COMP-SCI 128 PSET 6

Trey Whitehead

Worked with: I. Edsparr!

Due: 3 May 24 11:59 PM

```
# imports
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
from scipy.optimize import minimize
from matplotlib.patches import Rectangle
```

Problem 1a: Generate an Example

Howdy! I can't believe we've arrived at the last PSET. It has been a wonderful term.

Before we start Problem 1, i'd like to provide some key terms, which will give us intuition as we go through the problems:

(1) $D \in R^{m \times n}$: Ground truth matrix
(2) $M \in [0, 1]^{m \times n}$: Mask matrix that yields a value of 1 for known entries
(3) $D_M := M \odot D$: Represents our available and known data!

Our cost function will be:

$$min_X ||D_M - M \odot X||_F^2 + \lambda ||X||_*$$

The sub-differential of the Frobenius term of this will be:

$$-2M \cdot (D_M - M \cdot X)$$

```
# Set dimensions
m = 100
n = 25
r = 6
prob = 0.5

# Generate ground truth matrix
D = np.random.randn(m, r) @ np.random.randn(r, n)
M = np.random.choice([0, 1], size=(m, n), p = [1 - prob, prob])

# Available data that we actually know
D_M = D * M

# Initialize X:
np.random.seed(1738)
X = np.random.randn(m, n)


# Compute cost function
def cost(M, X, D_M, l, S):
    return np.linalg.norm(D_M - M * X, 'fro')**2 + l * np.sum(S)

# Compute MSE
def mse(D, X, M):
    m, n = D.shape
    mse = np.linalg.norm(D - X, 'fro')**2 / (m*n)
    return mse
```

Problem 1b: Subgradient Descent Method and Error

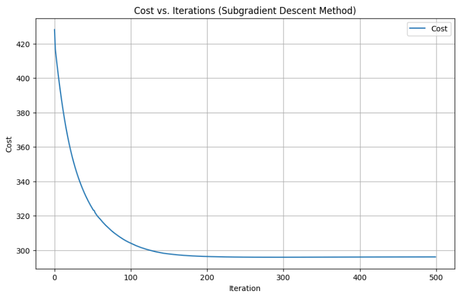
Before we dive into the deep end of the subgradient method, I would like to briefly summarize the specifications for our optimization process.

Given that we will have no W, our simplified sub-differential of $||X||_*$ will be:

$$\partial ||X||_* = UV^T$$

where $X = U\Sigma V^T$ in its SVD definition.

```
# Parameters

# Regularization paramater
l = 1
# Step size
step = 0.5
# Iterations
iters = 500


# Subdifferential for Frobenius norm term
def f_grad(M, X, D_M):
    return -2 * M * (D_M - M * X)

# Subdifferential for nuclear norm term
def n_grad(X):
    U, S, Vh = np.linalg.svd(X, full_matrices = False)
    return U @ Vh

# Run subgradient descent
def subgradient_method(M, D_M, X, l, step, iters):
    # Initialize subgradient costand subgradient cost mse
    sg_cost = np.zeros(iters)
    sg_mse = np.zeros(iters)

    # Run through iterations
    for i in range(iters):

        # Update X with subgradient
        X -= step * (f_grad(M, X, D_M) + l * n_grad(X))

        # Compute SVD of new X
        U, S, Vh = np.linalg.svd(X, full_matrices = False)

        # Find cost and MSE
        sg_cost[i] = cost(M, X, D_M, l, S)
        sg_mse[i] = mse(D_M, X, M)

    return sg_cost, sg_mse

f1_cost = subgradient_method(M, D_M, X.copy(), l, step, iters)


# Plot cost vs. iterations
plt.figure(figsize=(10, 6))
plt.plot(f1_cost[0], label='Cost')
plt.title('Cost vs. Iterations (Subgradient Descent Method)')
plt.xlabel('Iteration')
plt.ylabel('Cost')
plt.legend()
plt.grid(True)
plt.show()
```



Problem 1c: Proximal Descent Method and Error

Assuming that $X = U\Sigma V^T$, our proximal operator is given by:

$$prox_{tg}(X) = U\phi_t(\Sigma)V^T$$

where:

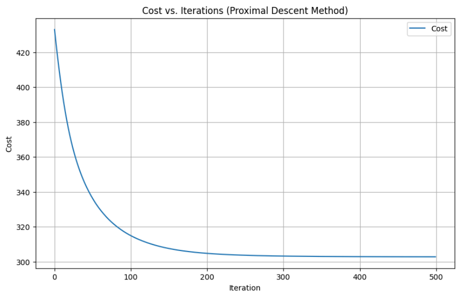
$\phi_t(\sigma) = 0$ if $-t \le \sigma \le t$

$\phi_t(\sigma) = \sigma - t$ if $\sigma > t$

$\phi_t(\Sigma) = diag(\phi_t(\sigma_i))$

The proximal gradient descent method helps us deal with our non-smooth operator!

```
# Set up proximal operator
def prox(U, S, Vh, step):
    # Now, LeLet's LeDo LeThresholding LeFunctionBron
    threshold = np.maximum(S - step, 0)
    # Matrix reconstructiond
    return U @ np.diag(threshold) @ Vh

# Run proximal descent
def proximaldescent_method(M, D_M, X, l, step, iters):
    # Initialize subgradient costand subgradient cost mse
    pd_cost = np.zeros(iters)
    pd_mse = np.zeros(iters)

    # Run through iterations
    for i in range(iters):

        # Update X with gradient descent
        X -= step * f_grad(M, X, D_M)

        # Compute SVD of X
        U, S, Vh = np.linalg.svd(X, full_matrices = False)
        # Use our proximal operator
        X = prox(U, S, Vh, step)

        # Find cost and MSE
        pd_cost[i] = cost(M, X, D_M, l, S)
        pd_mse[i] = mse(D_M, X, M)

    return pd_cost, pd_mse

f2_cost = proximaldescent_method(M, D_M, X.copy(), l, step, iters)


# Plot cost vs. iterations
plt.figure(figsize=(10, 6))
plt.plot(f2_cost[0], label='Cost')
plt.title('Cost vs. Iterations (Proximal Descent Method)')
plt.xlabel('Iteration')
plt.ylabel('Cost')
plt.legend()
plt.grid(True)
plt.show()
```



Problem 1d: An Alternative Method

We can also approach the optimization of our cost function with the following function:

$min_X ||D_M - M \odot X||_F^2$

subject to $rank(X) \le k$

Our rank function is non-convex in the lone constraint, so **this is not a convex problem!**

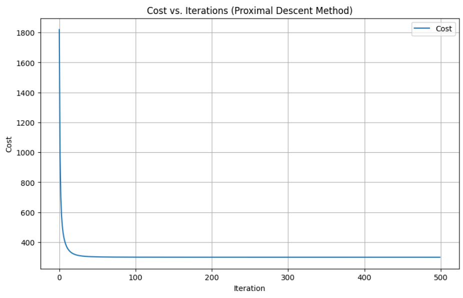
We will solve this problem with a projection operator. A projection operator is useful because it enables us to ensure that our iterates remain within the feasible region, even within a non-cvx problem.

Now, let's derive the projection operator. Our constraint dicattes that the rank of X must be less than or equal to k.Basically, this means that we need to project a matrix onto a set of matrices with a rank of k (at most). This can be done trivially with SVD. Once we have decomposed X, we can zero out all entries greater than k in the singular value matrix, effectively retaining the largest k singular values.

Problem 1e: Projected Descent Method and Error

```
# Define projection operator for constraint, set k to be original matrix rank
def proj(U, S, Vh, k=r):
    # Zero the entries beyond rank k
    S = np.concatenate((S[:k], np.zeros(len(S) - k)))
    # Rebuild X
    return U @ np.diag(S) @ Vh

# Run proximal descent
def projecteddescent_method(M, D_M, X, l, step, iters):
    # Initialize subgradient costand subgradient cost mse
    proj_cost = np.zeros(iters)
    proj_mse = np.zeros(iters)

    # Run through iterations
    for i in range(iters):

        # Update X with gradient descent
        X -= step * f_grad(M, X, D_M)

        # Compute SVD of X
```

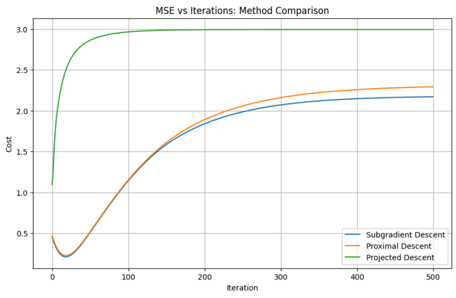
```
# Plot cost vs. iterations
plt.figure(figsize=(10, 6))
plt.plot(f3_cost[0], label='Cost')
plt.title('Cost vs. Iterations (Proximal Descent Method)')
plt.xlabel('Iteration')
plt.ylabel('Cost')
plt.legend()
plt.grid(True)
plt.show()
```



Problem 1f: MSE Comparison

Below, I have plotted each appraoch on a single plot with a commaon cost function of MSE:

```
# Plot cost vs. iterations
plt.figure(figsize=(10, 6))
plt.plot(f1_cost[1], label='Subgradient Descent')
plt.plot(f2_cost[1], label='Proximal Descent')
plt.plot(f3_cost[1], label='Projected Descent')
plt.title('MSE vs Iterations: Method Comparison')
plt.xlabel('Iteration')
plt.ylabel('Cost')
plt.legend()
plt.grid(True)
plt.show()
```



Discussion: It appears that the subgradient descent method and proximal descent methods perform similarly, although the subgradient method edges the proximal descent method out. At first, the projected descent method has the highest error, but it's cost remains very consistent after about 100 iterations. Therefore, it could potentially be the better if it was combined with some regularizing term. However, the main takeaway from the graph should be that subgradient descent performs pretty well!

Explanation of Results The increase in MSE for the SG and Proximal Descent Methods is interesting, especially considering their initial "dip" towards 0 error. I suspect that this is due, at least in part, to overfitting. As the models begin to take in new data, which doesn't fit with the trends of the original training data, it might struggle to accomodate it until the $300 - 400$ iteration range, where the error begins to level out. For projected descent, which involves a non-convex problem, perhaps the model gets recursively trapped in a locally-optimal point. In turn, this might keep the error constant at 3 from $200+$ iterations onward.

Problem 2ai: Intro to ADAM (Adaptive Moment Estimation)

The ADAM algorithm uses auxillary $\hat{m_t}$ and $\hat{v_t}$ and four parameters $\beta_1, \beta_2, \epsilon$, and α:

(1) $m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla f(x_t)$

(2) $v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\nabla f(x_t))^2$

(3) $\hat{m_t} = \frac{m_t}{1-\beta_1^t}$

(4) $\hat{v_t} = \frac{v_t}{1-\beta_2^t}$

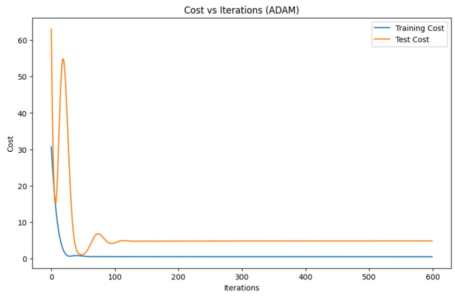
(5) $x_{t+1} = x_t - \frac{\alpha \hat{m_t}}{\sqrt{\hat{v_t}}+\epsilon}$

```python
# Imports
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
from scipy.optimize import minimize
from matplotlib.patches import Rectangle
```

```python
# Load the dataset
california = fetch_california_housing()

# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(california.data, california.target, test_size=0.95, random_state=42)

# Scale the data
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Add column of ones for bias term
X_train = np.c_[np.ones(X_train.shape[0]), X_train]
X_test = np.c_[np.ones(X_test.shape[0]), X_test]

# Run ADAM
def ADAM(X_train, X_test, y_train, y_test, iterations=600, epsilon=10e-8, alpha=0.1, b1=0.9, b2=0.999, random_seed=1738):
    # Initialize first moment vector
    m = np.zeros(X_train.shape[1])
    # Initialize second moment vector
    v = np.zeros(X_train.shape[1])
    # Initialize beta parameters
    np.random.seed(random_seed)
    beta = np.random.randn(X_train.shape[1])

    # Initialize lists for plotting
    train_cost_list = np.zeros(iterations)
    test_cost_list = np.zeros(iterations)

    for t in range(iterations):
        # Evaluate on train set
        y_pred_train = X_train @ beta
        train_cost_list[t] = mean_squared_error(y_train, y_pred_train)

        # Take a gradient step
        gradient = X_train.T @ (y_pred_train - y_train) / len(y_train)

        # The ADAM Project
        m = b1 * m + (1 - b1) * gradient
        v = b2 * v + (1 - b2) * (gradient ** 2)
        m_hat = m / (1 - b1 ** (t + 1))
        v_hat = v / (1 - b2 ** (t + 1))
        beta -= alpha * m_hat / (np.sqrt(v_hat) + epsilon)

        # Evaluate on test set after beta updates
        y_pred_test = X_test @ beta
        test_cost_list[t] = mean_squared_error(y_test, y_pred_test)

    # Print final coefficients
    print("Solution for regression:")
    print("Bias: " + str(beta[0]))
    for i in range(1, X_train.shape[1]):
        print(california.feature_names[i - 1] + ": " + str(beta[i]))

    # Plot train and test cost vs iterations
    plt.figure(figsize=(10, 6))
    plt.plot(train_cost_list, label='Training Cost')
    plt.plot(test_cost_list, label='Test Cost')
    plt.legend()
    plt.xlabel('Iterations')
    plt.ylabel('Cost')
    plt.title('Cost vs Iterations (ADAM)')
    plt.show()
```

```python
ADAM(X_train, X_test, y_train, y_test, iterations=600, epsilon=10e-8, alpha=0.1, b1=0.9, b2=0.999, random_seed=1738)
```

```
Solution for regression:
Bias: 2.1134976065891102
MedInc: 0.9153895586616435
HouseAge: 0.13336391726302554
AveRooms: -0.3082045802706534
AveBedrms: 0.2926706117473876
Population: 0.07496383579717215
AveOccup: -0.19941002280229367
Latitude: -0.823225316460559
Longitude: -0.7888419847740828
```

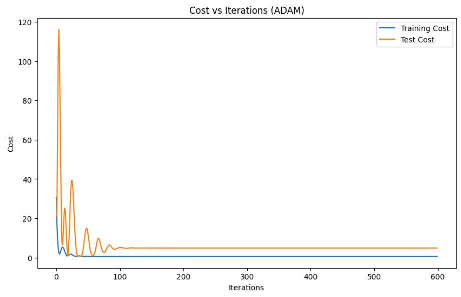


Problem 2aii: Tuning the System

Test One: Increasing α

```python
ADAM(X_train, X_test, y_train, y_test, iterations=600, epsilon=10e-8, alpha=0.5, b1=0.9, b2=0.999, random_seed=1738)
```

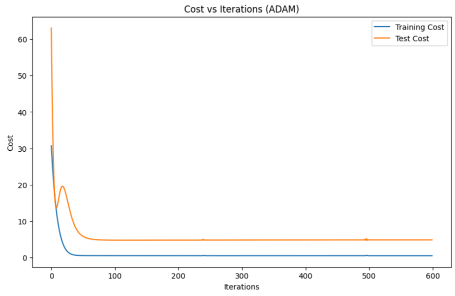
```
Solution for regression:
Bias: 2.1134976065891675
MedInc: 0.9144098220359018
HouseAge: 0.13301953258122842
AveRooms: -0.3070875086033457
AveBedrms: 0.29209156101760864
Population: 0.07491407009809338
AveOccup: -0.19943105377179743
Latitude: -0.8273962809935674
Longitude: -0.7929007482997744
```



Test Three: Decreasing β_1

```
ADAM(X_train, X_test, y_train, y_test, iterations=600, epsilon=10e-8, alpha=0.1, b1=0.5, b2=0.999, random_seed=1738)
```

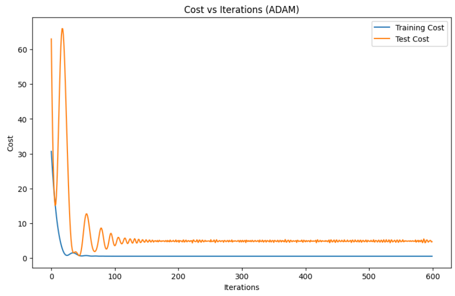
```
Solution for regression:
Bias: 2.113497606589175
MedInc: 0.9162502646290325
HouseAge: 0.13366019574870955
AveRooms: -0.30919358445211725
AveBedrms: 0.29318939239653924
Population: 0.07500510531105133
AveOccup: -0.19939059735995412
Latitude: -0.8196119796017746
Longitude: -0.7853259985634296
```



Test Four: Decreasing β_2

```
ADAM(X_train, X_test, y_train, y_test, iterations=600, epsilon=10e-8, alpha=0.1, b1=0.9, b2=0.9, random_seed=1738)
```

```
Solution for regression:
Bias: 2.1093993444189323
MedInc: 0.9107203234390112
HouseAge: 0.12829753271262687
AveRooms: -0.3087451559641791
AveBedrms: 0.29141689407788807
Population: 0.0800549730694198
AveOccup: -0.19553124906469813
Latitude: -0.8272051414482544
Longitude: -0.7925244573111149
```



Discussion: It appears that increasing α, or the step size, leads to the smallest difference between train and test error!

Other takeaways:

- Decreasing β_1: Reduces amplitude of initial oscillations.
- Decreasing β_2: Can increase amplitude of oscillations around optimum.

Problem 2b: Explain how ADAM works

Given our equations and parameter explanation above, I feel that it would be best to go equation by equation to describe how ADAM works. I will not describe the roll of the train vs. test split very deeply, as this was covered in PSET 4.

For context, a **moment** is a quantitative measure that summarizes certain aspects of the shape, distribution, or structure of a set of values or a probability distribution.

The role of each parameter appears to be as follows:

- β_1: Controls the exponential decay rate for the first moment estimate m_t. In other words, it determines how much of the past gradients are incorporated into the current estimate of the first moment
- β_2: Controls the exponential decay rate for the second moment estimate v_t. It governs the accumulation of past squared gradients and influences the scaling of the learning rates for individual parameters.

- ϵ: Prevents division by 0 in Equation 5
- α: The step size for iterations.

(1) $m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla f(x_t)$

In this equation for the first moment vector, we find the next iterate of the first moment, m_t with a weighted, running average. The bigger the β_1 decay paramter is, the more the value of the previous first moment is taken into account. When β_1 is smaller, the gradient of $\nabla f(x_t)$ plays a larger role. I would guess that this helps smooth-out the function over time.

(2) $v_t = \beta_2 v_{t-1} + (1-\beta_2)(\nabla f(x_t))^2$

In this equation for the second moment vector, we also find the next iterate of the second moment, v_t with a weighted average. The bigger the β_2 decay paramter is, the more the value of the previous second moment is taken into account. When β_2 is smaller, the gradient of $\nabla f(x_t)$ plays a larger role. What distinguishes this equation from the first, however, is the squared gradient. While I am not 100% sure what this does, the graphs above indicate that this plays an increased importance for oscillations (or lack thereof) around the optimum. The big idea here is that the adaptive scaling technique helps us take a moving average that plays an integral role in equation (5), which I will explain below.

(3) $\hat{m_t} = \frac{m_t}{1-\beta_1^t}$
(4) $\hat{v_t} = \frac{v_t}{1-\beta_2^t}$

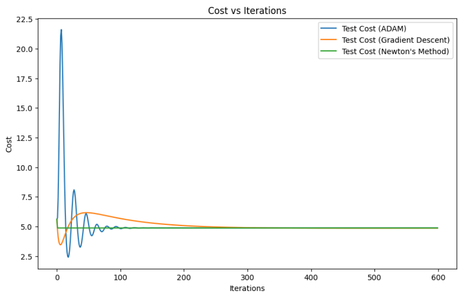
Equations (3) and (4) serve a similar purpose, so I will explain what they do together. They both correct for the bias introduced by the initialization of m and v at the beginning of ADAM by dividing the current estimate by a constant factor (tied to correlation). I wasn't 100% sure what these equations did, so I used this link here to motivate my answer to this question!

(5) $x_{t+1} = x_t - \frac{\alpha \hat{m_t}}{\sqrt{\hat{v_t}}+\epsilon}$

Equation (5) is the update step! It finds the $t+1$ input value by subtracting the new first moment, scaled by the step size, divided by the square root of second moment added to a small value to prevent division by 0. The units of the second term correspond to those of the first (x_t), which makes the update computationally work! The step in the second term is proportional to the first moment and inversely proportional to the second moment, which probably explains why decreasing β_2 induces more oscillations with increased iterations around the optimum.

Problem 2c: Comparing ADAM, Newton's Method, and Gradient Descent

```
# ADAM
def ADAM_simple(X_train, X_test, y_train, y_test, iterations=600, epsilon=10e-8, alpha=0.1, b1=0.9, b2=0.999):
    # Initialize first moment vector
    m = np.zeros(X_train.shape[1])
    # Initialize second moment vector
    v = np.zeros(X_train.shape[1])

    # Initialize beta
    beta = np.zeros(X_train.shape[1])

    # Initialize test cost list for plotting
    test_cost_list = np.zeros(iterations)

    # Compute initial MSE
    y_pred_test_initial = X_test @ beta
    test_cost_list[0] = mean_squared_error(y_test, y_pred_test_initial)

    for t in range(1, iterations):
        # Evaluate on train set
        y_pred_train = X_train @ beta

        # Take a gradient step
        gradient = X_train.T @ (y_pred_train - y_train) / len(y_train)

        # The ADAM Project AGAIN
        m = b1 * m + (1 - b1) * gradient
        v = b2 * v + (1 - b2) * (gradient ** 2)
        m_hat = m / (1 - b1 ** (t + 1))
        v_hat = v / (1 - b2 ** (t + 1))
        beta -= alpha * m_hat / (np.sqrt(v_hat) + epsilon)

        # Evaluate on test set after beta updates
        y_pred_test = X_test @ beta
        test_cost_list[t] = mean_squared_error(y_test, y_pred_test)

    # Plot train and test cost vs iterations
    plt.figure(figsize=(10, 6))
    plt.plot(test_cost_list, label='Test Cost (ADAM)')

# Gradient Descent
def gradient_descent(X_train, X_test, y_train, y_test, alpha=0.1, iterations=600):
    # Initialize beta parameters
    beta = np.zeros(X_train.shape[1])

    # Initialize test cost list for plotting
    test_cost_list = np.zeros(iterations)

    # Compute initial MSE
    y_pred_test_initial = X_test @ beta
    test_cost_list[0] = mean_squared_error(y_test, y_pred_test_initial)

    for t in range(1, iterations):
        # Evaluate on train set
        y_pred_train = X_train @ beta

        # Take a gradient step
        gradient = X_train.T @ (y_pred_train - y_train) / len(y_train)
        beta -= alpha * gradient

        # Evaluate on test set after beta updates
        y_pred_test = X_test @ beta
        test_cost_list[t] = mean_squared_error(y_test, y_pred_test)

    # Plot train and test cost vs iterations
    plt.plot(test_cost_list, label='Test Cost (Gradient Descent)')

# Newton's Method
def newtons_method(X_train, X_test, y_train, y_test, iterations=600):
    # Initialize beta parameters
    beta = np.zeros(X_train.shape[1])

    # Initialize test cost list for plotting
    test_cost_list = np.zeros(iterations)

    # Compute initial MSE
    y_pred_test_initial = X_test @ beta
    test_cost_list[0] = mean_squared_error(y_test, y_pred_test_initial)

    for t in range(1, iterations):
        # Evaluate on train set
        y_pred_train = X_train @ beta

        # Compute Hessian and gradient
        gradient = X_train.T @ (y_pred_train - y_train) / len(y_train)
        hessian = X_train.T @ X_train / len(y_train)

        # Update beta using Newton's Method
        beta -= np.linalg.inv(hessian) @ gradient

        # Evaluate on test set after beta updates
        y_pred_test = X_test @ beta
        test_cost_list[t] = mean_squared_error(y_test, y_pred_test)

    # Plot train and test cost vs iterations
    plt.plot(test_cost_list, label='Test Cost (Newton\'s Method)')

# Plot all three methods on the same graph
ADAM_simple(X_train, X_test, y_train, y_test)
gradient_descent(X_train, X_test, y_train, y_test)
newtons_method(X_train, X_test, y_train, y_test)
plt.xlabel('Iterations')
plt.ylabel('Cost')
plt.title('Cost vs Iterations')
plt.legend()
plt.show()
```



Discussion:

Between the three methods, here is what was held constant:

- Iterations = 600
- α = 0.1 (for ADAM and GD)

Between the three methods, here is what was different:

- ADAM: Has β parameters and an $epsilon$ input, which the other methods do not have.
- GD: Has α-step input, which Newton's does not have.
- NEWTON's: Aside from the test and training data inputs, Newton's Method only requires an iteration number input (like the other two methods).

Results:

- NEWTON'S METHOD: For uncontrained QPs, it looks like Newton's Method converges in a single iteration!
- GRADIENT DESCENT: For gradient descent, finding the optimal step size is important. If we were to tweak it more, perhaps we could make it more efficient, but I have displayed the results for different alpha values below:
  - $\alpha = 0.01$: Slow convergence (600+ iterations needed)
  - $\alpha = 0.1$: Convergence around 250 iterations (displayed above)
  - $\alpha = 1$: Divergence!
- ADAM: Adam's Method, for the current paramter values, converges at aroudn 100 iterations. At $\alpha = 0.1$, it beats GD, but nothing comes close to the one-step convergence of Newton's Method.

Conclusions:

- Newton's Method, a second-order method, converges quite quickly for this model.
- ADAM, an adaptive learning rate optimization algorithm, seems to converge more quickly than the GD method.
- GD: Slowest convergence for high-dimensional problems!

Problem 3a: Barrier Method

```
# Imports
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
from scipy.optimize import minimize
from matplotlib.patches import Rectangle
```

We have been instructed with implement a barrier method and a primal-dual method for solving the following QP:

Deriving the Barrier Function and its various Derivatives

minimize $\frac{1}{2}x^TPx + q^Tx$
subject to $Ax \leq b$

We will begin with the barrier method, which is a method for solving convex optimization problems that include inequality constraints. Fortunately, the CS128 course staff took mercy on us, so we will not have to graple with equality constraints for this problem!

Please note, the following logic is pulled from chapter 11 of the B&V textbook.

We begin by rewriting our objective function so that the m constraints are implicit within it. Let $f_0(x) = \frac{1}{2}x^TPx + q^Tx$ and $f_i(x) = (Ax)_i - b_i$.

$$f_0(x) + \sum_{i=1}^{m} I_-(f_i(x))$$

where the indicator function works as follows:

$I_-(u) = 0$ if $x \leq 0$
$I_-(u) = \infty$ if $x \geq 0$

Per B&V, the big idea of the barrier method is to approximate $I_-(u)$ as $-(1/t)log(-u)$ on a domain of $-R_{++}$ (where t is the step size).

For our QP, we will call our barrier function:

$\phi(x) = -\sum_{i=1}^{m} log(-f_i(x))$
$\phi(x) = -\sum_{i=1}^{m} log(b_i - (Ax)_i)$

Its gradient and Hessian are included below:

$\nabla\phi(x) = \sum_{i=1}^{m} \frac{A}{b_i-(Ax)_i}$

$\nabla^2\phi(x) = \sum_{i=1}^{m} \frac{A_iA_i^T}{((Ax)_i-b_i)^2}$

From here, we have enough bones to build the skeleton of the barrier function, which combines our original function and our inequality constraint. The barrier function, its gradient, and Hessian have been included below:

$B(x,t) = tf_0(x) + \phi(x)$

$B(x,t) = t(\frac{1}{2}x^TPx + q^Tx) - \sum_{i=1}^{m} log(-f_i(x))$

$\nabla B(x,t) = t(Px + q) + \sum_{i=1}^{m} \frac{A}{b_i-(Ax)_i}$

$\nabla^2 B(x,t) = tP + \sum_{i=1}^{m} \frac{A_iA_i^T}{((Ax)_i-b_i)^2}$

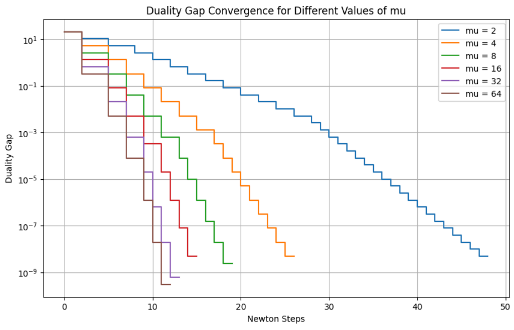
What is "Newton's Method?"

The negative Hessian of the objective function represents the curvature of the function at the current paramter values. By projecting the gradient onto it, we can find the direction of steepest descent, which if immplemented correctly, will enable our optimization process!

$x^{(k)} = x^{(k-1)} - \theta(\nabla^2 f(x^{(k-1)}))^{-1}\nabla f(x^{(k-1)})$

Putting it all together Now, let's put together everything we explained above into graphs with a little bit of Python:

```python
# Set initial paramter values
alpha = 0.1
beta = 0.5
seed = 69

# Generate random inputs
def generate_random_values(seed):
    np.random.seed(seed)

    # Set dimensions randomly
    m = np.random.randint(10, 30)
    n = np.random.randint(10, 30)

    # Pick X, y, and q randomly
    X = np.random.rand(m, n)
    y = np.random.rand(m, 1)
    q = np.random.rand(m, 1)

    # Ensure P is positive-definite
    P = np.eye(m)

    # Constraint wasn't working
    # ChatGPT suggested working with this constraint, which admittedly is
    # not m x n, but it enabled my hessian to compute
    # Furthermore, the rest of the problems in the graph work!
    A = np.vstack((X.T, -X.T))
    b = 10 * np.ones((2 * n, 1))

    # Initial guess for x
    x = np.zeros((m, 1))
    return m, n, X, y, P, q, A, b, x

m, n, X, y, P, q, A, b, x = generate_random_values(seed)

# Build out mus array, which will grow exponentially to show impact
mus = []
current = 2
for i in range(6):
    mus.append(current)
    current *= 2

# Function to evaluate the objective function f
def QP(x, P, q):
    return x.T @ P @ x + q.T @ x

# Barrier function
def bf(x, P, q, t, A, b):
    return t * (x.T @ P @ x + q.T @ x) - np.sum(np.log(b - A @ x))

# Gradient of the barrier function
def bf_g(x, P, q, A, b, t):
    grad = t * (2 * P @ x + q)
    for i in range(len(b)):
        grad += A[i, np.newaxis].T / (b[i] - A[i] @ x)
    return grad

# Hessian of the barrier function
def bf_h(x, P, q, A, b, t):
    H = 2 * t * P
    for i in range(len(b)):
        factor = A[i] / (b[i] - A[i] @ x)
        H += np.outer(factor, factor)
    return H

# Backtracking line search
# Debugged by ChatGPT!
def backtracking_line_search(P, q, x, t, bf, bf_g, A, b, dx, t_initial, alpha, beta, tol):
    f = bf(x, P, q, t_initial, A, b)
    grad_f = bf_g(x, P, q, A, b, t_initial)
    t_step = 1.0
    while (bf(x + t_step * dx, P, q, t_initial, A, b) > f + alpha * t_step * (grad_f.T @ dx) or np.any(b - A @ (x + t_step * dx) < 0)) and t_step > tol:
        t_step *= beta
    return x + t_step * dx


# Barrier method with backtracking line search
def barrier_method(P, q, A, b, x, t0=1, mu=10, alpha=0.1, beta=0.7, eps=1e-10, max_iter=300, NTTOL=1e-4, tol=1e-8):
    t = t0
    iters = 0
    duality_gaps = []

    while True:
        iters += 1

        # Step 1: Solve the Newton system
        H = bf_h(x, P, q, A, b, t)
        grad = bf_g(x, P, q, A, b, t)
        fprime = np.linalg.solve(H, -grad)

        # Step 2: Backtracking line search
        x_new = backtracking_line_search(P, q, x, t, bf, bf_g, A, b, fprime, t, alpha, beta, tol)

        # Step 3: Compute duality gap
        duality_gap = m / t
        duality_gaps.append(duality_gap)

        # Step 4: Check for convergence
        if abs(np.linalg.norm(fprime)) < NTTOL:
            gap = (2 * m) / t
            # print(f"Iteration: {iters}; Gap = {gap}")
            if gap < tol or iters >= max_iter:
                break
            t = mu * t

        # Step 5: Update x for next iteration
        x = x_new

    return x, duality_gaps

# Plot duality gap for all mus
plt.figure(figsize=(10, 6))
for mu_value in mus:
    x_sol, duality_gaps = barrier_method(P, q, A, b, x, t0=1, mu=mu_value)
    plt.step(range(len(duality_gaps)), duality_gaps, label=f'mu = {mu_value}')

plt.xlabel('Newton Steps')
plt.ylabel('Duality Gap')
plt.yscale('log')
plt.title('Duality Gap Convergence for Different Values of mu')
plt.legend()
plt.grid(True)
plt.show()
```



Problem 3b: Primal-Dual Interior-Point (PDIP) Methods

The PDIP is similar to the barrier method, so we will recycle some of our code for our functions. However, there are a couple key differences.

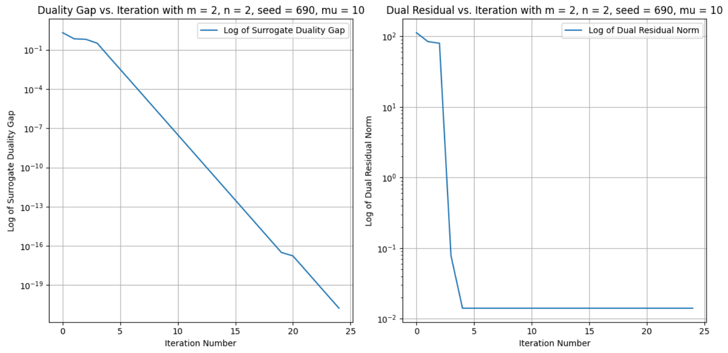
- There is only 1 loop/iteration
- The search directions we obtain from Newton's Method (steepest descent) are applied to modified KKT conditions, which are the optimality conditions for the logarithmic barrier centering problem.
- In a PDIP method, the primal and dual iterates are not necessarily feasible

The benefits of PDIP methods are:

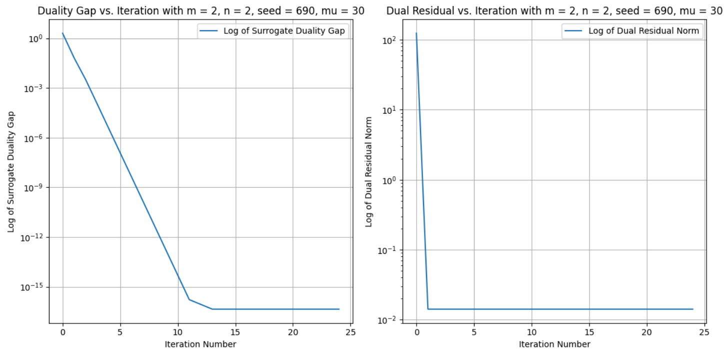
- Quicker covergence, which can improrve accuracy!
- For many SOCPs, PDIP outperforms the barrier method

Prior to implementation, here is an important note: (1) The surrogate gap is the duality gap IF x is primally feasible and λ is dual feasible.

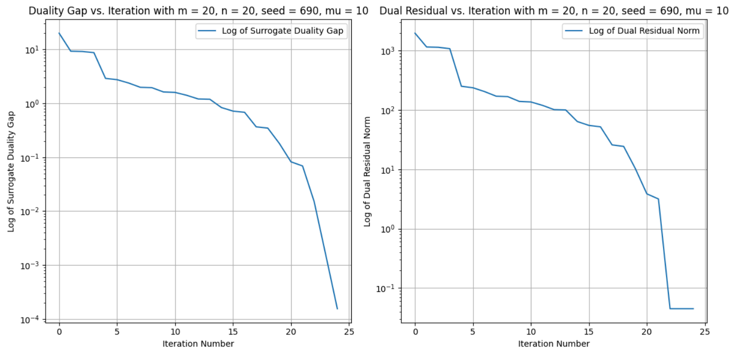
```
# Find a feasible starting point for optimization
def find_feasible_starting_point(A, b):
    return np.linalg.lstsq(A, b, rcond=None)[0]

# Function to perform primal-dual interior point optimization
def pdip(m, n, max_iters=25, tol=1e-8, mu=10, alpha=0.01, beta=0.5, seed=690):

    # Build out matrices
    P = np.random.randn(n, n)
    P = P.T @ P  # Make P symmetric positive definite
    q = np.random.randn(n)
    A = np.random.randn(m, n)
    b = np.random.randn(m)

    m, n = A.shape

    # Initialize variables (main, slack, dual)
    x = find_feasible_starting_point(A, b)
    # Initialize s with slack variables
    s = np.maximum(0.01, b - A @ x)
    # Initialize z as the reciprocal of s
    z = 1.0 / s

    # Initialize lists for plotting
    gap_values = []
    res_norm_values = []

    # Main optimization loop
    for iters in range(1, max_iters + 1):
        # Calculate residuals for the current iteration
        r_dual = P @ x + q + A.T @ z
        r_prim = A @ x - b + s

        # Compute residuals norm and duality gap
        res = np.linalg.norm(r_dual) + np.linalg.norm(r_prim)
        gap = np.dot(s, z)

        # Store the residuals and gap for plotting
        res_norm_values.append(res)
        gap_values.append(gap)

        # Compute t for the central residual
        t = gap / (m * mu)

        # Form the KKT system and solve for the step direction
        M = np.block([[P, A.T], [A, np.diag(-s / z)]])
        rhs = np.concatenate([r_dual, -s + t * (1.0 / z)])
        delta = -np.linalg.solve(M, rhs)
        dx = delta[:n]
        dz = delta[n:]
        ds = -A @ dx

        # Ensure gap and residual norm fall below tolerance
        if gap < tol and res < tol:
            break

        # Line search for step size
        r = np.concatenate([P @ x + q + A.T @ z, z * s - t])
        step = min(1.0, 0.99 / max(-dz / z))
        while min(s + step * ds) <= 0:
            step *= beta

        # Update x, z, and s using the step size
        x1 = x + step * dx
        z1 = z + step * dz
        s1 = s + step * ds

        # Update the residuals for the new variables
        r1 = np.concatenate([P @ x1 + q + A.T @ z1, z1 * s1 - t])

        # Perform backtracking line search until the Armijo condition is met
        while np.linalg.norm(r1) > (1 - alpha * step) * np.linalg.norm(r):
            step *= beta
            x1 = x + step * dx
            z1 = z + step * dz
            s1 = s + step * ds
            r1 = np.concatenate([P @ x1 + q + A.T @ z1, z1 * s1 - t])

        # Update variables for the next iteration
        x, z, s = x1, z1, s1

    # Return the optimized variables and the logged values
    return m, n, seed, mu, x, gap_values, res_norm_values, iters

def plot(m, n, seed, mu, gap_values, res_norm_values):
    # Plotting the results
    plt.figure(figsize=(12, 6))

    # Plot the duality gap vs. iteration number
    plt.subplot(1, 2, 1)
    plt.plot(gap_values, label='Log of Surrogate Duality Gap')
    plt.xlabel('Iteration Number')
    plt.ylabel('Log of Surrogate Duality Gap')
    plt.yscale('log')
    plt.title(f'Duality Gap vs. Iteration with m = {m}, n = {n}, seed = {seed}, mu = {mu}')
    plt.grid(True)
    plt.legend()

    # Plot the dual residual norm vs. iteration number
    plt.subplot(1, 2, 2)
    plt.plot(res_norm_values, label='Log of Dual Residual Norm')
    plt.xlabel('Iteration Number')
    plt.ylabel('Log of Dual Residual Norm')
    plt.yscale('log')
    plt.title(f'Dual Residual vs. Iteration with m = {m}, n = {n}, seed = {seed}, mu = {mu}')
    plt.grid(True)
    plt.legend()

    plt.tight_layout()
    plt.show()


# Example 1: Base Case
m, n, seed, mu, x, gap_values, res_norm_values, iters = pdip(2, 2, max_iters=25, tol=1e-8, mu=10, alpha=0.01, beta=0.5, seed=690)
plot(m, n, seed, mu, gap_values, res_norm_values)
```



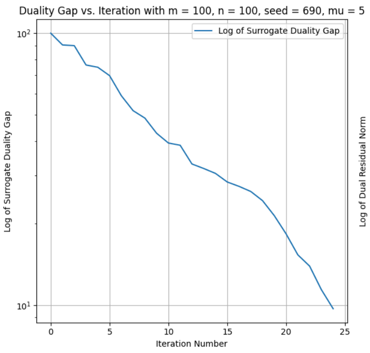
```
# Example 2: Tripling mu
m, n, seed, mu, x, gap_values, res_norm_values, iters = pdip(2, 2, max_iters=25, tol=1e-8, mu=30, alpha=0.01, beta=0.5, seed=690)
plot(m, n, seed, mu, gap_values, res_norm_values)
```

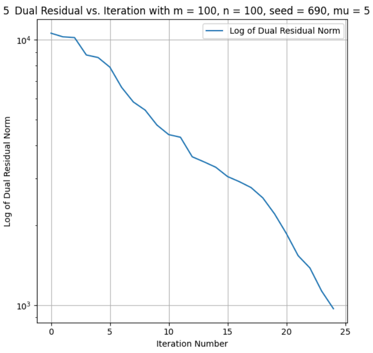


```
# Example 3: 10xing both m and n
m, n, seed, mu, x, gap_values, res_norm_values, iters = pdip(20, 20, max_iters=25, tol=1e-8, mu=10, alpha=0.01, beta=0.5, seed=690)
plot(m, n, seed, mu, gap_values, res_norm_values)
```



```
# Example 4: The Cowabunga Case (changing a lot)
m, n, seed, mu, x, gap_values, res_norm_values, iters = pdip(100, 100, max_iters=25, tol=1e-8, mu=5, alpha=0.01, beta=0.5, seed=690)
plot(m, n, seed, mu, gap_values, res_norm_values)
```


Duality Gap vs. Iteration with m = 100, n = 100, seed = 690, mu = 5
Log of Surrogate Duality Gap
Log of Surrogate Duality Gap
Iteration Number
10^2
10^1
0
5
10
15
20
25

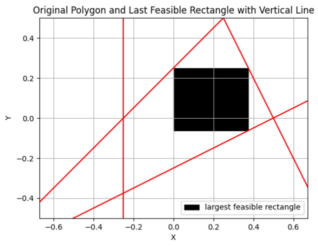

Dual Residual vs. Iteration with m = 100, n = 100, seed = 690, mu = 5
Log of Dual Residual Norm
Log of Dual Residual Norm
Iteration Number
10^4
10^3
0
5
10
15
20
25

Problem 4: Max Volume Rectangle Inside a Polyhedron

Given that we discussed the barrier method in problem 3, my work on this problem will be results-focused! I have included my pseudocode and code below. I used ChatGPT to debug my modified script from 3 along the way. I also used Chatper 11 from B&V to motivate my solution!

```
# Imports
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
from scipy.optimize import minimize
from matplotlib.patches import Rectangle


# Define A and b
A = np.array([[0, -1],
              [2, -4],
              [2, 1],
              [-4, 4],
              [-4, 0]])

m, n = A.shape

b = np.ones(A.shape[0])

# Separate Aminus and Aplus
def separator(A):
    Aminus = np.maximum(-A, 0)
    Aplus = np.maximum(A, 0)
    return Aminus, Aplus

Aminus, Aplus = separator(A)

# Initialize parameters
maxiters = 200
alpha = 0.1
beta = 0.5
NTTOL = 1e-8
mu = 20
TOL = 1e-4
r = max(Aplus @ np.ones(n) + Aminus @ np.ones(n))
u = (.5/r) * np.ones(n)
l = -(.5/r) * np.ones(n)
t = 1

# # Initialize objective function
# def of(u, l):
#     return -np.sum(np.log(u - l))

# Initialize constraint
def C(A, b, u, l):
    A_plus = np.maximum(A, 0)
    A_minus = np.maximum(-A, 0)
    return A_plus @ u - A_minus @ l - b

# Barrier function
def bf(u, l):
    return -t * np.sum(np.log(u - l)) - np.sum(np.log(y))

# Barrier function gradient
def bf_g(u, l):
    t1 = (t * np.vstack((1/(u-l),-1/(u-l)))).flatten()
    t2 = (np.vstack((-Aminus.T,Aplus.T)) @ (1/y))
    return t1 + t2

# Barrier function Hessian
def bf_h(u, l):
    t1 = t * np.block([[(np.diag(1 / (u - l) ** 2)), -(np.diag(1 / (u - l) ** 2))],
                      [-(np.diag(1 / (u - l) ** 2)), (np.diag(1 / (u - l) ** 2))]])
    t2 = np.vstack((-Aminus.T, Aplus.T)) @ np.diag(1 / y ** 2) @ np.hstack((-Aminus, Aplus))
    return t1 + t2

# Initialize matrix for storing rectangle points, derived from u and l (upper and lower bound)
rect = []

# Set iteration counter to 0
iter = 0

# Run a while loop before max iteration counter
while iter < maxiters:
    # Find initial constraint value
    y = -C(A, b, u, l)
    # Find initial barrier function value
    val = bf(u, l)
    # Find gradient and hessian of barrier function for centering step calculation
    grad = bf_g(u, l)
    hess = bf_h(u,l)

    # Calculate Newton step and directional derivative
    step = -np.linalg.inv(hess) @ grad
    fprime = grad.T @ step

    # Check for convergence
    if abs(fprime) < NTTOL:
        gap = (2 * m) / t
        print(f"Iteration:{iter}; Gap = {gap}")
        if gap < TOL:
            break
        t = mu * t
    else:
        # Compute centering step
        dl = step[:n]
        du = step[n:]
        dy = Aminus @ dl - Aplus @ du

        tls = 1
        # Backtracking line search
        # ChatGPT and B&V helped me complete this part, as my initial inputs were faulty
        while np.min([u - l + tls * (du - dl)]) <= 0 or np.min(y + tls * dy) <= 0:
            tls = beta * tls

        while -t * np.sum(np.log(u - l + tls * (du - dl))) - np.sum(np.log(y + tls * dy)) >= \
                val + tls * alpha * fprime:
            tls = beta * tls

        # Update lower and upper bounds
        l = l + tls * dl
        u = u + tls * du

    # Update iterations and append bounds
    iter += 1
    rect.append(np.vstack((l, u)))

# Print final upper and lower bound
print(f"\nLower Bound: {l}")
print(f"Upper Bound: {u}")

# Plotting the original polygon A
plt.figure()

# Plot each line defined by a row of the matrix A
for i in range(len(A)):
    # Generate x values for the line
    x = np.linspace(-10, 10, 100)
    if A[i, 1] != 0:
        # Calculate y values for the line
        y = (b[i] - A[i, 0] * x) / A[i, 1]
        plt.plot(x, y, color='red')

# Add a vertical line at x = -0.25
plt.axvline(x=-0.25, color='red')

# Plot the last feasible rectangle
width = u[0] - l[0]
height = u[1] - l[1]
rect = Rectangle((l[0], l[1]), width, height, linewidth=1, edgecolor='black', facecolor='black', label='largest feasible rectangle')
plt.gca().add_patch(rect)
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Original Polygon and Last Feasible Rectangle with Vertical Line')
plt.legend()
plt.grid(True)
plt.axis('equal')
plt.xlim(-0.5, 0.5)
plt.ylim(-0.5, 0.5)
plt.show()

# Print rectangle volume
print(f"Max Rectangle Volume: {(width * height).round(5)}")
```

```
Iteration:3; Gap = 10.0
Iteration:9; Gap = 0.5
Iteration:20; Gap = 0.025
Iteration:29; Gap = 0.00125
Iteration:35; Gap = 6.25e-05

Lower Bound: [-5.20291016e-08 -6.25003776e-02]
Upper Bound: [0.37499534 0.2499976 ]
```



```
Max Rectangle Volume: 0.11719
```