دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

تمرین شماره ۳ بهینه‌سازی محدب – بخش پیاده سازی

نگارش
امین عبدی‌پوراصل
۴۰۱۱۳۳۰۱۱

استاد درس
دکتر امیرمزلقانی

بهمن ۱۴۰۲

شما همچنین می‌توانید از طریق لینک گیت‌هاب به کدها به صورت کامل نیز دسترسی داشته باشید.

سوال ۱

۱- هدفی که در این سوال دنبال می‌شود پیاده‌سازی روش AdaGrad در پایتورچ است. برای این منظور می‌بایست یک کلاس تعریف کنید که از `torch.optim.Optimizer` ارث برده باشد. سپس کافی است در این کلاس توابع `__init__()` و `step()` مشابه قطعه کد زیر تعریف و تکمیل گردند.

```
import torch
import torch.nn as nn

class MyAdaGrad(torch.optim.Optimizer):

    def __init__(self, params, lr):
        super(MyAdaGrad, self).__init__(params, defaults={'lr': lr})
        pass

    def step(self):
        pass

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

پس از تعریف این کلاس، می‌توان از آن برای آموزش یک شبکه عصبی دلخواه در پایتورچ استفاده کرد.

روش AdaGrad را در پایتورچ پیاده‌سازی کنید و از Optimizer خود برای آموزش یک شبکه عصبی دولایه، به منظور دسته‌بندی داده‌های MNIST استفاده کنید و نمودار تابع خطای آن را برای داده‌های آموزشی رسم کنید. در آموزش این شبکه از تابع خطای `nn.CrossEntropyLoss()` استفاده کرده و شبکه‌ی خود را براساس قطعه کد زیر تعریف کنید. در نهایت نتایج خود را با روش GD که در تمرین سری قبل پیاده‌سازی کرده بودید مقایسه کنید.

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 داشـــت، 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 ۳ لایه تمام متصـــل و با تابع محاســـبه خطای Cross Entropy که برای طبقه‌بندی مجموعه دادگان چند طبقه‌ای مناسب است، مجموعه دادگان MNIST که شامل ۱۰ طبقه عکس اعداد ۰ تا ۹ است را طبقه بندی کنیم. پس از لود کردن این دیتاســـت و نرمالیزه کردن آن به مقادیر بین -۱ و ۱ و تبدیل آن به تنسور، به تعریف مدل می‌پردازیم. مدل استفاده شده در این بخش یک شبکه خطی با ۳ لایه تمام متصل دارد که ورودی آن به صـورت ۲۸*۲۸ که به دلیل عکس‌ها انتخاب شـده اسـت و در خروجی ۱۰ نورون قرار دارد که بیانگر ۱۰ طبقه عکس‌ها (هر یک ارقام) می‌باشـد. این بخش به مانند تمرین قبل پیاده شـده 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()
```

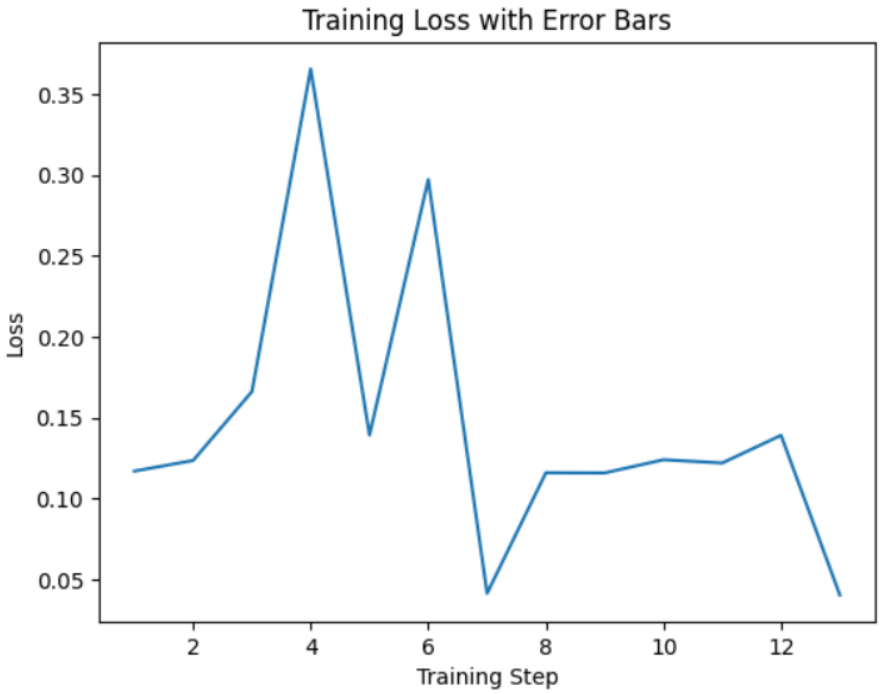
پس از آن به آموزش شبکه با استفاده از بهینه‌ساز تعریف شده و تابع خطای Cross Entropy به صورت پس انتشار خطا می‌پردازیم.

```
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 ۱۳ ایپاک به صحت ۹۶ درصد برای دادگان تست دیتاست رسیدیم که نشان از رشد ۵ درصدی در صحت طبقه‌بندی دادگان دارد. در ادامه نتایج آموزش و تغییر خطا را مشاهده می‌نمایید.

```
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
```

```
Test Accuracy: 96.17%
```



شکل ۱: نمودار تابع خطا برای آموزش شبکه با دادگان MNIST با استفاده از بهینه ساز AdaGrad

سوال ۲

۲- روش $barrier$ را پیاده‌سازی نموده و از آن برای حل مساله‌ی

$$\begin{aligned} min \quad & \frac{1}{2}x^T A_i x - b^T x \\ s.t. \quad & Px \preccurlyeq q \end{aligned}$$

استفاده کنید. سپس نمودار duality gap برحسب newton iterations را به ازای $\mu =$ $\{1.5, 3, 5, 10\}$ رسم کنید. در این مساله $b \in \mathbb{R}^n$ برداری است که تمام عناصر آن مقدار ۱ دارد و $A_i, i = 1,2$ به‌صورت زیر تعریف می‌شوند. $P \in \mathbb{R}^{2n \times n}$ و $q \in \mathbb{R}^n$ هستند که به‌صورت تصادفی مقداردهی می‌شوند. (مساله برای ماتریس با اندازه‌های $n = 100, 400$ حل گردد).

$$A_1 = tridiag(-1,4,-1)_{n\times n} = \begin{bmatrix} 4 & -1 & 0 & 0 & \dots & 0 \\ -1 & 4 & -1 & 0 & \dots & 0 \\ 0 & -1 & 4 & -1 & \dots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & -1 & 4 & -1 \\ 0 & 0 & 0 & \dots & -1 & 4 \end{bmatrix}_{n\times n}$$

$$A_2 = hilb(n) = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \dots & \frac{1}{n} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \dots & \frac{1}{n+1} \\ \frac{1}{3} & \frac{1}{4} & \ddots & \ddots & \frac{1}{n+2} \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ \frac{1}{n} & \frac{1}{n+1} & \frac{1}{n+2} & \dots & \frac{1}{2n-1} \end{bmatrix}_{n\times n}$$

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] 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)
```

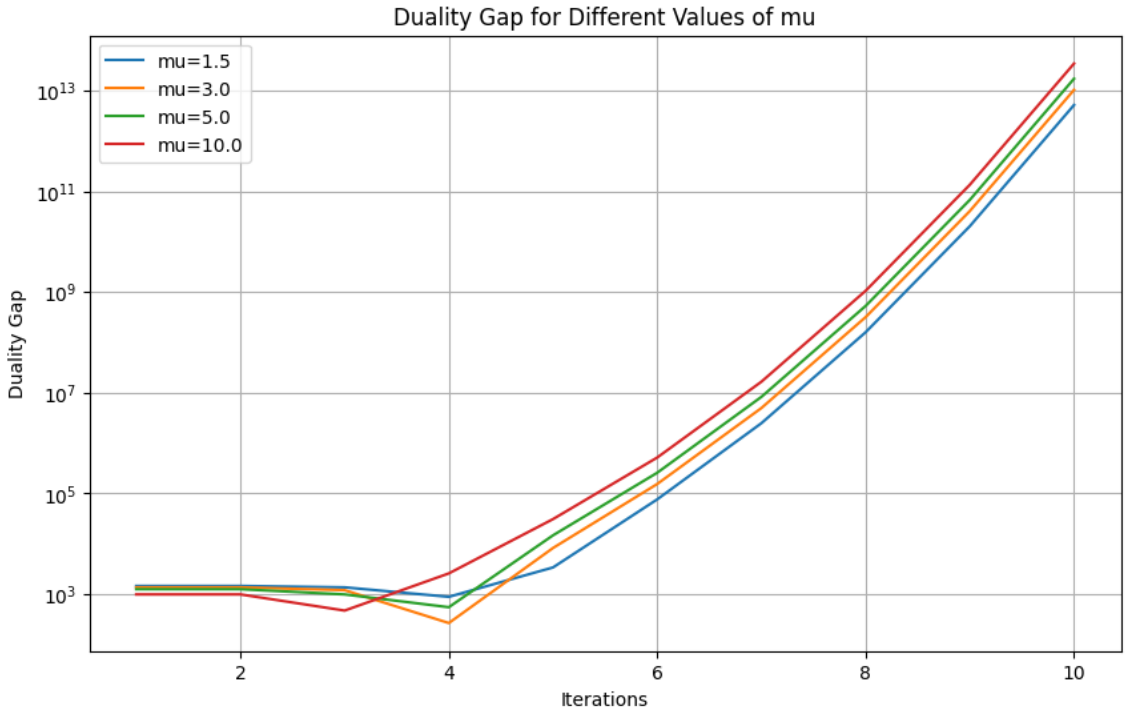
[1] Constrained Optimization

پس از تعریف متغیرهای مسـاله، به سـراغ الگوریتم روش Barrier می‌رویم. این روش به طور مکرر یک سـری از مسـائل فرعی بهینه سـازی نامحدود را با افزودن یک عبارت لگاریتمی‌به عنوان Barrier به تابع هدف حل می‌کند، که نقاط خارج از منطقه امکان پذیر[1] را جریمه می‌کند.

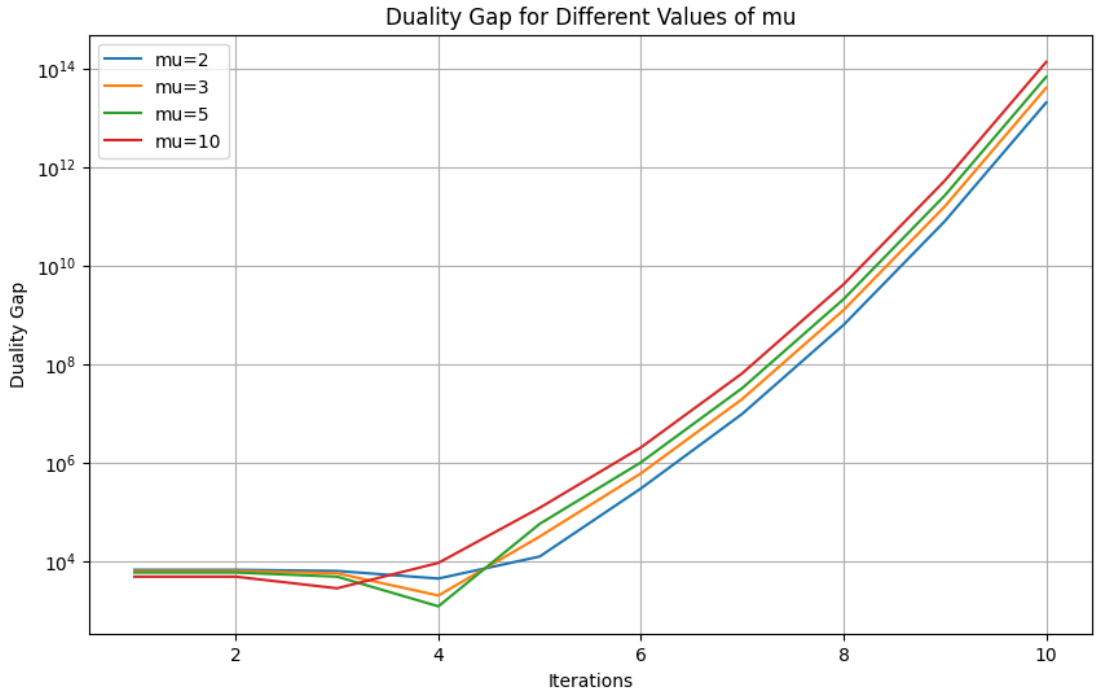
```
def barrier_method(A, b, P, q, x0, mu, tol=1e-6, max_iter=10):

    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
```

[1] Feasible

مسـائل فرعی بهینه سـازی با اسـتفاده از روش نیوتن با line search حل می‌شـود. این کلاس، متغیر Primal و Dual را به همراه x بهینه به ما خروجی می‌دهد. duality gap در هر تکرار از روش Barrier محاسـبه می‌شـود. 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 mu barrier در شـکل ۲و۳ رسم شده است. با افزایش mu، duality gap کاهش می‌یابد که نشان دهنده همگرایی بهبود یافته است.



شکل ۲: duality gap برای مقادیر مختلف mu با n=100



شکل ۳: duality gap برای مقادیر مختلف mu با n=400

سوال سوم

۳- روش Primal-dual interior-point را برای مساله‌ی سوال ۲ پیاده‌سازی کنید سپس نمودار surrogate duality gap برحسب تعداد تکرارها و همچنین نمودار r_{feas} برحسب تعداد تکرارها را رسم نمایید. r_{feas} مطابق با رابطه‌ی زیر تعریف می‌شود. همچنین فرض کنید $\mu = 10$ است. (مساله برای ماتریس با اندازه‌های $n = 100, 400$ حل گردد).

$$r_{feas} = \left(\|r_{pri}\|_2^2 + \|r_{dual}\|_2^2\right)^{\frac{1}{2}}$$

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

۴- در این تمرین به بررسی SVM خواهیم پرداخت. مدل SVM را به‌عنوان یک تابع $f(x): \mathbb{R}^n \to \{-1,1\}$ تصور کنید که به‌صورت زیر تعریف می‌شود:

$$f(x) = sgn(a^T x - b)$$

در این رابطه $a \in \mathbb{R}^n$ و $b \in \mathbb{R}$ پارامترهای مدل هستند که با استفاده از داده‌ها یادگرفته می‌شوند. داده‌های مفروض $X = (x_1, \dots, x_N) \subset \mathbb{R}^n$ را به همراه برچسب‌های متناظرشان $Y = (y_1, \dots, y_N) \subset \{-1,1\}$، درنظر بگیرید. می‌دانیم درصورتی که این داده‌ها خطی جدایی‌پذیر باشند می‌توانیم ابرصفحه‌ی جداکننده با بیشترین حاشیه[1] را با حل مساله‌ی QCQP زیر پیدا کنیم.

$$\begin{array}{ll} max & t \\ subject\ to & y_i(a^T x_i - b) \geq t, \quad \forall i \\ & \|a\|_2^2 \leq 1 \\ & t \geq 0 \end{array}$$

در اینجا مقدار بهینه‌ی t نشان‌دهنده‌ی کمترین فاصله هرکدام از نقاط از ابرصفحه موردنظر است. اگر هیچ ابرصفحه جداکننده‌ای وجود نداشته باشد تنها مقادیر شدنی برای متغیرهای مساله $a = 0, b = 0, t = 0$ خواهد بود.

الف) تابعی بنویسید که ابرصفحه‌ی جداکننده را برای مجموعه داده‌ی X با برچسب‌های Y بیابد (درصورت وجود). این تابع همچنین بایستی کمترین فاصله میان ابرصفحه و نقاط را نیز برگرداند. در صورتی که پاسخی وجود نداشت مقدار None برگردانده شود.

هدف از این سوال پیاده سازی تابعی است که با استفاده از ماشین بردار پشتیبان[1] (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

[1] Support Vector Machine

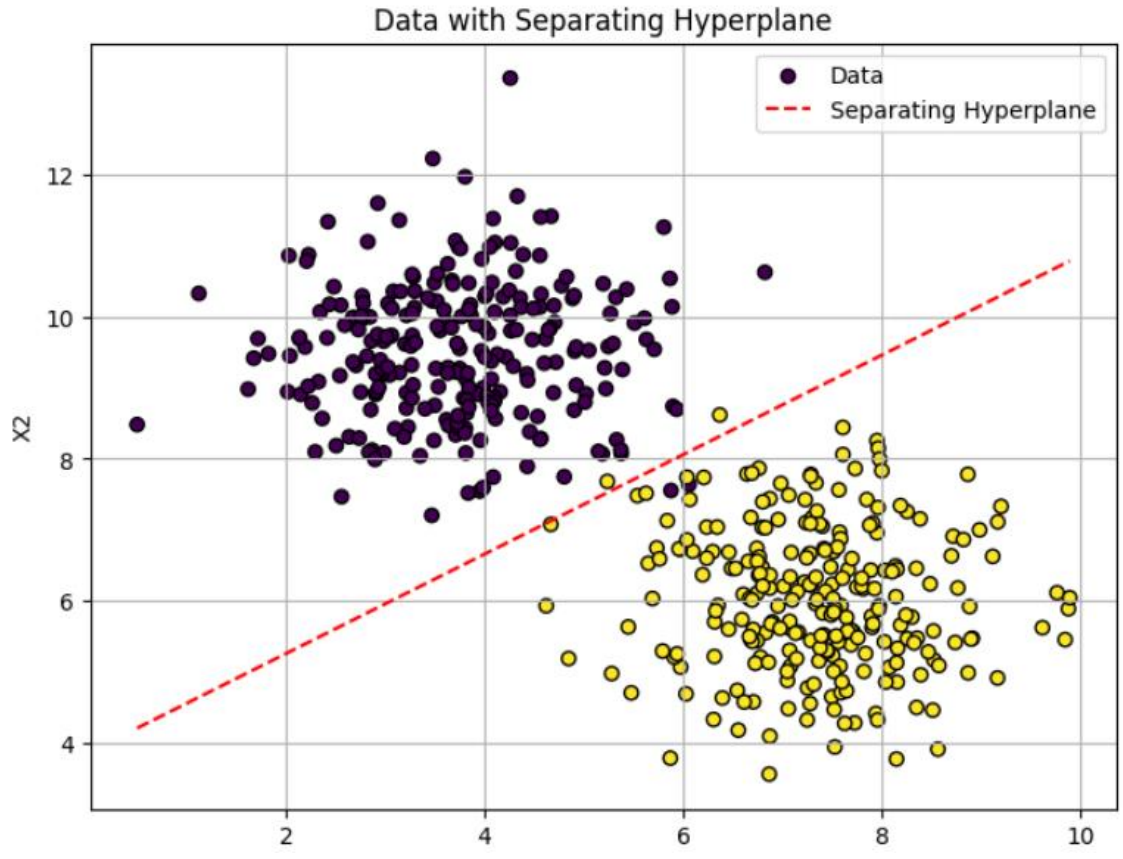
حداکثر رساندن حداقل فاصله t است. این تضمین می‌کند که هایپرپلن جداکننده حداکثر حاشیه بین دو کلاس نقطه داده را داشته باشد.

```
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 شـده، 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 تجسـم ارائه شـده اسـت، که مجموعه داده و فوق صـفحه جداکننده شـناسـایی شـده را نشان می‌دهد.

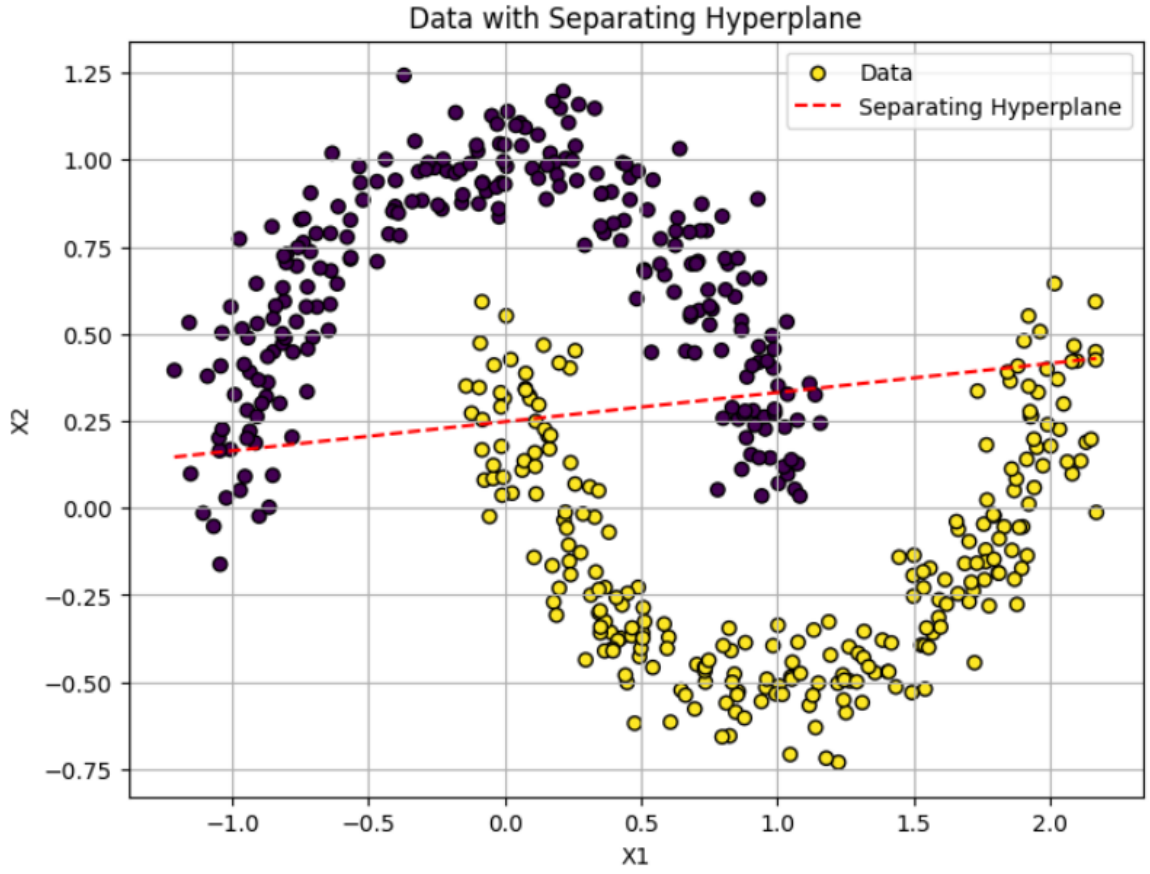
در شکل ۴و۵، پیاده سازی کارایی ماشین بردار پشتیبان را در تشخیص یک ابر صفحه جداکننده بین دو کلاس داده نشـان می‌دهد. با اسـتفاده از فرمول QCQP و کتابخانه CVXPY، ما توانسـتیم پارامترهای هایپرپلان و حداقل فاصله از نقاط تا هایپرپلین را به دقت پیدا کنیم.

```
Separating Hyperplane Parameters:
a: [ 3.57016340e-13 -5.09695213e-13]
b: -1.9658428473801325e-12
Minimum Distance: 1606957369899.0645
```



شکل ۴: رسم هایپرپلین و پارامترهای آن برای ۱۰۰ نقطه از داده make_blob

```
Separating Hyperplane Parameters:
a: [ 2.23280884e-14 -2.66601026e-13]
b: -6.606031265477912e-14
Minimum Distance: 3737837193842.915
```



شکل ۵: رسم هایپرپلین و پارامترهای آن برای ۱۰۰ نقطه از داده ماه

سوال چهارم – ب)

ب) تابعی بنویسید که به حل این مساله بهینه‌سازی بپردازد و در ورودی خود مقدار C را به عنوان پارامتر ورودی دریافت کند. این تابع بایستی مقادیر بهینه (a, b, ξ) را برگرداند.

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

۱. 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 حاشـــیه نرم را پیاده‌ســـازی می‌کنیم، که امکان طبقه‌بندی اشـــتباه 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:

$$\begin{aligned} max \quad & \frac{1}{2} \| a \|_2^2 + c \sum_{i=1}^{N} \xi_i \\ \text{subject to} \quad & y_i(a^\top x_i - b) \geq 1 - \xi_i, \forall i \\ & \xi_i \geq 0 \end{aligned}$$

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 ۱۰۰ نقطه داده آزمایش شدند. مقادیر بهینه (a، b، ξ) را مشاهده می‌نمایید:

```
Optimal values:
a: [-0.6366957   0.49748621]
b: -0.19880106788854485
ξ: [ 1.18623928e+00  4.55893082e-01  1.23140561e+00  1.42488874e+00
  1.76762953e+00  1.26216299e+00  9.29141871e-02 -1.03417490e-11
  8.66138439e-11  1.07433447e+00  1.53308206e-01  1.55298643e+00
  1.60058967e+00 -4.69064055e-12  1.43611152e+00 -5.44623501e-12
  6.81198088e-01  1.93226262e+00  9.09096467e-02 -4.96952961e-12
  1.18612407e+00  1.12263889e+00  1.78205709e+00  2.82394899e-12
  1.02932611e-01  1.19911493e+00  1.93409491e+00  9.94332872e-01
  4.20701640e-01  5.26792683e-01  5.88473538e-01  6.91896518e-01
  6.43819911e-01  1.74389338e+00  6.47610783e-01  1.73385403e+00
 -2.05170188e-11 -4.46720859e-12  1.00537346e+00  1.84841076e+00
  1.51632379e+00  1.94523208e-11  5.36051745e-01  7.79517623e-01
  1.79144058e+00  3.81120262e-01  1.48278785e+00  1.27962387e+00
  1.14013396e+00  9.21158513e-01  1.09303719e-01  9.82115954e-01
  4.97499283e-01  8.47246748e-02  1.00221597e+00 -5.00321079e-12
 -5.28941400e-12  1.47129211e+00  1.36049931e+00  8.45247016e-01
  1.75725329e+00 -4.39136274e-12  1.91487316e+00  4.52254029e-01
  8.84881276e-01 -5.34394760e-12 -4.56852061e-12 -5.08346977e-12
  1.53727391e+00  6.85001281e-02  2.95736195e-01 -5.35358895e-12
  1.42228091e+00  6.70546696e-01  9.85771101e-01  7.88333444e-01
  1.74731289e+00  6.56823103e-01  2.46595810e-01 -5.53005475e-12
 -5.20285308e-12  1.94714989e+00  7.90776451e-01  3.80972741e-01
  1.01993472e+00  6.40703291e-01  5.82390201e-01  1.43414549e+00
  1.91363598e+00 -4.98841704e-12  3.74017293e-01  2.12065709e+00
  3.03715736e-01  1.13873867e+00  9.84335376e-01  9.08581901e-02
  4.42750635e-01  7.19772398e-01  1.83802542e+00  1.40687214e+00]
```

بنابراین، اگر ۱۰۰ نقطه داده داشته باشید، ۱۰۰ متغیر ξ_i متناظر خواهید داشت، یکی برای هر نقطه داده، و به همین دلیل است که پس از بهینه سازی، ۱۰۰ مقدار ξ را مشاهده می‌کنید. هر ξ_i 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.