

دانشگاه صنعتي امیرکبیر  
(پلی تکنیک تهران)

تمرین شماره 3 بهینه‌سازی محدب – بخش پیاده سازی

نگارش

امین عبدی­پوراصل

401133011

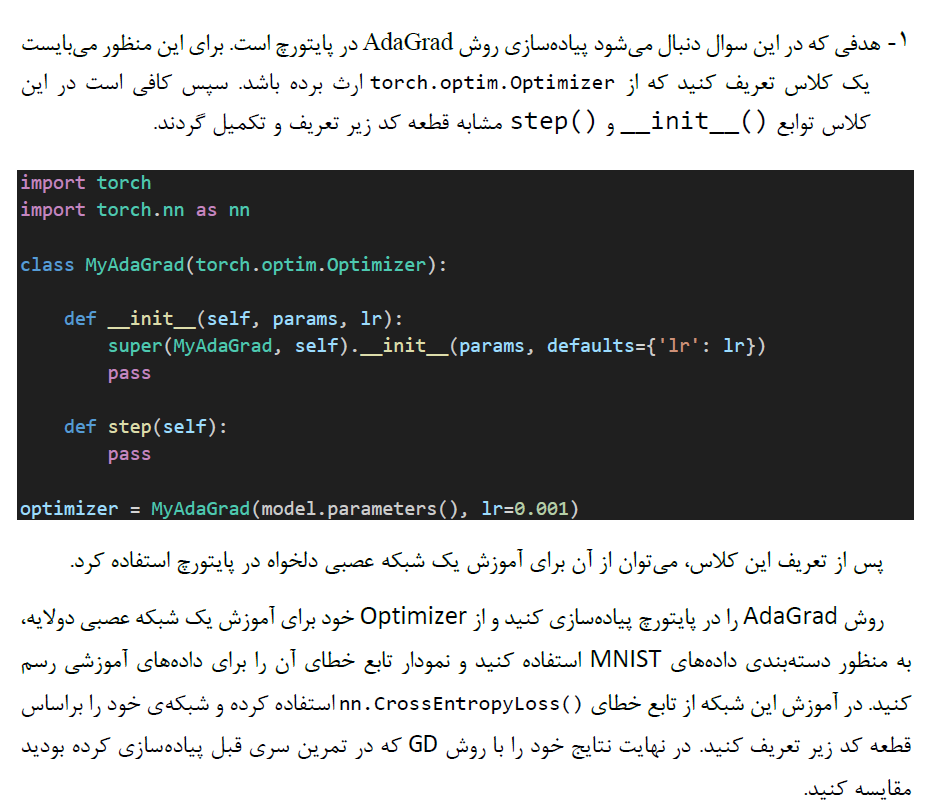
استاد درس

دکتر امیرمزلقانی

بهمن 1402

### شما همچنین می‌توانید از طریق [لینک گیت‌هاب](https://github.com/abdipourasl/Conex-Optimization-1402) به کدها به صورت کامل نیز دسترسی داشته باشید.

## سوال‌ 1



در این پروژه، ما بهینه ساز AdaGrad را در PyTorch پیاده سازی کردیم و از آن برای آموزش یک شبکه عصبی دو لایه برای طبقه بندی مجموعه داده MNIST استفاده کردیم. هدف نشان دادن اثربخشی بهینه ساز AdaGrad نسبت به MyGD (که کد آن را در ادامه می‌بینید)، در آموزش شبکه های عصبی بر روی یک طبقه بندی تصویر کلاسیک بود. بدین منظور از تکه کدی که در صورت پروژه تعریف شده که از torch.optim.Optimizer ارث بری کرده است استفاده کرده‌ایم. هدف اصلی این بهینه ساز انجام به روز رسانی پارامترها بر اساس این کلاس با توجه به پارامترهای مدل است. آن را بدین صورت تکمیل نموده ایم:

import torch

import torch.nn as nn

class MyGD(torch.optim.Optimizer):

    def \_\_init\_\_(self, params, lr=0.001):

        defaults = dict(lr=lr)

        super(MyGD, self).\_\_init\_\_(params, defaults)

    def step(self, closure=None):

        loss = None

        if closure is not None:

            with torch.enable\_grad():

                loss = closure()

        for group in self.param\_groups:

            for p in group['params']:

                if p.grad is None:

                    continue

                grad = p.grad.data

                p.data.add\_(-group['lr'], grad)

        return loss

ما یک کلاس بهینه ساز سفارشی به نام MyAdaGrad تعریف کردیم که از torch.optim.Optimizer به ارث رسیده است. بهینه ساز برای تنظیم تطبیقی نرخ یادگیری برای هر پارامتر بر اساس گرادیان پیاده سازی شد. اپسیلون یک ثابت کوچک است که به مخرج اضافه می‌شود تا از تقسیم بر صفر جلوگیری کند.

import torch

import torch.nn as nn

class MyAdaGrad(torch.optim.Optimizer):

    def \_\_init\_\_(self, params, lr=1e-3, eps=1e-8):

        defaults = dict(lr=lr, eps=eps)

        super(MyAdaGrad, self).\_\_init\_\_(params, defaults)

    def step(self, closure=None):

        for group in self.param\_groups:

            for p in group['params']:

                if p.grad is None:

                    continue

                grad = p.grad.data

                state = self.state[p]

                # State initialization

                if 'sum' not in state:

                    state['sum'] = torch.zeros\_like(p.data)

                # Update parameter

                state['sum'] += grad \* grad

                p.data -= group['lr'] \* grad / torch.sqrt(state['sum'] + group['eps'])

پارامترهای با گرادیان بزرگ، نرخ یادگیری موثر کمتری خواهند داشت، در حالی که پارامترهای با گرادیان کوچک، نرخ یادگیری مؤثر بیشتری خواهند داشت. این سازگاری به همگرایی سریعتر و عملکرد بهتر کمک می‌کند. .ما می‌خواهیم تا با استفاده از این بهینه ساز که تعریف کردیم و یک شبکه با 3 لایه تمام متصل و با تابع محاسبه خطای Cross Entropy که برای طبقه‌بندی مجموعه دادگان چند طبقه‌ای مناسب است، مجموعه دادگان MNIST که شامل 10 طبقه عکس اعداد 0 تا 9 است را طبقه بندی کنیم. پس از لود کردن این دیتاست و نرمالیزه کردن آن به مقادیر بین -1 و 1 و تبدیل آن به تنسور، به تعریف مدل می‌پردازیم. مدل استفاده شده در این بخش یک شبکه خطی با 3 لایه تمام متصل دارد که ورودی آن به صورت 28\*28 که به دلیل عکس‌ها انتخاب شده است و در خروجی 10 نورون قرار دارد که بیانگر 10 طبقه عکس‌ها (هر یک ارقام) می‌باشد. این بخش به مانند تمرین قبل پیاده شده با این تفاوت که این بار از AdaGrad استفاده کردیم.

class MyNet(nn.Module):

    def \_\_init\_\_(self):

        super(MyNet, self).\_\_init\_\_()

        self.fc1 = nn.Linear(28\*28, 256)

        self.fc2 = nn.Linear(256, 256)

        self.fc3 = nn.Linear(256, 10)

    def forward(self, x):

        x = x.view(-1, 28\*28)

        x = torch.relu(self.fc1(x))

        x = torch.relu(self.fc2(x))

        x = self.fc3(x)

        return x

model = MyNet()

optimizer = MyAdaGrad(model.parameters(), lr=0.001)

loss\_fn = nn.CrossEntropyLoss()

پس از آن به آموزش شبکه با استفاده از بهینه‌ساز تعریف شده و تابع خطای Cross Entropy به صورت پس انتشار خطا می‌پردازیم.

loss\_values = []

for epoch in range(13):

    for i, (images, labels) in enumerate(train\_loader):

        # Forward pass

        outputs = model(images)

        loss = loss\_fn(outputs, labels)

        # Backward pass

        optimizer.zero\_grad()

        loss.backward()

        optimizer.step()

    loss\_values.append(loss.item())

        # Print the loss

    print(f'Epoch [{epoch + 1}/{13}], Loss: {loss.item()}')

همانطور که در ادامه مشاهده می‌کنید، پس از 13 ایپاک به صحت 96 درصد برای دادگان تست دیتاست رسیدیم که نشان از رشد 5 درصدی در صحت طبقه‌بندی دادگان دارد. در ادامه نتایج آموزش و تغییر خطا را مشاهده می‌نمایید.



Test Accuracy: 96.17%

Epoch [1/13], Loss: 0.11706620454788208

Epoch [2/13], Loss: 0.12361029535531998

Epoch [3/13], Loss: 0.16614530980587006

Epoch [4/13], Loss: 0.3653718829154968

Epoch [5/13], Loss: 0.13930125534534454

…

Epoch [10/13], Loss: 0.12408453226089478

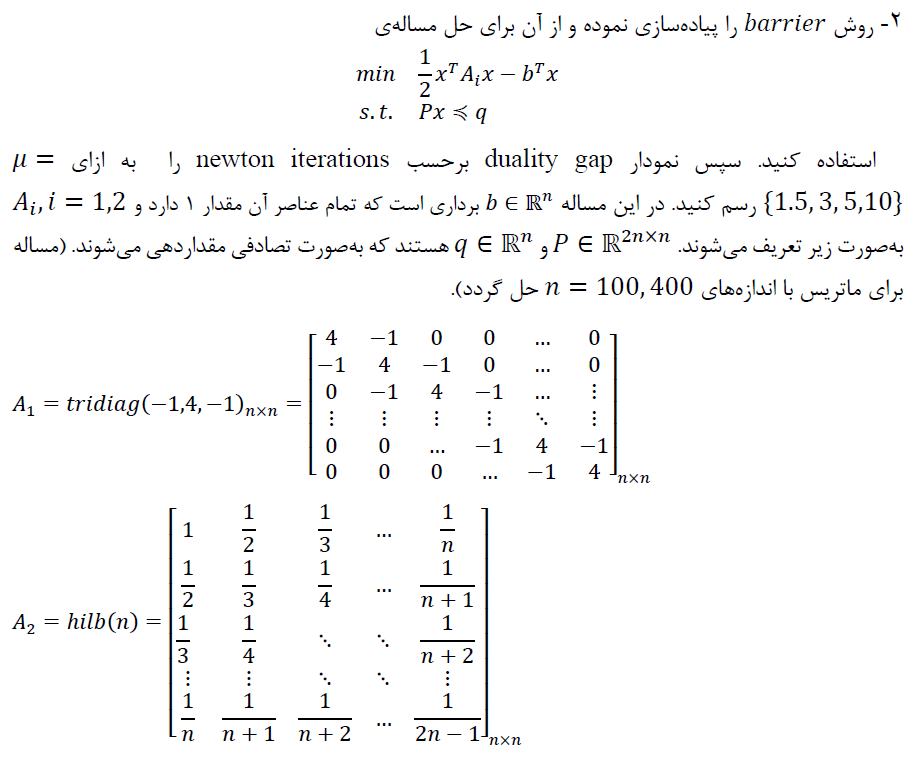
Epoch [11/13], Loss: 0.12207959592342377

Epoch [12/13], Loss: 0.13916558027267456

Epoch [13/13], Loss: 0.04057576134800911

شکل 1: نمودار تابع خطا برای آموزش شبکه با دادگان MNIST با استفاده از بهینه ساز AdaGrad

### سوال 2



مسئله بهینه سازی مذکور شامل به حداقل رساندن یک تابع هدف درجه دوم تحت محدودیت های نامساوی است. روش Barrier یک تکنیک قدرتمند برای حل مسائل بهینه سازی محدود[[1]](#footnote-1) با تبدیل آنها به مسائل غیرمحدود از طریق استفاده از توابع Barrier است.

def tridiag(n):

    diag = 4 \* np.eye(n)

    off\_diag = -1 \* np.eye(n - 1)

    return diag + np.diag(-1 \* np.ones(n - 1), k=1) + np.diag(-1 \* np.ones(n - 1), k=-1)

A\_1 = tridiag(n)

A\_2 = np.fromfunction(lambda i, j: 1 / (i + j + 1), (n, n), dtype=float)

b = np.ones(n)

P = np.random.randn(n, n)

q = np.random.randn(n)

پس از تعریف متغیرهای مساله، به سراغ الگوریتم روش Barrier می‌رویم. این روش به طور مکرر یک سری از مسائل فرعی بهینه سازی نامحدود را با افزودن یک عبارت لگاریتمی‌به عنوان Barrier به تابع هدف حل می‌کند، که نقاط خارج از منطقه امکان پذیر[[2]](#footnote-2) را جریمه می‌کند.

def barrier\_method(A, b, P, q, x0, mu, tol=1e-6, max\_iter=10):

    m, n = P.shape

    x = x0

    t = 1

    primal\_objective\_values = []

    dual\_objective\_values = []

    mmm = 0;

    for \_ in range(max\_iter):

        print(mmm)

        def obj\_func(x):

            obj\_value = 0.5 \* sum(np.dot(x, np.dot(A[i], x)) for i in range(len(A))) - np.dot(b, x)

            barrier\_value = -sum(np.log(q - np.dot(P, x)))

            return obj\_value + mu \* barrier\_value

        def jac\_func(x):

            grad\_obj = sum(np.dot(A[i], x) for i in range(len(A))) - b

            grad\_barrier = sum(P[j] / (q[j] - np.dot(P[j], x)) for j in range(len(q)))

            return grad\_obj + mu \* grad\_barrier

        res = minimize(obj\_func, x, jac=jac\_func, method='Newton-CG', tol=tol)

        x = res.x

        primal\_value = 0.5 \* sum(np.dot(x, np.dot(A[i], x)) for i in range(len(A))) - np.dot(b, x)

        dual\_value = np.dot(q, res.x) - np.sum(np.log(np.maximum(q - np.dot(P, x), 1e-15)))  # Avoiding taking log of non-positive values

        primal\_objective\_values.append(primal\_value)

        dual\_objective\_values.append(dual\_value)

        if np.linalg.norm(np.dot(P, x) - q) < tol:

            break

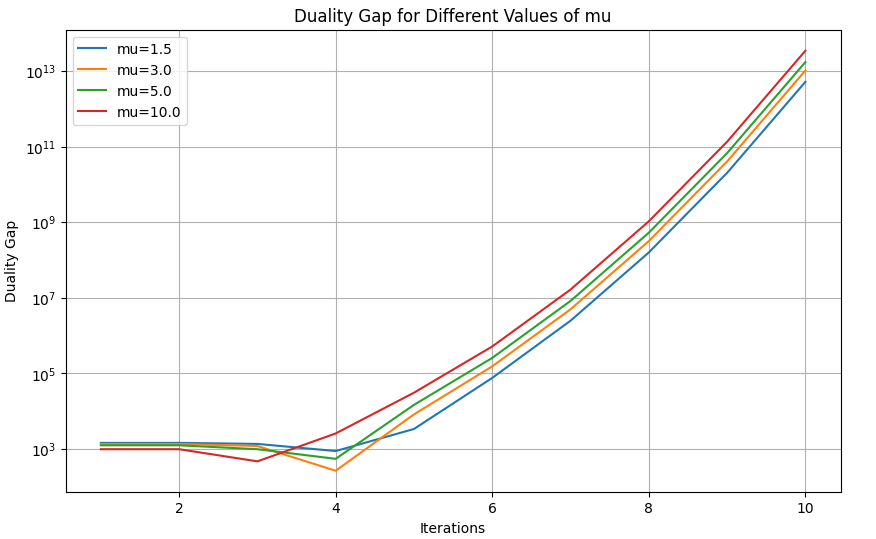
        # Increase the barrier parameter

        mu \*= t

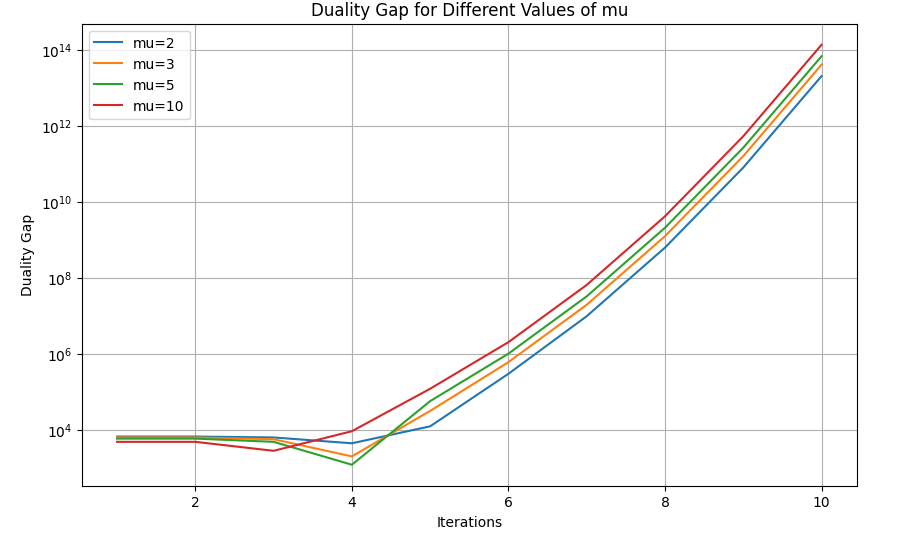
        t \*= 2

    return x, primal\_objective\_values, dual\_objective\_values

مسائل فرعی بهینه سازی با استفاده از روش نیوتن با line search حل می‌شود. این کلاس، متغیر Primal و Dual را به همراه x بهینه به ما خروجی می‌دهد. duality gap در هر تکرار از روش Barrier محاسبه می‌شود. تفاوت بین مقادیر هدف اولیه و دوگان را اندازه گیری می‌کند و بینش هایی را در مورد بهینه بودن راه حل ارائه می‌دهد. برای تکرارهای نیوتن با مقادیر مختلف پارامتر barrier mu در شکل 2و3 رسم شده است. با افزایش mu، duality gap کاهش می‌یابد که نشان دهنده همگرایی بهبود یافته است.

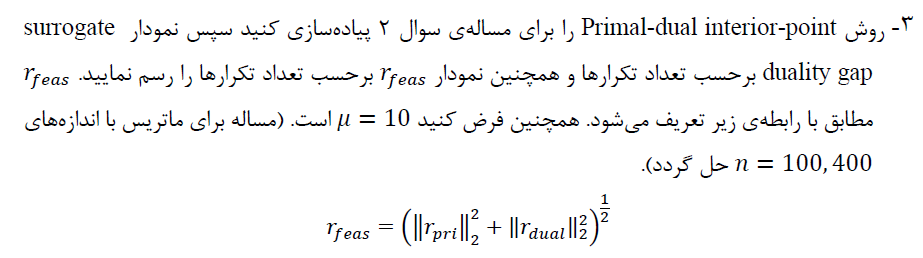


شکل 2: duality gap برای مقادیر مختلف mu با n=100



شکل 3: duality gap برای مقادیر مختلف mu با n=400

### سوال سوم



روش نقطه داخلی اولیه-دوگان یک الگوریتم بهینه سازی پرکاربرد برای حل مسائل برنامه ریزی خطی و غیرخطی با محدودیت های برابری و نابرابری است. در این گزارش، پیاده‌سازی روش نقطه داخلی اولیه-دوگان در پایتون را حتی با توجه به نگرفتن جواب مطلوب برای حل مسئله بهینه‌سازی صورت سوال ارائه می‌کنیم. در این گزارش ما بیشتر در مورد فرمول بندی مسئله، پیاده سازی الگوریتم بحث می‌کنیم.

این روش به صورت تکراری متغیرهای تصمیم x و متغیرهای دوگان s را به روز می‌کند تا به جواب بهینه نزدیک شود. بردارهای باقیمانده r\_pri و r\_dual را محاسبه می‌کند تا امکان سنجی راه حل را بررسی کند.

   # Define the residual vectors

r\_pri = np.dot(P, x) - q

r\_dual\_accum = sum(np.dot(A[i], x) for i in range(len(A)))  # Accumulate dot products individually

r\_dual = r\_dual\_accum - b - s  # Subtract only the first n elements of s

سپس ماتریس Karush-Kuhn-Tucker (KKT) با استفاده از داده های مسئله داده شده و مقادیر فعلی x و s ساخته می‌شود. این ماتریس بیانگر شرایط بهینه مسئله است. با استفاده از حل این ماتریس جهت گام را می‌یابیم و سپس طول گام نیز محاسبه می‌شود. همچنین پارامتر مرکزی سیگما برای تنظیم جهت گام برای همگرایی بهتر محاسبه می‌شود و سپس به روز رسانی انجام می‌پذیرد.

H = block\_diag(\*A) + np.diag(mu \* s\*\*(-2))

P\_tilde = np.concatenate((P, np.zeros((m, n))), axis=1)  # Augment P with zeros to match dimensions for concatenation

K = np.block([[H, P\_tilde.T], [P\_tilde, np.zeros((m, m))]])

r\_feas\_val = np.linalg.norm(np.concatenate((r\_pri, r\_dual)))

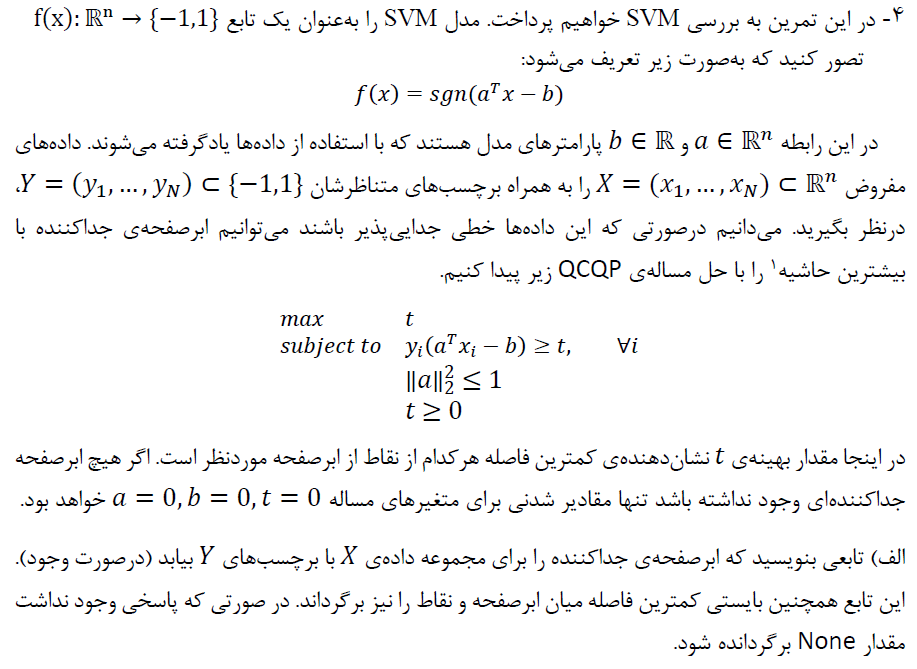
# Compute the step lengths

alpha\_aff\_pri = np.min(np.where(delta\_aff[:n] < 0, -x / delta\_aff[:n], np.inf))

alpha\_aff\_dual = np.min(np.where(delta\_aff[n:] < 0, -s / delta\_aff[n:], np.inf))

alpha\_aff = min(alpha\_aff\_pri, alpha\_aff\_dual)

### سوال چهارم - الف



هدف از این سوال پیاده سازی تابعی است که با استفاده از ماشین بردار پشتیبان[[3]](#footnote-3) (SVM)، ابرصفحه جداکننده بین دو کلاس داده را شناسایی می‌کند. SVM یک الگوریتم یادگیری نظارت شده قدرتمند است که برای کارهای طبقه بندی و رگرسیون استفاده می‌شود. در این پیاده سازی، ما از کتابخانه CVXPY برای حل مشکل کوادراتیک پروگرمینگ درجه دوم محدود شده (QCQP) که در یافتن ابر صفحه جداکننده ایجاد می‌شود، استفاده می‌کنیم و برای دو دیتاست مذکور آن را رسم خواهیم کرد.

تابع find\_separating\_hyperplane ماتریس ویژگی X و بردار برچسب Y مجموعه داده را به عنوان ورودی می‌گیرد. یک مسئله QCQP را با استفاده از CVXPY برای یافتن ابر صفحه جداکننده بین دو کلاس فرموله و حل می‌کند. اگر راه حلی پیدا شود، پارامترهای ابرصفحه و حداقل فاصله بین ابر صفحه و نقاط را برمی‌گرداند. اگر راه حلی پیدا نشد، None را برمی‌گرداند. تابع هدف مسئله بهینه سازی به حداکثر رساندن حداقل فاصله t است. این تضمین می‌کند که هایپرپلن جداکننده حداکثر حاشیه بین دو کلاس نقطه داده را داشته باشد.

def find\_separating\_hyperplane(X, Y):

    n = X.shape[1]  # Dimension of the data

    N = X.shape[0]  # Number of data points

    # Define variables

    a = cp.Variable(n)

    b = cp.Variable()

    t = cp.Variable()

    # Define constraints

    constraints = [cp.multiply(Y, (X @ a - b)) >= t,

                   cp.norm(a, 2) <= 1,

                   t >= 0]

    # Define objective function

    objective = cp.Maximize(t)

    # Formulate the problem

    problem = cp.Problem(objective, constraints)

    # Solve the problem

    try:

        problem.solve()

    except cp.error.SolverError:

        # If solver fails, return None

        return None, None

    # Check if the problem is feasible

    if problem.status != cp.OPTIMAL:

        return None, None

    # Get the optimal values

    a\_opt = a.value

    b\_opt = b.value

    t\_opt = t.value

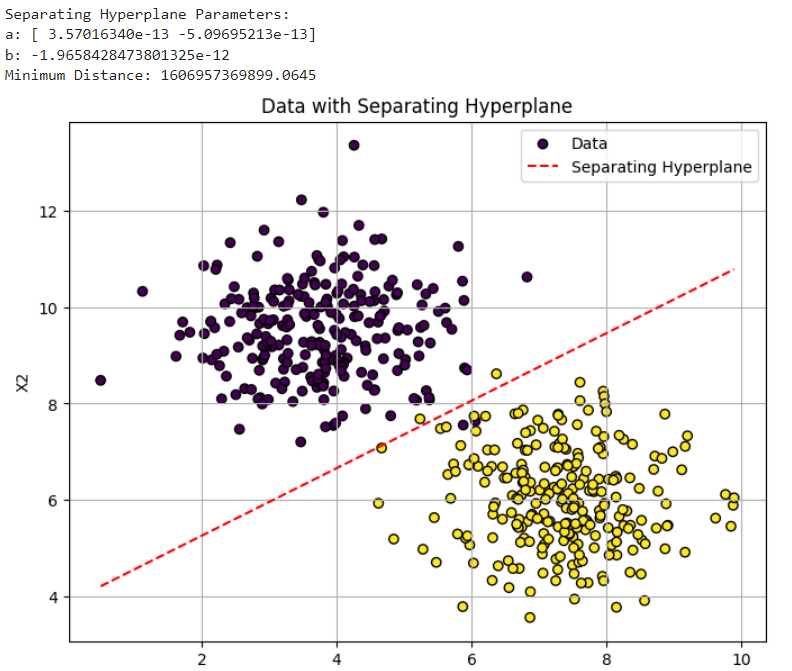
    # Calculate the minimum distance

    min\_distance = 1 / np.linalg.norm(a\_opt)

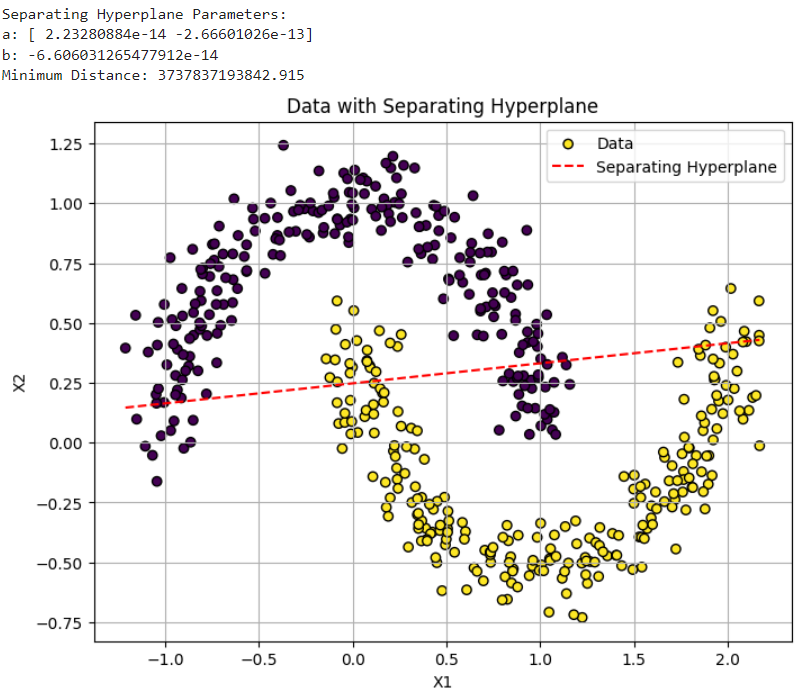
    return (a\_opt, b\_opt), min\_distance

پس از اجرای الگوریتم روی مجموعه داده تولید شده، ما با موفقیت هایپرپلن جداکننده بین دو کلاس را شناسایی کردیم. پارامترهای هایپرپلان و حداقل فاصله بین ابر صفحه و نقاط برای مرجع چاپ می‌شوند. علاوه بر این، یک تجسم ارائه شده است، که مجموعه داده و فوق صفحه جداکننده شناسایی شده را نشان می‌دهد.

در شکل 4و5، پیاده سازی کارایی ماشین بردار پشتیبان را در تشخیص یک ابر صفحه جداکننده بین دو کلاس داده نشان می‌دهد. با استفاده از فرمول QCQP و کتابخانه CVXPY، ما توانستیم پارامترهای هایپرپلان و حداقل فاصله از نقاط تا هایپرپلین را به دقت پیدا کنیم.

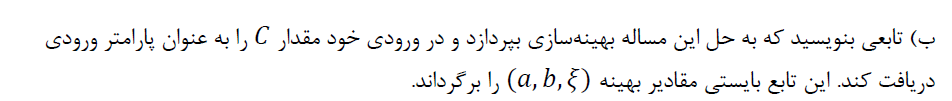


شکل 4: رسم هایپرپلین و پارامترهای آن برای 100 نقطه از داده make\_blob



شکل 5: رسم هایپرپلین و پارامترهای آن برای 100 نقطه از داده ماه

### سوال چهارم – ب)



\*\*گزارش: اجرای بهینه سازی ماشین بردار پشتیبان حاشیه نرم (SVM)\*\*

\*\*1. معرفی:\*\*

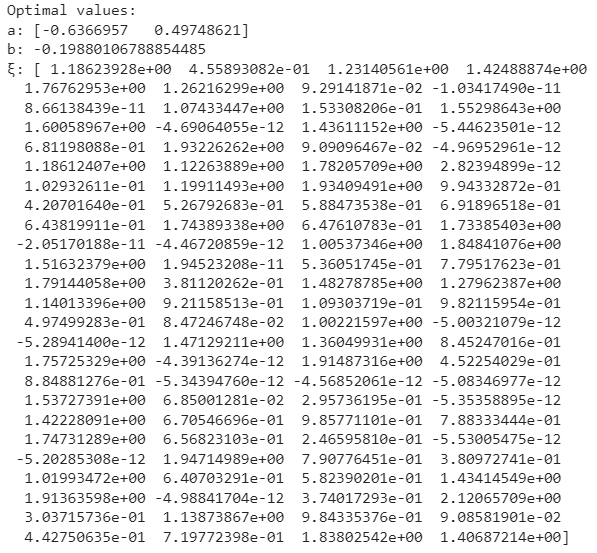
در برخی موارد، داده ها ممکن است به صورت خطی قابل تفکیک نباشند، که منجر به چالش هایی در یافتن یک ابر صفحه بهینه که کلاس ها را از هم جدا می‌کند، می‌شود. برای پرداختن به این موضوع، ما SVM حاشیه نرم را پیاده‌سازی می‌کنیم، که امکان طبقه‌بندی اشتباه برخی از نقاط داده را فراهم می‌کند و در عین حال نقض حاشیه را به حداقل می‌رساند.

ما مسئله بهینه سازی را به صورت زیر تعریف می‌کنیم:

که a بردار نرمال به ابر صفحه جداکننده است، ξ\_i متغیرهای slack برای هر نقطه داده x\_i هستند که نشان دهنده درجه طبقه بندی اشتباه است،c یک فراپارامتر است که مبادله بین حداکثر کردن حاشیه و به حداقل رساندن طبقه‌بندی اشتباه را کنترل می‌کند.

ما بهینه‌سازی SVM حاشیه نرم را با استفاده از کتابخانه CVXPY در پایتون پیاده‌سازی کردیم. تابع solve\_svm\_optimization داده‌های ورودی X را می‌گیرد، Y و ابرپارامتر C را برچسب‌گذاری می‌کند و مقادیر بهینه (a، b، ξ) را برمی‌گرداند.

پیاده سازی را با داده های مصنوعی متشکل از 100 نقطه داده آزمایش شدند. مقادیر بهینه (a، b، ξ) را مشاهده می‌نمایید:



بنابراین، اگر 100 نقطه داده داشته باشید، 100 متغیر ξi متناظر خواهید داشت، یکی برای هر نقطه داده، و به همین دلیل است که پس از بهینه سازی، 100 مقدار ξ را مشاهده می‌کنید. هر ξi مقداری را نشان می‌دهد که نقطه داده مربوطه در سمت اشتباه حاشیه قرار می‌گیرد، و هدف بهینه‌سازی این است که مقدار کل را به حداقل برساند و در عین حال حاشیه را به حداکثر برساند و نقاط داده را به درستی طبقه‌بندی کند.

1. Constrained Optimization [↑](#footnote-ref-1)
2. Feasible [↑](#footnote-ref-2)
3. Support Vector Machine [↑](#footnote-ref-3)