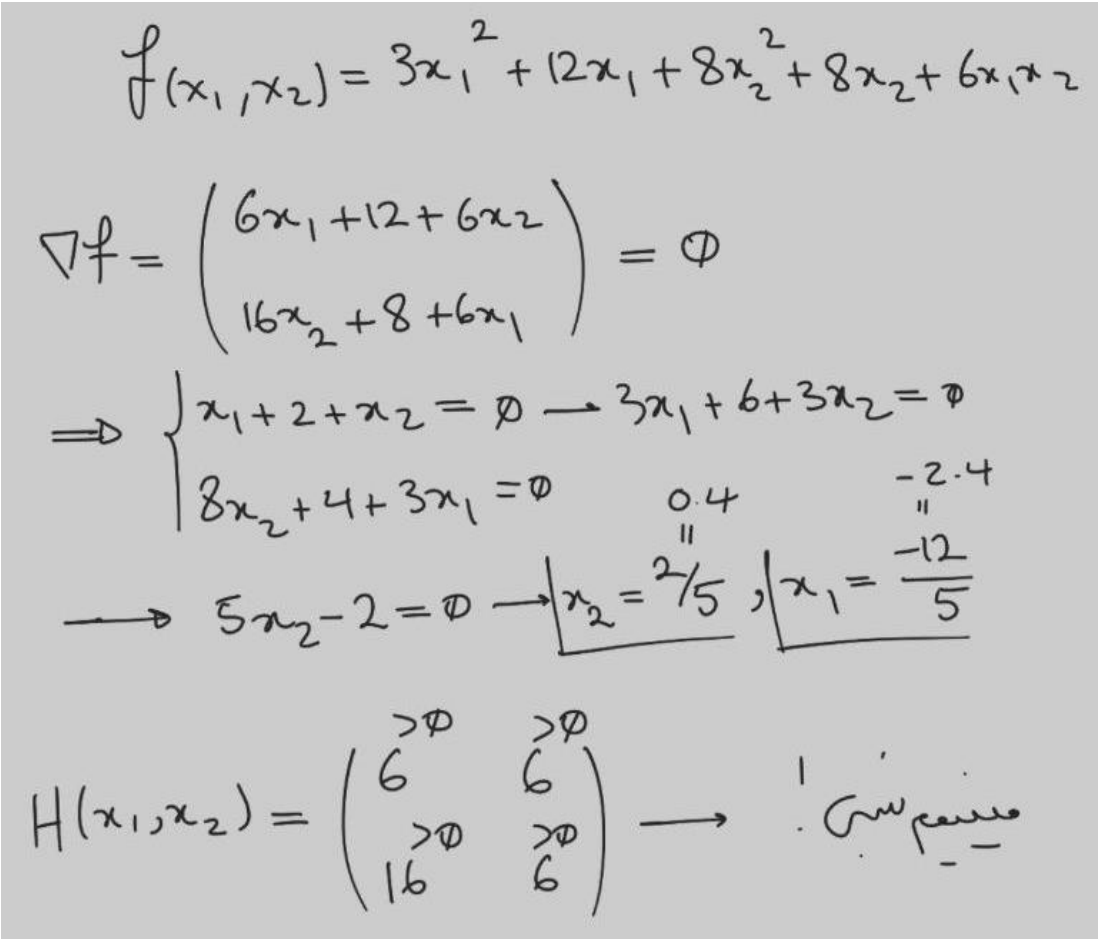
در این بخش ابتدا به کمک برابر با صفر قرار دادن گرادیان تابع، نقاط ایستا را می‌یابیم.

در قدم بعدی با استفاده از علامت ماتریس هسین، نوع این نقاط را می‌یابیم.

در مرحله آخر با شروع از یک نقطه، با استفاده از روش جست و جوی خط، به سمت نقطه بهینه حرکت می‌کنیم.

نهایتاً نقطه ایستای تابع (-2,4,0.4) بدست آمد که با توجه به علامت مثبت ماتریس هسین در تمام نقاط، این نقطه مینیمم مطلق می باشد.

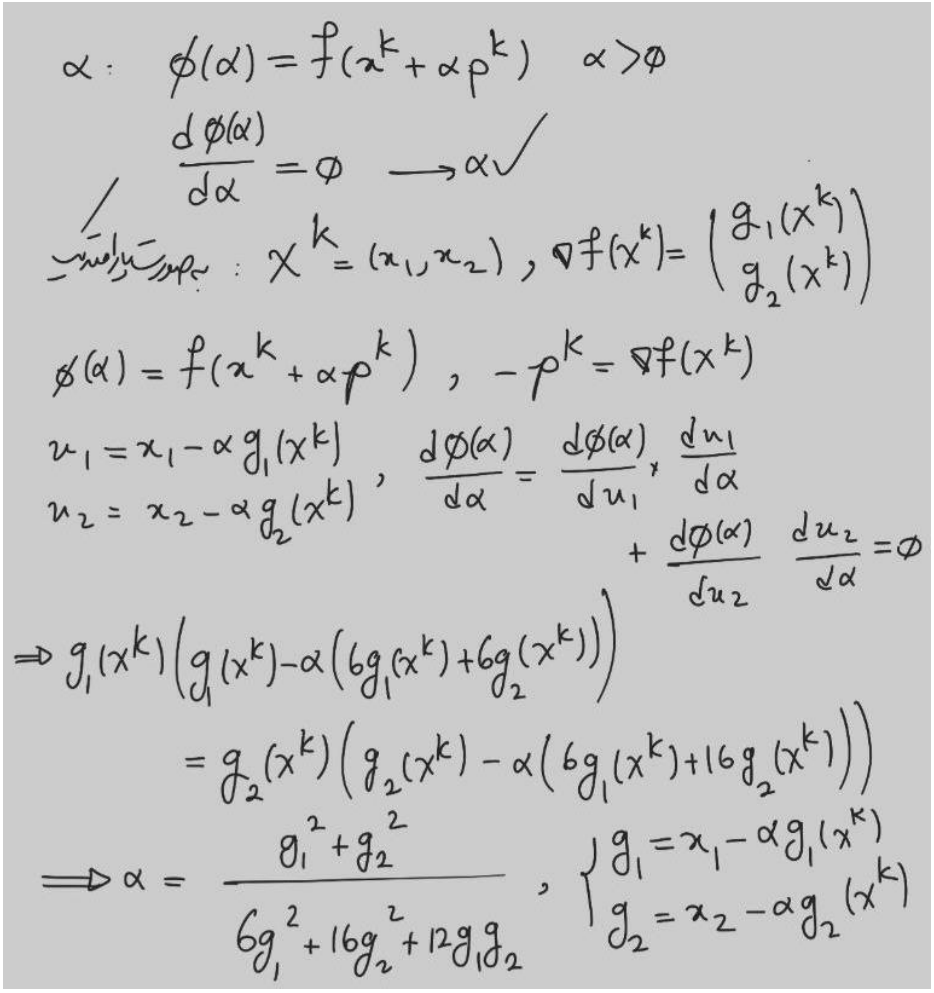
در شکل ۱-۱ محاسبات مذکور قابل رویت است.



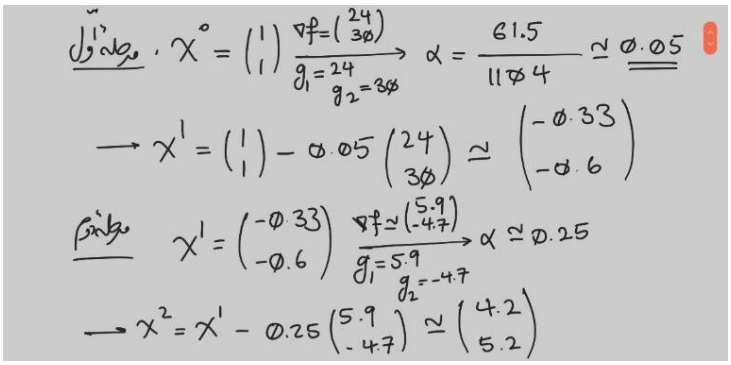
شکل ۱-۱ روند محاسبه نقاط ایستای تابع و نوع آن ها

ب:

هدف این بخش یافتن طول بهینه گام برای الگوریتم گرادیان نزولی به کمک روش تحلیلی است. در شکل ۲-۱ توضیح مختصری در مورد روش تحلیلی، در شکل ۳-۱، محاسبات دستی برای دو مرحله و در فایل Q1.ipynb شبیه سازی کامپیوتری مربوطه قابل رویت است.(در شکل ۱-۳-۱ تصویر قطعه کد آمده است)



شکل ۲-۱ محاسبات پارامتریک تحلیلی برای بدست آوردن طول بهینه گام



شکل ۳-۱ محاسبات تحلیلی طول بهینه گام برای دو مرحله ابتدایی

Question1

```
def gradient_func(X):
    grad = np.array([6*X[0]+12+6*X[1],16*X[1]+8+6*X[0]])
    return grad
def gradient_descent(start, alpha):
    vector = start
    while (True):
        vector =np.add(vector,-1*alpha * gradient_func(vector))
        if norm(gradient_func(vector)) <= 1e-06:
            break
    return vector
gradient_descent(np.array([1,1]),0.06)
```

```
array([-2.39999975,  0.39999988])
```

شکل ۱-۳-۱ قطعه کد شبیه سازی گرادیان نزولی به همراه نتایج

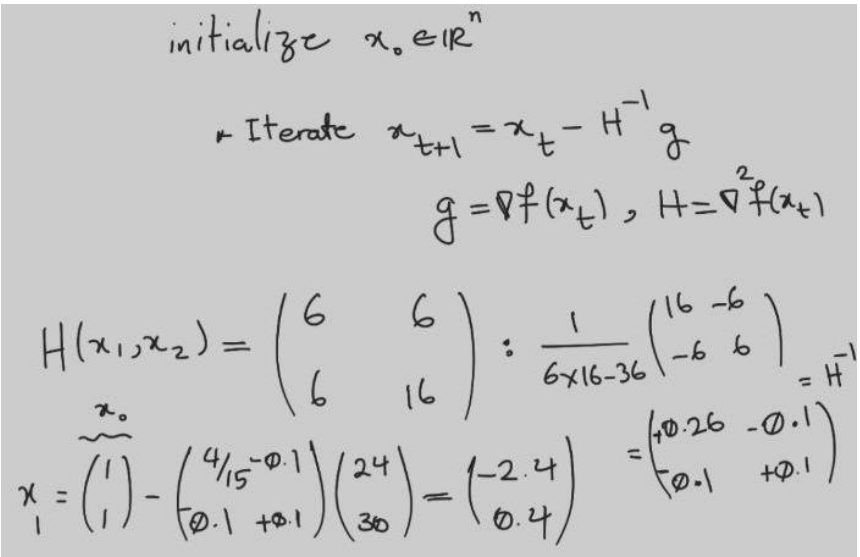
کد به صورت تایپی به شرح زیر است:

```
import numpy as np
from numpy import linalg as LA
from array import *


def gradient_func(X):
    grad = np.array([6*X[0]+12+6*X[1],16*X[1]+8+6*X[0]])
    return grad
def gradient_descent(start, alpha):
    vector = start
    while (True):
        vector =np.add(vector,-1*alpha * gradient_func(vector))
        if LA.norm(gradient_func(vector)) <= 1e-06:
            break
    return vector
gradient_descent(np.array([1,1]),0.06)
```

ج:

به این روش کاربرد دارد اما همیشه به نقطه بهینه مینیمم همگرا نمی‌شود و دقتش بستگی به مرتبه تابع دارد. برای مثال در این تابع، روش نیوتون به دلیل مرتبه دو بودن تابع در اولین گام به نقطه مینیمم مطلق همگرا می‌شود.

محاسبات مربوطه در شکل ۴-۱ قابل مشاهده است.



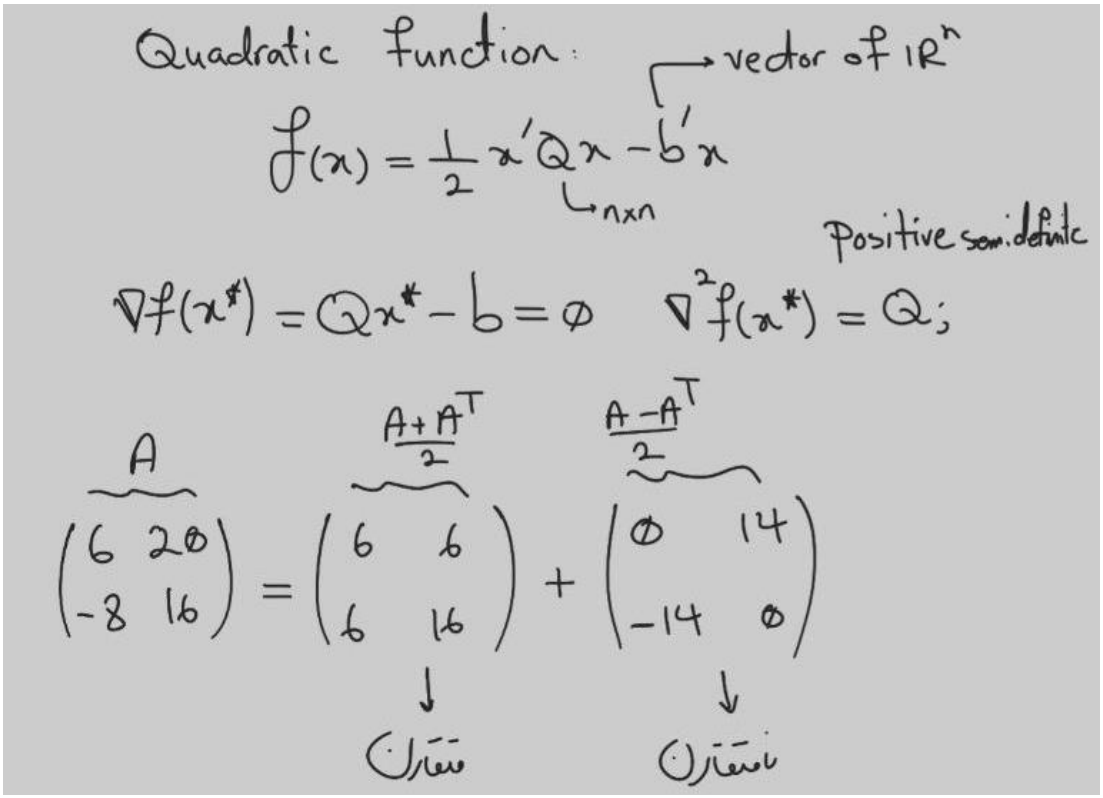
شکل ۴-۱ توضیح کلی روش نیوتون به همراه محاسبات ملزوم برای همگرایی

د:

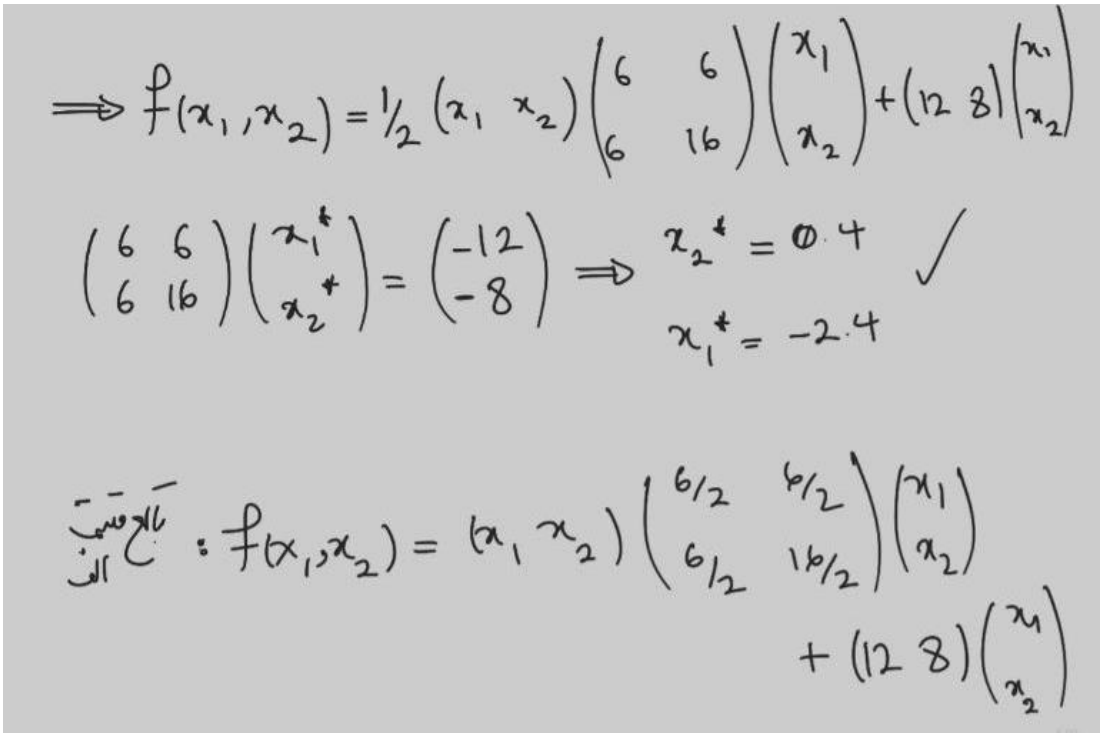
در این بخش قصد داریم محاسبات روش درجه دو را به کمک عملیات ماتریسی انجام دهیم. همانطور که در شکل ۵-۱ مشاهده میشود میتوان در تابع این قسمت، ماتریس $\begin{pmatrix} 6 & 20 \\ -8 & 16 \end{pmatrix}$ را میتوان به صورت مجموع دو تابع متقارن و پادمتقارن نوشت. از جبرخطی میدانیم تابع پادمتقارن تاثیری در محاسبات ما نخواهد داشت(وقتی یک بردار و ترانهاده آن در دوطرف ماتریس ضرب شوند) فلذا با تجزیه ماتریس مذکور مشاهده میکنیم که ماتریس متقارن بدست آمده دقیقاً برابر با ماتریس هسین تابع قسمت الف میباشد و حال میتوانیم با توجه به متقارن بودن ماتریس به دست آمده، محاسبات دیفرانسیلی ماتریسی را انجام دهیم. لازم به ذکر است با توجه به استدلالی که برای برابر تابع بخش د و الف صورت گرفت، میتوان نتیجه گیری کرد که نقطه ایستا در تو تابع برابر و از نوع مینیمم مطلق است.

در شکل ۶-۱ محاسبات مربوط برای مشتق ماتریس و اثبات برابری با قسمت الف آمده است.

در شکل ۵-۱ علارقم توضیحات مربوط به تجزیه ماتریس اصلی، توضیحات روش ماتریسی درجه دو آمده است.



شکل ۵-۱ توضیح کلی روش ماتریسی درجه دو به همراه تجزیه ماتریس تابع بخش د



شکل ۶-۱ محاسبه نقطه ایستا به کمک روش دیفرانسیل ماتریسی

سوال ۲

الف:

در این بخش قصد داریم به کمک روش گرادیان نزولی کمینه های محلی تابع مذکور را پیدا کنیم.

برای طول گام بنده با استفاده از روش طول گام ثابت(0.001)، الگوریتم را پیاده سازی کرده ام.

نقاط شروع را هم یکبار (10,7) و یکبار (11,5) در نظر گرفته ام که در هر مورد به مقدار 75- همگرا شده‌اند. توضیح اضافه تر آنکه شرط اتمام حلقه گرادیان نزولی را تکرار ۱۰۰۰ تایی درنظر گرفتم.

گرادیان تابع مذکور در ادامه آمده است که با استفاده از آن پیاده سازی کامپیوتری صورت گرفته.

$$\nabla f = \begin{pmatrix} 2x_1 + 20\pi x_2 \sin(0.2\pi x_1) - 15\cos(0.4\pi x_2) \\ -10\cos(0.2\pi x_1) + 2\,x_2 + 6\pi x_1 \sin(0.4\pi x_2) \end{pmatrix}$$

کد مربوطه به نام Q2_1.ipynb در پوشه codes ضمیمه شده است. در شکل ۱-۲ قطعه کد به همراه نتیجه نهایی آمده است.

```
def gradient_func(X):
    grad = np.array([2*X[0]+2*np.pi*X[1]*np.sin(0.2*np.pi*X[0])-15*np.cos(0.4*np.pi*X[1]),
                                                2*X[1]+6*np.pi*X[0]*np.sin(0.4*np.pi*X[1])-10*np.cos(0.2*np.pi*X[0])])
    return grad
def gradient_descent(start, alpha):
    vector = start
    iteration=10000
    while (iteration>0):
        vector =np.add(vector,-1*alpha * gradient_func(vector))
        #if norm(gradient_func(vector)) <= 1e-02:
         #   break
        iteration-=1;
    return vector
def main_func(X):
    func = X[0]**2 - 10*X[1]*np.cos(0.2*np.pi*X[0])+ X[1]**2 - 15*X[0]*np.cos(0.4*np.pi*X[1])
    return func
main_func(gradient_descent(np.array([10,7]),0.001))
```

```
-75.57593528101775
```

شکل ۱-۲ قطعه کد برای محاسبه گرادیان نزولی تابع صورت سوال ۲

کد به صورت تایپی به شرح زیر است:

```
def gradient_func(X):
    grad = np.array([2*X[0]+2*np.pi*X[1]*np.sin(0.2*np.pi*X[0])-15*np.cos(0.4*np.pi*X[1]),
                                                2*X[1]+6*np.pi*X[0]*np.sin(0.4*np.pi*X[1])-
10*np.cos(0.2*np.pi*X[0])])
    return grad
def gradient_descent(start, alpha):
    vector = start
    iteration=10000
    while (iteration>0):
        vector =np.add(vector,-1*alpha * gradient_func(vector))
        #if norm(gradient_func(vector)) <= 1e-02:
         #   break
        iteration-=1;
    return vector
def main_func(X):
    func = X[0]**2 - 10*X[1]*np.cos(0.2*np.pi*X[0])+ X[1]**2 - 15*X[0]*np.cos(0.4*np.pi*X[1])
    return func
main_func(gradient_descent(np.array([10,7]),0.001))
```

ب:

هدف این بخش یافتن مقدار طول گام بهینه به دو روش تحلیلی و آرمیجو میباشد. در تمامی پیاده سازی ها نقطه شروع (0,0) میباشد.

در روش تحلیلی با توجه به روابط موجود در سوال یکم (رجوع شود به شکل۲-۱) پیاده سازی کامپیوتری صورت گرفته است.

قطعه کد مربوطه در شکل۲-۲ آمده است و در پوشه codes با نام Q2_2.m ضمیمه شده است.

```
fun = @(a) (15*a)^2 - 10*(10*a)*cos(0.2*pi*15*a) + (10*a)^2 - 15*15*a*cos(0.4*pi*10*a);
alpha0 = fminbnd(fun,0,1)
vector0=[0,0];
vector1 = [0,0] - 1*alpha0*gradient_func(vector0)

grad2=gradient_func(vector1)
fun1 = @(a) (7.8+30.9*a)^2 - 10*(5.2-46.4*a)*cos(0.2*pi*(7.8+30.9*a)) + (5.2-46.4*a)^2 - 15*(7.8+30.9*a)*cos(0.4*pi*((5.2-46.4*a)));
alpha1 = fminbnd(fun1,0,1)


function grad=gradient_func(X)
   grad = [2*X(1)+2*pi*X(2)*sin(0.2*pi*X(1))-15*cos(0.4*pi*X(2)),2*X(2)+6*pi*X(1)*sin(0.4*pi*X(2))-10*cos(0.2*pi*X(1))];
end
```

شکل ۲-۲ قطعه کد برای محاسبه طول بهینه گام

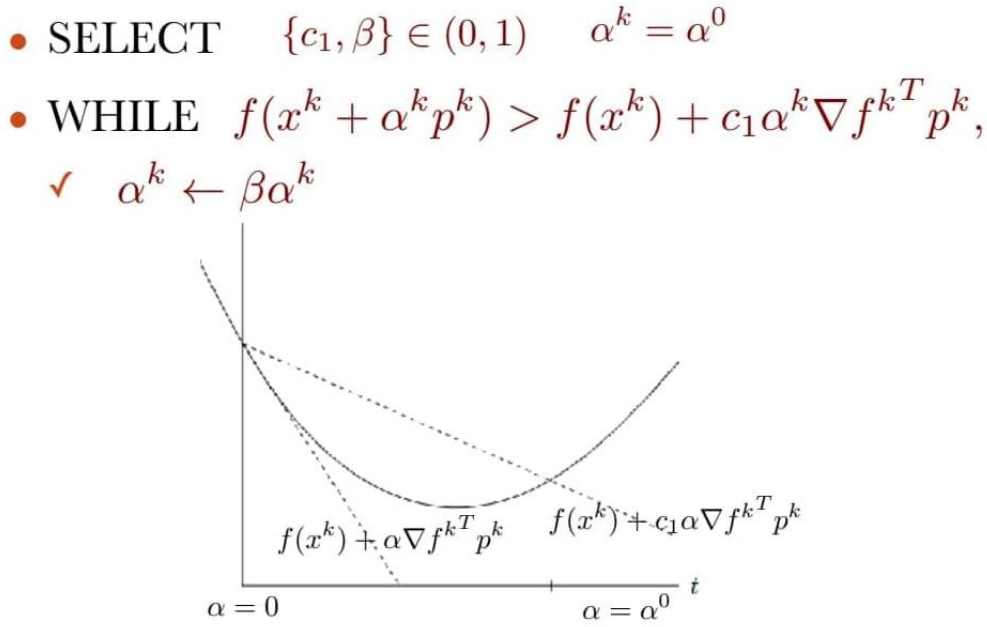
کد تایپی به شرح زیر است:

```
fun = @(a) (15*a)^2 - 10*(10*a)*cos(0.2*pi*15*a) + (10*a)^2 - 15*15*a*cos(0.4*pi*10*a);
alpha0 = fminbnd(fun,0,1)
vector0=[0,0];
vector1 = [0,0] - 1*alpha0*gradient_func(vector0)

grad2=gradient_func(vector1)
fun1 = @(a) (7.8+30.9*a)^2 - 10*(5.2-46.4*a)*cos(0.2*pi*(7.8+30.9*a)) + (5.2-46.4*a)^2 -
15*(7.8+30.9*a)*cos(0.4*pi*((5.2-46.4*a)));
alpha1 = fminbnd(fun1,0,1)


function grad=gradient_func(X)
   grad = [2*X(1)+2*pi*X(2)*sin(0.2*pi*X(1))-
15*cos(0.4*pi*X(2)),2*X(2)+6*pi*X(1)*sin(0.4*pi*X(2))-10*cos(0.2*pi*X(1))];
end
```

نتایج روش تحلیلی با پیاده سازی کامپیوتری در دو مرحله به شرح زیر است:

```
alpha0 =

   0.5208

vector1 =

   7.8115   5.2076

grad2 =

  -30.9653   46.4567

alpha1 =

   0.2196
```

در روش آرمیجو به کمک روابطی که در شکل۳-۲ آمده است، پیاده سازی کامپیوتری صورت گرفته است.



شکل ۳-۲ توضیح تئوریک روش آرمیجو

پیاده سازی کامپیوتری این قسمت در پوشه codes در فایلی به نام Q2_2_2.m آمده است و در شکل ۴-۲ قابل رویت است.

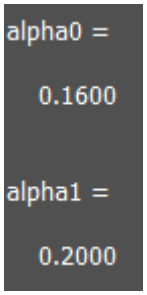
```
alpha0 = armijo([0,0],0.4,0.05,0.4)
gradient_descent([0,0]);


function alpha=armijo(X,init_alpha,c1,beta)
    alpha = init_alpha;
    while (main_func(X+alpha*gradient_func(X))) > (main_func(X) + c1*alpha*gradient_func(X)*-1*gradient_func(X)')
        alpha = beta*alpha;
    end
end

function vector=gradient_descent(start)
    vector = start;
    alpha=armijo([0,0],0.5,0.05,0.4);
    iteration=2;
    while (iteration>1)
        vector = vector - 1*alpha*gradient_func(vector)
        iteration=iteration-1;
        alpha=armijo(vector,alpha,0.05,0.4);
    end
    alpha1=alpha
end

function grad=gradient_func(X)
    grad = [2*X(1)+2*pi*X(2)*sin(0.2*pi*X(1))-15*cos(0.4*pi*X(2)),2*X(2)+6*pi*X(1)*sin(0.4*pi*X(2))-10*cos(0.2*pi*X(1))];
end

function func = main_func(X)
    func = X(1)^2 - 10*X(2)*cos(0.2*pi*X(1))+ X(2)^2 - 15*X(1)*cos(0.4*pi*X(2));
end
```

شکل۴-۲ قطعه کد پیاده سازی روش آرمیجو

کد به صورت تایپی به شرح زیر می‌باشد:

```
alpha0 = armijo([0,0],0.4,0.05,0.4)
gradient_descent([0,0]);


function alpha=armijo(X,init_alpha,c1,beta)
    alpha = init_alpha;
    while (main_func(X+alpha*gradient_func(X))) > (main_func(X) + c1*alpha*gradient_func(X)*-
1*gradient_func(X)')
        alpha = beta*alpha;
    end
end

function vector=gradient_descent(start)
    vector = start;
    alpha=armijo([0,0],0.5,0.05,0.4);
    iteration=2;
    while (iteration>1)
        vector = vector - 1*alpha*gradient_func(vector)
        iteration=iteration-1;
        alpha=armijo(vector,alpha,0.05,0.4);
    end
    alpha1=alpha
end

function grad=gradient_func(X)
    grad = [2*X(1)+2*pi*X(2)*sin(0.2*pi*X(1))-
15*cos(0.4*pi*X(2)),2*X(2)+6*pi*X(1)*sin(0.4*pi*X(2))-10*cos(0.2*pi*X(1))];
end

function func = main_func(X)
    func = X(1)^2 - 10*X(2)*cos(0.2*pi*X(1))+ X(2)^2 - 15*X(1)*cos(0.4*pi*X(2));
end
```

نتایج پیاده سازی کامپیوتری با روش آرمیجو در دو مرحله به شرح زیر است:



همانطور که مشاهده میشود نتایج با روش تحلیلی همخوانی دارد و هرچه مراحل بیشتر میشود مقدار طول بهینه به مقدار تحلیلی نزدیک تر میشود.

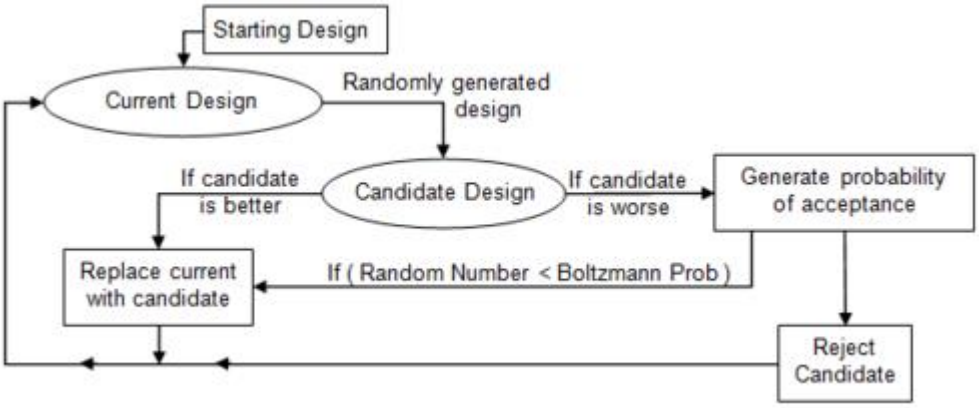
ج:

در این بخش قصد داریم روش تبرید شبیه سازی شده را پیاده سازی کنیم. این روش پیشینه فیزیکی دارد.

ماهیت فیزیکی به این صورت است که برای رسیدن به صحیح ترین ساختاری اتمی در فلزات ابتدا دمای فلزات را بالا میبرند و بعد به آرامی فلز را خنک میکنند. در این روند اتم ها به بهترین ساختار ممکن میل پیدا خواهند کرد. واکنش اتم ها اینگونه است که یا به ساختاری با آنتروپی پایین تر میروند و یا طبق رابطه بلتزمن در هر مرحله با یک احتمالی به ساختاری میروند که میتوان از آن ساختار در مرحله بعدی به ساختاری با آنتروپی پایین تر رفت. رابطه بلتزمن به صورت زیر است:

$$P = \exp\left(\frac{-\Delta E}{k_b T}\right)$$

حال این روند فیزیکی در ریاضیات الگوریتمی خواهد داشت که به روش تبرید شبیه سازی شده معروف است. روند کلی الگوریتم در فلوچارت ۵-۲ آمده است.



شکل ۵-۲ فلوچارت کلی الگوریتم تبرید شبیه سازی شده

پیاده سازی این بخش(الگوریتم تبرید شبیه سازی شده) در فایل Q2_3.ipynp آمده است.

قطعه کد مذکور به همراه نتیجه همگرایی در شکل۶-۲ آمده است.

```
T0 = 50
Lim = [-15, 15]
alpha = 7
itera = 1000
point = np.random.rand(1,2)*(Lim[1] - Lim[0]) + np.ones(2)*Lim[0]
pres = point
pt_func = main_func(point[0])
pres_func = pt_func
print ("first_value:", pt_func)

for i in range(itera):
    prob_next= alpha*np.random.randn(1,2) + pres
    prob_nt_func = main_func(prob_next[0])
    D = prob_nt_func - pres_func
    T = T0/float(i+1)
    probab = np.exp(-D/T)
    if (D<0) or (probab > np.random.rand()):
        pres = prob_next
        pres_func = prob_nt_func
    if prob_nt_func < pt_func:
        point = prob_next
        pt_func = prob_nt_func
print ("Optimom_value", pt_func)
```

```
first_value: 48.15101201256701
Optimom value -70.94097107666408
```

شکل ۶-۲ قطعه کد الگوریتم تبرید شبیه سازی شده برای تابع سوال سوم

قطعه کد فوق به صورت تایپی به صورت زیر است:

```
T0 = 50
Lim = [-15, 15]
alpha = 7
itera = 1000
point = np.random.rand(1,2)*(Lim[1] - Lim[0]) + np.ones(2)*Lim[0]
pres = point
pt_func = main_func(point[0])
pres_func = pt_func
print ("first_value:", pt_func)


for i in range(itera):
   prob_next= alpha*np.random.randn(1,2) + pres
   prob_nt_func = main_func(prob_next[0])
```

```
        D = prob_nt_func - pres_func
        T = T0/float(i+1)
        probab = np.exp(-D/T)
        if (D<0) or (probab > np.random.rand()):
            pres = prob_next
            pres_func = prob_nt_func
        if prob_nt_func < pt_func:
            point = prob_next
            pt_func = prob_nt_func
    print ("Optimom_value", pt_func)
```

نکته حائز اهمیتی که باید توضیح داده شود، مولفه های استفاده شده در الگوریتم فوق میباشد.

برای T که نمایانگر ریاضیاتی خاصیت فیزیکی است باید یک مقدار دمایی بالا مثل ۶۰ در نظر گرفت. به طور کلی بنده با T های کوچک شروع به اجرای کد کردم و به مرور رینج مناسب برای دما را پیدا کردم.

برای Lim بازه مناسب توسط صورت سوال گفته شده که در بازه (-15,15) میباشد.

کمیت probab دقیقا نمایانگر رابطه بولتزمن است (که بالاتر به آن اشاره شد است) و اکپونشنیال اختلاف D تقسیم بر دمای فعلیست.

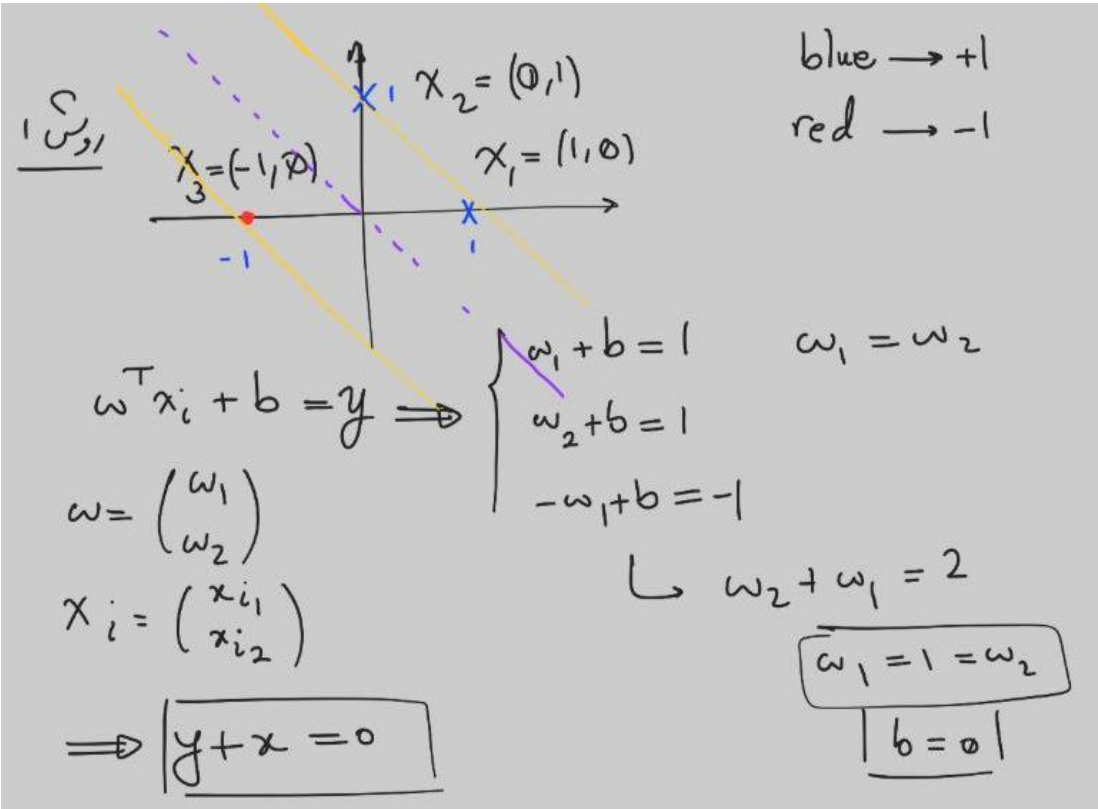
شرط تغییر نقطه این است که یا به نقطه بهتری برویم و یا احتمال اینکه با رفتن به یک نقطه بالاتر در مرحله نقطه مینیمم تری پیدا شود وجود داشته باشد.

سوال ۳

الف:

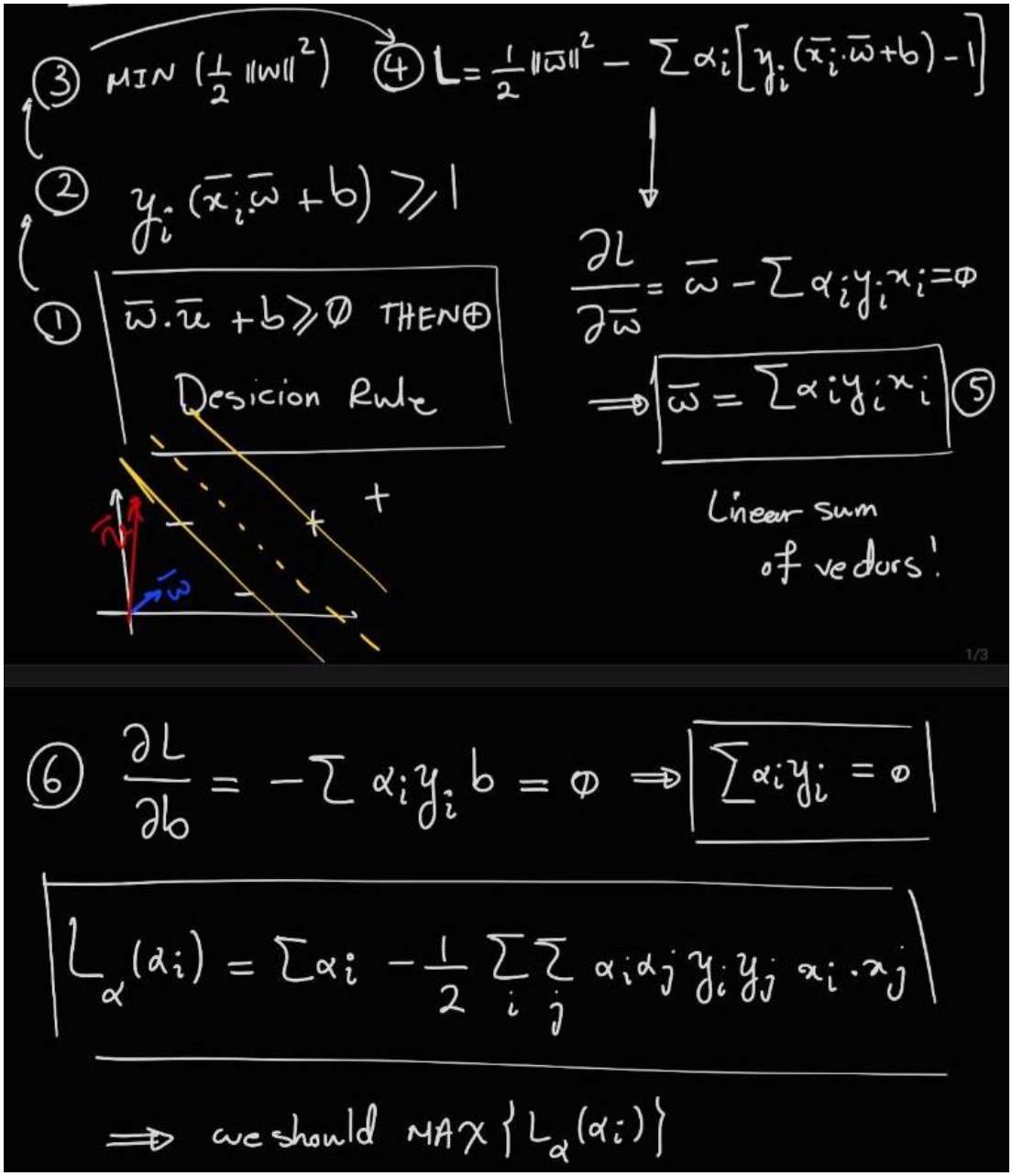
در این بخش ابتدا سعی بر نوشتن معادلات SVM (ماشین بردار پشتیبان) برای سه نقطه صورت سوال داریم. دو روش مد نظر داریم که به ترتیب روش استفاده از بردار پشتیبان و خلاصه شده مساله بهینه ساز لاگرانژ میباشد.

در شکل ۱-۳ روش استفاده از بردار پشتیبان آمده است. همانطور که مشاهده میشود هر سه نقطه به عنوان بردار پشتیبان درنظر گرفته شده است!

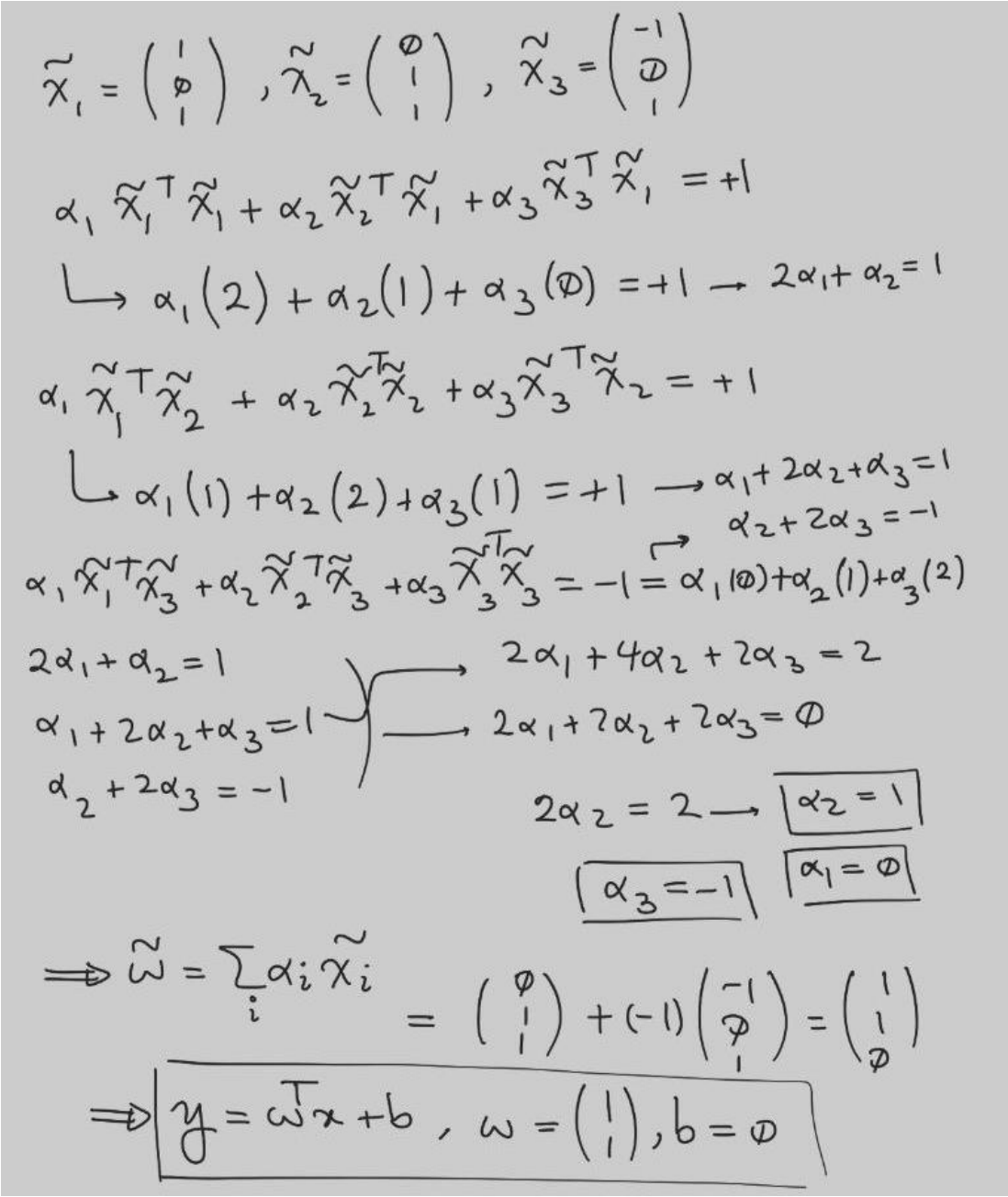


شکل۳-۱ روش استفاده از بردار های پشتیبان برای SVM

در شکل ۳-۳ روش استفاده از خلاصه شده تابع لاگرانژ به صورت خلاصه شده است. روند یافتن تابع لاگرانژ نهایی با برابر گذاشتن دیفرانسیل ها نسبت به متغیر های متفاوت مطابق شکل ۲-۳ است.



شکل ۲-۳ روند نهایی یافتن تابع لاگرانژ



شکل ۳-۳ روش محاسبه ماشین بردار پشتیبان به کمک خلاصه شده تابع لاگرانژ

ب:

در این بخش قصد داریم یک بار با استفاده از کتابخانه و بار دیگر بدون از استفاده از کتابخانه های موجود الگوریتم SVM را که به صورت One vs All پیاده سازی کنیم.

۱) پیاده سازی به کمک کتابخانه sklearn

در شکل ۳-۴ میتوان قطعه کد به همراه نتایج طبقه بند در قیاس با بقیه را مشاهده کنید که به پیوست در پوشه codes با نام Q3_2_1.ipynb ضمیمه شده است.

```python
import numpy as np
from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import SVC
from sklearn import datasets
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix

iris = datasets.load_iris()
data = iris.data[:, :2] #data
#print (data)
#print (label)
label = iris.target #label
clf = OneVsRestClassifier(SVC()).fit(data, label)
Y_Pred=clf.predict(data);
CM=confusion_matrix(label, Y_Pred)
print ("Confusion_Matrix = \n " ,CM)
print ("Confidence_Matrix = \n", CM/CM.max(axis=1))
print ("Accuarcy =" ,accuracy_score(label, Y_Pred))

#print (clf.get_params(deep=True))
#a = clf.score(data, label, clf.get_params(deep=True))
```

```
Confusion_Matrix =
  [[50  0  0]
 [ 0 37 13]
 [ 0 15 35]]
Confidence_Matrix =
 [[1.         0.         0.        ]
 [0.         1.         0.37142857]
 [0.         0.40540541 1.        ]]
Accuarcy = 0.8133333333333334
```

شکل ۴-۳ قطعه کدِ پیاده سازی طبقه بندِ در قیاس با بقیه به کمک کتابخانه به همراه نتایج

کد به صورت تایپی به شرح زیر است:

import numpy as np

from sklearn.multiclass import OneVsRestClassifier

from sklearn.svm import SVC

from sklearn import datasets

from sklearn.metrics import accuracy_score

from sklearn.metrics import confusion_matrix

iris = datasets.load_iris()

data = iris.data[:, :2] #data

```
#print (data)
#print (label)
label = iris.target #label
clf = OneVsRestClassifier(SVC()).fit(data, label)
Y_Pred=clf.predict(data);
CM=confusion_matrix(label, Y_Pred)
print ("Confusion_Matrix = \n " ,CM)
print ("Confidence_Matrix = \n", CM/CM.max(axis=1))
print ("Accuarcy =" ,accuracy_score(label, Y_Pred))
```

همانطور که مشاهده میشود نتایج پیاده سازی به شرح زیر میباشد:

دقت طبقه بند در حدود ۸۰ درصد است که دقت قابل قبولی برای پیاده سازی می باشد.

ماتریس آشفتگی به صورت $\begin{pmatrix} 50 & 0 & 0 \\ 0 & 37 & 13 \\ 0 & 15 & 35 \end{pmatrix}$ میباشد. این ماتریس نشان میدهد که کلاس با برچسب "0" به صورت کامل و دقت ۱۰۰ درصد تشخیص داده شده است. کلاس "1" اما از ۵۲ مورد، ۳۷ مورد درست تشخیص داده شده است و ۱۵ مورد به اشتباه به جای کلاس "1" کلاس "2" تشخیص داده شده است. در مورد کلاس "2" از مجموع ۴۸ موردی که در واقع کلاس "2" بوده‌اند ۳۵ مورد صحیح تشخیص داده شده‌اند و ۱۳ مورد به اشتباه به جای کلاس "2" کلاس "1" تشخیص داده شده‌اند.

در مورد ماتریس اطمینان هم از آنجا که دقیقا نرمال شده ستونی ماتریس آشفتگیست میتوان درصد تشخیص صحیح هر دسته و درصد تشخیص اشتباه یک دسته به جای یک دسته دیگر را ارزیابی کرد. (لازم به ذکر است توضیح چیف ای تی در مورد ماتریس اطمینان اینگونه بوده است که نرمال شده ستونی ماتریس آشفتگی میشود اما بنده شک دارم که شاید باید هر عنصر ستون را به مجموع کل ستون تقسیم کنیم! درهرصورت پیاده سازی مطابق صحبت تی ای صورت گرفته است.)

۲) پیاده سازی بدون استفاده از کتابخانه (from scratch)

در این قسمت از روش گرادیان نزولی استفاده کردم. روند به این صورت است که به کمک گرادیان نزولی تابع خطای هینج را کمینه میکنیم و اینگونه وزن ها را به دست می آوریم.

اما روش بنده این بود که در هربار یک دسته را +۱ و دو دسته دیگر را -۱ در نظر بگیرم و این کار را با توجه به label هایی که دیدتابیس ما دارد، سه بار انجام دهم که متاسفانه موفق نبودم.

تنها کاری که دیگر وقت نمیشود انجام دهم و لازم به اجراست این است که دیتا ها پیش پردازش شوند و همانطور که گفتم سه دسته دیتابیس به ورودی برنامه اعمال شود تا نهایتا طبقه بند قیاس با بقیه ساخته شود.

در شکل ۵-۳ قطعه کد مذکور آمده و با نام Q3_2_2.ipynb ضمیمه شده است.

```
import numpy as np
from sklearn import datasets


iris = datasets.load_iris()
data = iris.data[:, :2] #data
#print (data)
#print (label)
label = iris.target #label

def T_svm(X, Y, epochs, learning_rate):
    #our weight vector with 3 zeros
    w = np.zeros(len(X[0]))
    #print (len(X[0]))
    print("Training Starts")
    for epoch in range(1, epochs):
        for i, x in enumerate(X):
            if (Y[i] * np.dot(X[i], w)) < 1: #first class
                w = w + learning_rate * ((X[i] * Y[i]) + (-2 * (1/epochs) * w))
            else: #second class
                w = w + learning_rate * (-2 * (1/epochs) * w)
    return w

def Pred(X, weights):
    y = np.dot(X, weights)
    return y

w = T_svm(data,label,1000,1)
p = Pred (data,w)
print (p)
```

شکل ۵-۳ قطعه کدِ پیاده سازی طبقه بندِ در قیاس با بقیه بدون کمک کتابخانه

مفهوم و پشتوانه ریاضی این کد به این شرح است که:

ابتدا ساده شده عبارت لاگرانژ را مینویسیم:

$$min_w \lambda \parallel w \parallel^2 + \sum_{i=1}^{n} (1 - y_i \langle x_i, w \rangle)_+$$

سپس با مشتق گیری نسبت به پارامتر های مختلف برای محاسبه w به رابطه زیر میرسیم:

$$w = w + \eta(y_i x_i - 2\lambda w)$$