MARENGO Matteo
matteo.marengo@ens-paris-saclay.fr

1 Derivation of the dual problem of LASSO

Given $x_1, \ldots, x_n \in \mathbb{R}^d$ data vectors and $y_1, \ldots, y_n \in \mathbb{R}$ observations, we are searching for regression parameters $w \in \mathbb{R}^d$ which fit data inputs to observations y by minimizing their squared difference.

$$\text{minimize} \quad \frac{1}{2}\|Xw - y\|_2^2 + \lambda\|w\|_1 \tag{1}$$

in the variable $w \in \mathbb{R}^d$, where $X = (x_1^T, \ldots, x_n^T) \in \mathbb{R}^{n\times d}$; $y = (y_1, \ldots, y_n)^T \in \mathbb{R}^n$ and $\lambda > 0$ is a regularization parameter.

We have to derive the dual problem of LASSO and format it as a general Quadratic Problem as follows:

$$\begin{array}{ll} \text{minimize} & v^T Q v + p^T v \\ \text{subject to} & Av \leq b \end{array} \tag{2}$$

(QP) in variable $v \in \mathbb{R}^n$, where $Q \succeq 0$.

First, let's define a new variable $z = Xw - y$.

Then the Lasso problem becomes an equality constrained problem such as:

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}\|z\|_2^2 + \lambda\|w\|_1 \\ \text{subject to} & z = Xw - y \end{array} \tag{3}$$

It is a standard form problem, we can compute its dual problem.

1.1 Lagrangian

As there are no inequality counstraints, but only equality ones, we define $w \in \mathbb{R}^d$, $z \in \mathbb{R}^n$ and $\nu \in \mathbb{R}^n$ and then the Lagrangian is as follows :

$$\begin{aligned} L(w; z; \lambda) &= f_0(x) + \sum_{i=1}^{p} \nu_i h_i(x) \\ &= \frac{1}{2}\|z\|_2^2 + \lambda\|w\|_1 + \nu^T(z - Xw + y) \\ &= \frac{1}{2}\|z\|_2^2 + \nu^T z + \lambda\|w\|_1 - \nu^T X w + \nu^T y \end{aligned} \tag{4}$$

1.2 Lagrangian dual function:

$$\begin{aligned} g(\nu) &= \inf_{w,z} L(w; z; \nu) \\ &= \inf_{w,z}(\frac{1}{2}\|z\|_2^2 + \nu^T z + \lambda\|w\|_1 - \nu^T X w + \nu^T y) \\ &= \nu^T y + \inf_z \left(\frac{1}{2}\|z\|_2^2 + \nu^T z\right) + \inf_w \left(\lambda\|w\|_1 - \nu^T X w\right) \end{aligned} \tag{5}$$

Therefore we have to solve two distinct terms in order to compute the lagrangian dual function; $\inf_z \left(\frac{1}{2}\|z\|_2^2 + \nu^T z\right)$ and $\inf_w \left(\lambda\|w\|_1 - \nu^T X w\right)$.

1.2.1 Study of $\inf_z \left(\frac{1}{2}\|z\|_2^2 + \nu^T z\right)$

Here we can define a function $f(z) = \frac{1}{2}\|z\|_2^2 + \nu^T z$. This function is convex as a linear combination of a linear function and a polynomial one. Furthermore this function is differentiable.

Let's compute the gradient of this function:

- The squared Euclidean norm $\|z\|_2^2$ is given by $\sum_i z_i^2$, where z_i are the components of the vector z.
- The dot product $\lambda^T z$ is given by $\sum_i \lambda_i z_i$, where λ_i are the components of the vector λ.

The gradient of $f(z)$ is the vector of partial derivatives with respect to each component of the input vector.

1. **Gradient of the Squared Norm:** The derivative of $\frac{1}{2}\|z\|_2^2$ with respect to z_i is z_i.
2. **Gradient of the Dot Product:** The derivative of $\lambda^T z$ with respect to z_i is λ_i.

Combining these results, the gradient of $f(z)$ is:

$$\begin{aligned} \nabla h(z) &= \left[\frac{\partial h}{\partial z_1}, \frac{\partial h}{\partial z_2}, \ldots, \frac{\partial h}{\partial z_n}\right] \\ \text{where each component} \qquad & \frac{\partial h}{\partial z_i} = z_i + \lambda_i \end{aligned} \tag{6}$$

Thus, the gradient is:

$$\nabla h(z) = z + \lambda \tag{7}$$

Then, $\nabla h(z) = 0 \longleftrightarrow z = -\nu$

Hence:

$$\begin{aligned} \inf_z \left(\frac{1}{2}\|z\|_2^2 + \nu^T z\right) &= \frac{1}{2}\|\nu\|_2^2 - \|\nu\|_2^2 \\ &= -\frac{1}{2}\|\nu\|_2^2 \end{aligned} \tag{8}$$

1.2.2 Study of $\inf_w \left(\lambda\|w\|_1 - \nu^T X w\right)$

$$\begin{aligned} \inf_w \left(\lambda\|w\|_1 - \nu^T X w\right) &= \lambda \inf_w \left(\|w\|_1 - \frac{\nu^T X w}{\lambda}\right) \\ &= -\lambda \sup_w \left(\frac{\nu^T X w}{\lambda} - \|w\|_1\right) \\ &= -\lambda \sup_w \left((\frac{X^T \nu}{\lambda})^T w - \|w\|_1\right) \end{aligned} \tag{9}$$

From exercise 2 of the Homework 2, "Regularized Least-Square", we encountered a similar situation. To briefly recall, the conjugate of a function f is $f^*(y) = \sup_{x \in dom f}(y^T x - f(x))$. In our case, if we define $u = \frac{X^T \nu}{\lambda}$, we have that 9 becomes $-\lambda \sup_w \left(u^T w - \|w\|_1\right)$.

Then, we have that:

$$\inf_w \left(\lambda\|w\|_1 - \nu^T X w\right) = \begin{cases} 0 & \text{if } \|X^T \nu\|_\infty \leq \lambda, \\ -\infty & \text{otherwise} \end{cases} \tag{10}$$

1.2.3 Conclusion

Finally we have for the **Lagrangian dual function** (combining the two previous results):

$$\begin{aligned} g(\nu) &= \begin{cases} -\frac{1}{2}\|\nu\|_2^2 + \nu^T y & \text{if } \|X^T \nu\|_\infty \leq \lambda, \\ -\infty & \text{otherwise} \end{cases} \\ &= \begin{cases} -\frac{1}{2}\|\nu\|_2^2 + \nu^T y & \text{if } \|X^T \nu\|_\infty \leq \lambda, \\ -\infty & \text{otherwise} \end{cases} \end{aligned} \tag{11}$$

In addition, g is a quadratic function with negative coefficients on a convex set: $\{\nu \in \mathbb{R}^n \mid \|X^T \nu\|_\infty < \lambda\}$. We can then conclude that g is a concave function.

1.3 Dual Problem

The dual problem can then be written as follows:

$$\begin{aligned} \text{maximize} \quad & -\frac{1}{2}\|\nu\|_2^2 + \nu^T y \\ \text{subject to} \quad & \|X^T\nu\|_\infty \leq \lambda \end{aligned} \tag{12}$$

Now, we have to format is as a general Quadratic Problem.

$$\begin{aligned} \text{Dual Problem} &\longleftrightarrow \begin{cases} \text{maximize} & -\frac{1}{2}\|\nu\|_2^2 + \nu^T y \\ \text{subject to} & \|X^T\nu\|_\infty \leq \lambda \end{cases} \\ &\longleftrightarrow \begin{cases} \text{minimize} & \frac{1}{2}\|\nu\|_2^2 - \nu^T y \\ \text{subject to} & \|X^T\nu\|_\infty \leq \lambda \end{cases} \end{aligned} \tag{13}$$

We can then look at the inequality condition.

$$\|X^T\nu\|_\infty \longleftrightarrow \begin{pmatrix} X^T \\ -X^T \end{pmatrix} \cdot \nu \leq \lambda \cdot I_{2d} \tag{14}$$

Let's consider these variables:

$$\begin{aligned} Q &= \frac{1}{2} \cdot I_n \\ p &= -y \\ A &= \begin{pmatrix} X^T \\ -X^T \end{pmatrix} \\ b &= \lambda \cdot I_{2d} \end{aligned} \tag{15}$$

We can finally conclude that to derive the dual problem of LASSO is equivalent to solve the general quadratic problem (QP).

$$\begin{aligned} \text{minimize} \quad & v^T Q v + p^T v \\ \text{subject to} \quad & Av \leq b \end{aligned}$$

2 Implement the barrier method to solve QP

2.1 LASSO problem

The LASSO Problem is defined as :

$$\text{minimize} \quad \frac{1}{2}\|Xw - y\|_2^2 + \lambda\|w\|_1 \tag{16}$$

where X are the data inputs, y are the observations and λ is a regularization parameter.
We have demonstrated in Q1 of the Homework that when we define:

$$\begin{aligned} Q &= \frac{\mathbf{l}_n}{2} \\ A &= \begin{pmatrix} X^T \\ -X^T \end{pmatrix} \\ p &= -y \\ b &= \lambda \cdot \mathbf{1}_2 n \end{aligned}$$

Solving Lasso problem is equivalent to find a solution to the Quadratic Problem. Therefore, let's see what are the steps needed to solve the QP.

2.2 Quadratic Problem

The Quadratic Problem is defined as follows:

$$\begin{aligned} \min_{v} \quad & v^T Q v + p^T v \\ \text{subject to} \quad & Av \leq b \end{aligned}$$

$$v \in \mathbb{R}^n, \text{ where } Q \succeq 0.$$

The objective of the exercise is to implement the barrier method to solve QP.

2.3 Barrier Method

The Barrier Method is used to solve inequality constrained minimization. Given strictly feasible x, $t := t_0 > 0$, $\mu > 1$, tolerance $\epsilon > 0$.

Repeat

1. Centering step. Compute $x^*(t)$ by minimizing $tf_0 + \Phi$, subject to $Ax = b$.
2. Update. $x := x^*(t)$.
3. Stopping criterion. Quit if $\frac{m}{t} < \epsilon$.
4. Increase t. $t := \mu t$.

Therefore, the first step is the step that is required to be built with Newton's method.

2.4 Centering Problem

The centering problem is the following:

$$\text{minimize} \;\; tf_0(x) - \sum_{i=1}^{m} log(-f_i(x))$$

It is an unconstrained minimization problem so it can be solved with the Newton's method.

2.5 Newton's Method

Given a starting point x in dom$f$, tolerance $\epsilon > 0$.
Repeat

1. Compute the Newton step and decrement. $\Delta x_{\text{nt}} := -\nabla^2 f(x)^{-1}\nabla f(x)$;ibr¿ $\lambda^2 := \nabla f(x)^T \nabla^2 f(x)^{-1}\nabla f(x)$.
2. Stopping criterion. Quit if $\frac{\lambda^2}{2} \leq \epsilon$.
3. Line search. Choose step size t by backtracking line search.
4. Update. $x := x + t\Delta x_{\text{nt}}$.

2.6 Backtracking Line Search

For this problem, we have two parameters, $\alpha \in (0, 1/2)$ and $\beta \in (0, 1)$.

As explained in the lecture, we start at $t = 1$ and we repeat $t := \beta t$ until :

$f(x + t\Delta x) < f(x) + \alpha t \nabla f(x)^T \Delta x$

2.7 Problem statement

In our exercise, we have that:

- $f_0(x) = v^T Q v + p^T v$, $f_0(x)$ being the objective or cost function.

- $f_i(x) = Av - b$, $f_i(x)$ being the inequality constraint functions

Therefore in the centering problem we have to minimize this function that we can call $f(x)$:

$f(x) = t \cdot (v^T Q v + p^T v) - \sum_{i=1}^{m} log(-(Av - b))$

2.8 Scripts and Results

All the scripts and results are below.

Convex Optimization - Homework 3

Matteo MARENGO - matteo.marengo@ens-paris-saclay.fr

Date: 20 / 11 / 2023

Implement the barrier method to solve QP

```
# Import necessary libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import cvxpy as cp
from sklearn.datasets import make_regression

# Parameters for the Barrier method
# tolerance eps > 0
eps = 1e-4
t0 = 1

# Parameters for the Newton method
eps_newton = 1e-5

# Parameters for the backtracking line search
# alpha in [0, 0.5]
alpha  = 0.01
# beta in [0, 1]
beta = 0.5

# List of values for the parameter mu
# mu > 1
# large mu means fewer outer iterations but more inner (Newton)
iterations
mu_list = [2, 10, 15, 25, 50, 100, 200,500,600]

# Parameters for the Ridge regression
n = 75
d = 150
regularization_lambda = 10
```

1 - Write a function v_seq = centering_step(Q,p,A,b,t,v0,eps)

1.a - Define the functions

```
def objectf(Q,p,v):
    quadratic_part = v.T @ Q @ v
    linear_part = p.T @ v
    return quadratic_part + linear_part

def ftotal(Q,p,A,b,t,v):
    Av = A @ v
    constraint_value = Av - b
    if np.any(constraint_value >= 0):
        return None
    return (t * objectf(Q, p, v) - np.sum(np.log(-constraint_value)))

def gradient_f (Q, p, A, b, t, v) :
    Qv = Q @ v
    Av = A @ v
    return t * (2*Qv + p) + A.T @ (1 / (b - Av))

def hessian_f (Q, p, A, b, t, v) :
    Av = A @ v
    aux  = (1 /((b - Av)**2)).reshape(-1)
    sted = -A.T @ np.diag(aux) @ A
    return t * (2*Q) - sted
```

1.b - Backtracking Line Search

```
def comparison_criterion(Q,p,A,b,t,v,dv):
    value_of_f = ftotal(Q,p,A,b,t,v)
    if value_of_f != None :
        cond2 = value_of_f + alpha * t * np.dot((gradient_f(Q, p, A,
b, t, v)).T, dv)
        # If the Armijo condition is satisfied
        return value_of_f >= cond2
    return False

def backtrack_line_search (Q, p, A, b, t, v, dv) :
    # start at t = 1
    t_update = 1
    while True :
        v_update = v + t_update * dv
        # check if the new point is feasible
        if not comparison_criterion(Q,p,A,b,t,v_update,dv):
            break
        # update t
```

```
        t_update = t_update * beta
    return t_update
```

1.c - Centering Step | Newton's Method

```
def lambda_and_dv_compute(Q, p, A, b, t, v) :
    grad = gradient_f(Q, p, A, b, t, v)
    hess = hessian_f(Q, p, A, b, t, v)
    # Compute the Newton step
    dv = - np.linalg.solve(hess, grad)
    # Compute the Newton decrement
    lmbda_2 = - grad.T @ dv
    return lmbda_2,dv

def centering_step(Q, p, A, b, t, v0, eps) :
    v_seq = [v0]
    v_upd = v0
    lmbda_2,dv  = lambda_and_dv_compute(Q, p, A, b, t, v_upd)
    # Stopping criterion
    while lmbda_2 / 2 > eps :
        # Line Search: Choose step stize by backtracking line search
        step_size = backtrack_line_search (Q, p, A, b, t, v_upd, dv)

        # Update the solution
        v_upd = v_upd + step_size * dv
        v_seq.append(v_upd)

        lmbda_2,dv  = lambda_and_dv_compute(Q, p, A, b, t, v_upd)

    return v_seq
```

---

2 - Write a function v_seq = barr_method(Q,p,A,b,v0,eps)

```
def barr_method (Q, p, A, b, v0, eps) :

    num_cons = A.shape[0] # number of constraints
    v_upd = v0 # initial point
    t = t0
    v_seq = [v0]

    # Stopping criterion
    while (num_cons/t)>= eps :
        # STEP 1
        # Centering step
        v_upd = centering_step(Q, p, A, b, t, v_upd, eps_newton)

        # STEP 2
```

```
        # Update the solution
        v_upd = v_upd[-1] # take the last element of the list
        v_seq.append(v_upd)

        # STEP 4
        # Increase the value of t
        t *= mu

    return v_seq
```

---

3 - Test the Barrier Method

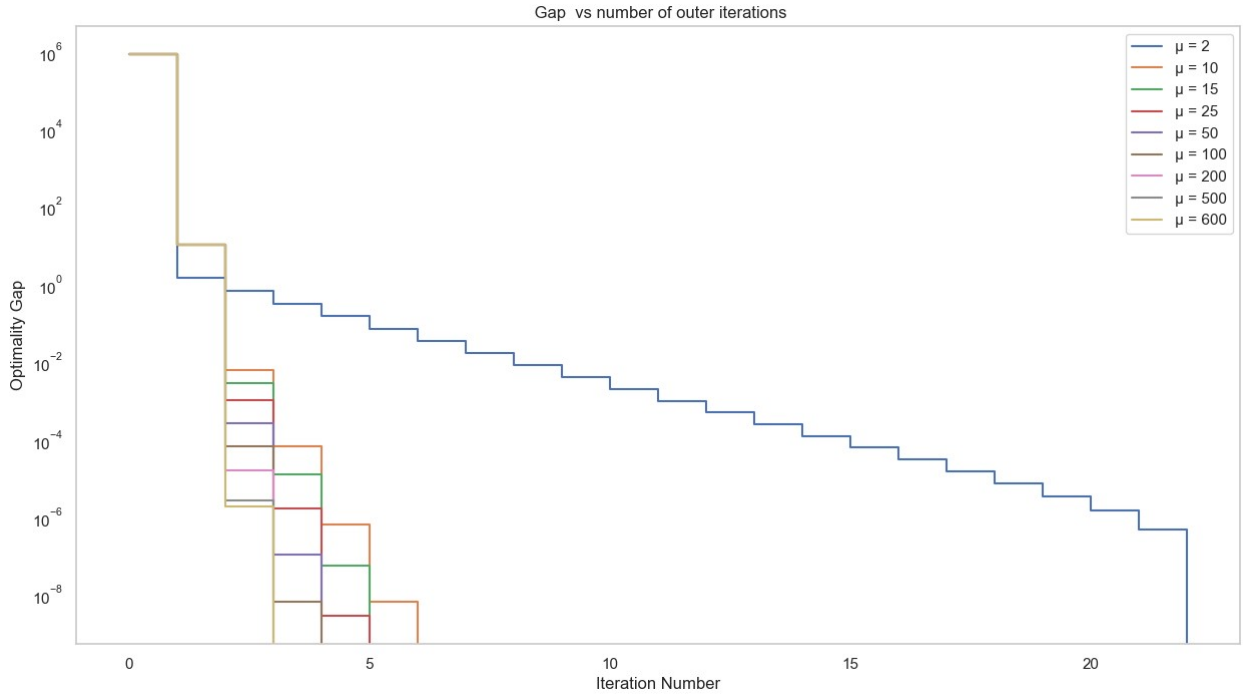
3.a - Generate random matrices & Variables for QP

```
# Random Generated Matrices X and observations y and weights w for the
regressor
X, y, w_true = make_regression( n_samples = n , n_features = d,
coef=True, random_state=123,noise=5)
y = y.reshape(-1 , 1)
# w_true = w_true.reshape(-1 , 1)

# Define the matrices for the optimization problem
Q = 0.5*np.eye(n)
A = np.vstack((X.T , -X.T))
p = - y
b = regularization_lambda * np.ones((2*d,1))
v0 = np.zeros((n,1))
```

3.b - Duality Gap

```
plt.figure(figsize=(15,8))

for mu in mu_list :
    # Run the barrier method
    # Output: list of v for each iteration
    v_seq = barr_method (Q, p, A, b, v0, eps)
    gap = [objectf(Q, p, v) - objectf(Q, p, v_seq[-1]) for v in v_seq]
    plt.step(range(len(gap)), np.squeeze(gap), where='post')

plt.grid()
# Setting the scale to logarithmic
plt.semilogy()

# Adding labels, title and legend
plt.xlabel("Iteration Number")
plt.ylabel("Optimality Gap")
plt.legend(list(map(lambda mu: f'μ = {mu}', mu_list)))
```

```
plt.title(' Gap  vs number of outer iterations')
plt.show()
```



Observations

- This figure, is the gap f - f* in semilog scale compared to the number of outer iterations. What we observe is that for every value of mu the gap is decreasing meaning that the optimization method is converging towards the optimal value as iterations increase. When mu is small there are a lot of outer iterations to achieve the optimization point whereas with large mu it converges rapidly. Indeed, it appears that higher values of mu result in a faster initial decrease in the gap, with the gap reaching lower values more quickly within the first few outer iterations.

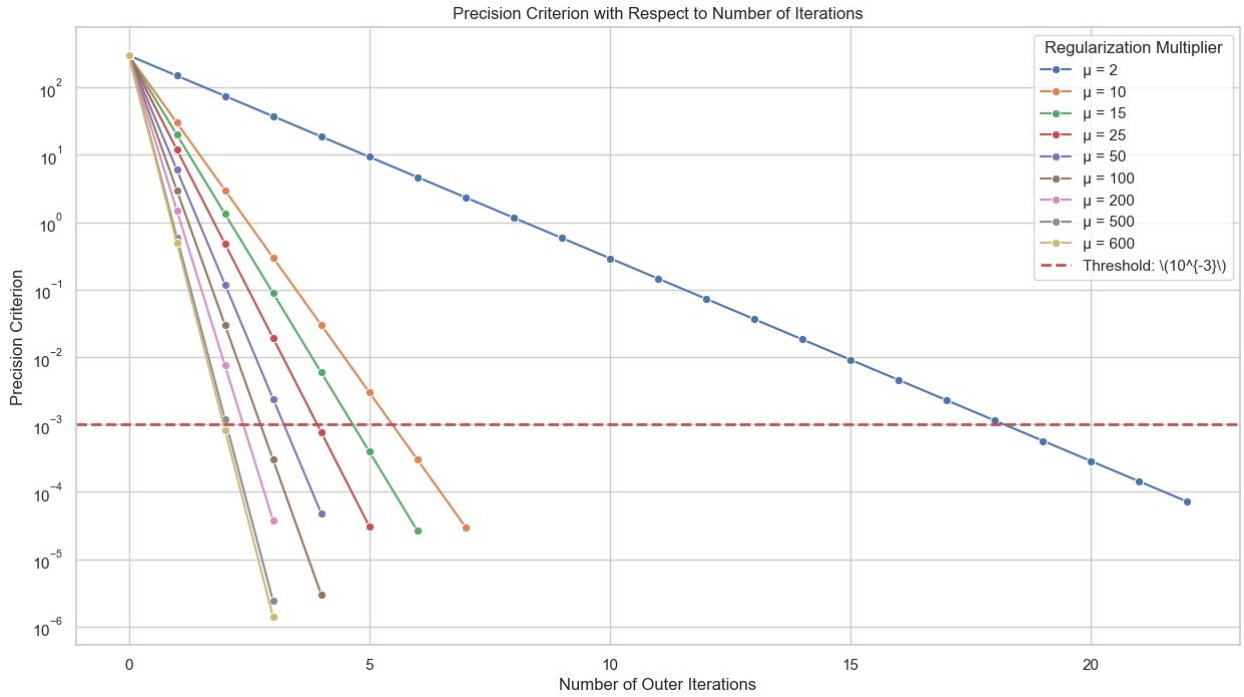
- However, after this initial decrease the rate of convergence slows down. In contrast, lower values of mu start with lower gap but a convergence rate that is smaller, therefore the steps are more cautious. It is then important to find a trade off between these two values.

- Something higher than 30 but smaller than 150 for mu might be interesting to consider.

- In the lectures, this duality gap is often plotted vs the newton iterations, this is also something that can be done, this would be the inner itterations, the trend is expected to be the same.

3.c - Precision criterion

```
# Set the aesthetic style of the plots
sns.set_theme(style="whitegrid")

# Initialize the figure
plt.figure(figsize=(15, 8))

# Plot the precision criterion for each value of mu
for mu in mu_list:
    v_seq = barr_method(Q, p, A, b, v0, eps)
    precision_criterion = [len(A) / (mu ** i) for i in
range(len(v_seq))]
    sns.lineplot(x=range(len(precision_criterion)),
y=precision_criterion, marker='o', label=f'μ = {mu}')

# Set the y-axis to a logarithmic scale
plt.yscale('log')

# Add a horizontal dotted line at y = 10^-3
plt.axhline(y=1e-3, color='r', linestyle='--', linewidth=2,
label='Threshold: \(10^{-3}\)')

# Customize the axes and title
plt.xlabel("Number of Outer Iterations")
plt.ylabel("Precision Criterion")
plt.title('Precision Criterion with Respect to Number of Iterations')

# Show the legend
plt.legend(title='Regularization Multiplier')

# Show the plot
plt.show()
```



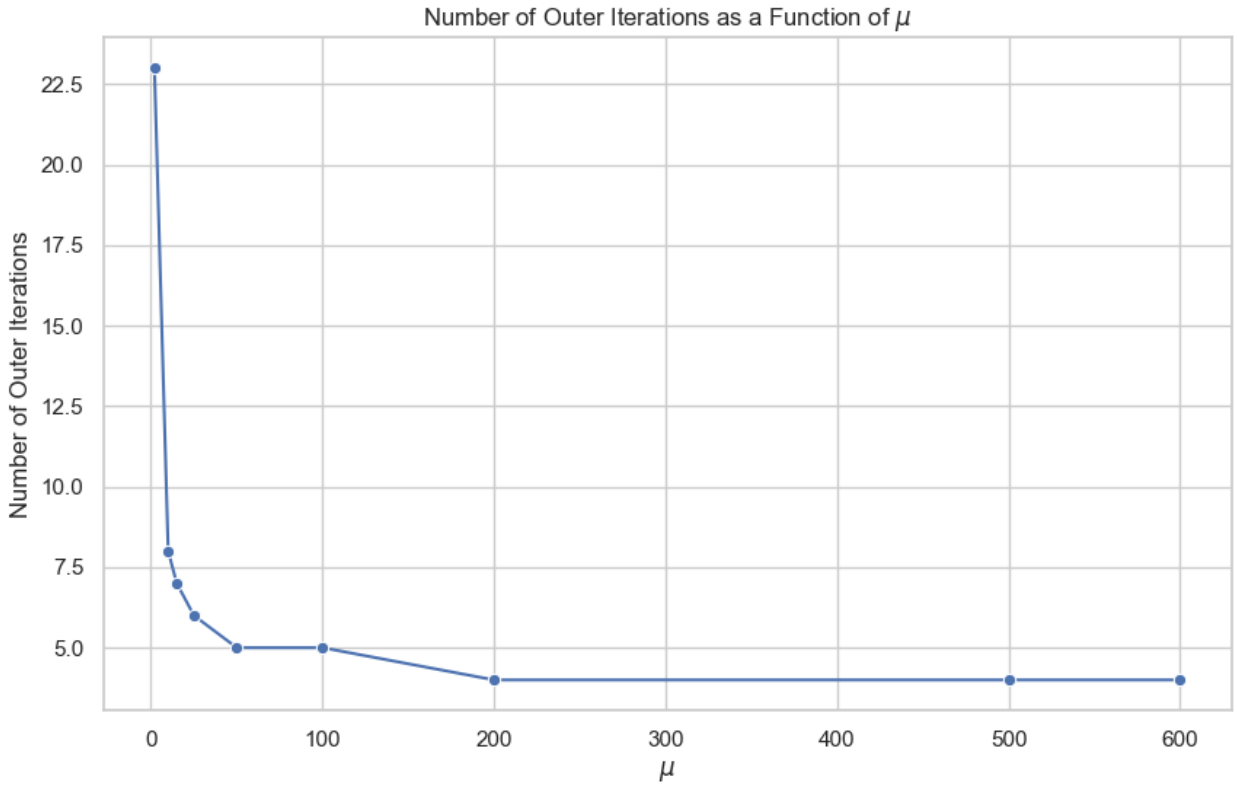
Observations

- We defined previously our eps as 1e-4. What we observe is that as for the gap, there is indeed a convergence of the optimization methods towards this value.

3.d - Outer iterations vs mu

```
# Set the aesthetic style of the plots
sns.set_theme(style="whitegrid")

# Prepare the data for plotting
num_iterations = []  # To store the number of iterations for each mu
for mu in mu_list:
    v_seq = barr_method(Q, p, A, b, v0, eps)
    num_iterations.append(len(v_seq))  # Number of iterations for this
mu

# Create a DataFrame to hold the values
data = pd.DataFrame({'$\mu$': mu_list, 'Number of Outer Iterations':
num_iterations})

# Now plot mu vs. the number of outer iterations using seaborn
plt.figure(figsize=(10, 6))
sns.lineplot(x='$\mu$', y='Number of Outer Iterations', data=data,
marker='o')

# Customize the axes and title
plt.xlabel('$\mu$')
plt.ylabel('Number of Outer Iterations')
plt.title('Number of Outer Iterations as a Function of $\mu$')
```

```
# Show the plot
plt.show()
```



Observations

- The number of outer iterations required decreases sharply as mu increases from 0 to 50. That suggests that a higher mu results in fewer outer iterations in the range observed. If mu is low, the number of outer iterations is high, the algorithm takes smaller steps to find the optimum. As mu increases, the algorithm needs less steps, it is more efficient. Beyond a certain point (mu ≈ 50), the curve becomes relatively flat, with the number of outer iterations varying only slightly between approximately 4.5 and 5.5.
- This plateau suggests that there is a limit to the effectiveness of increasing mu in terms of reducing the number of outer iterations. Therefore, based on the graph, there seems to be an optimal range of mu (around 50-200) where the number of outer iterations stabilizes. Operating within this range of mu values might provide a balance between computational efficiency and the precision of the steps taken by the algorithm.

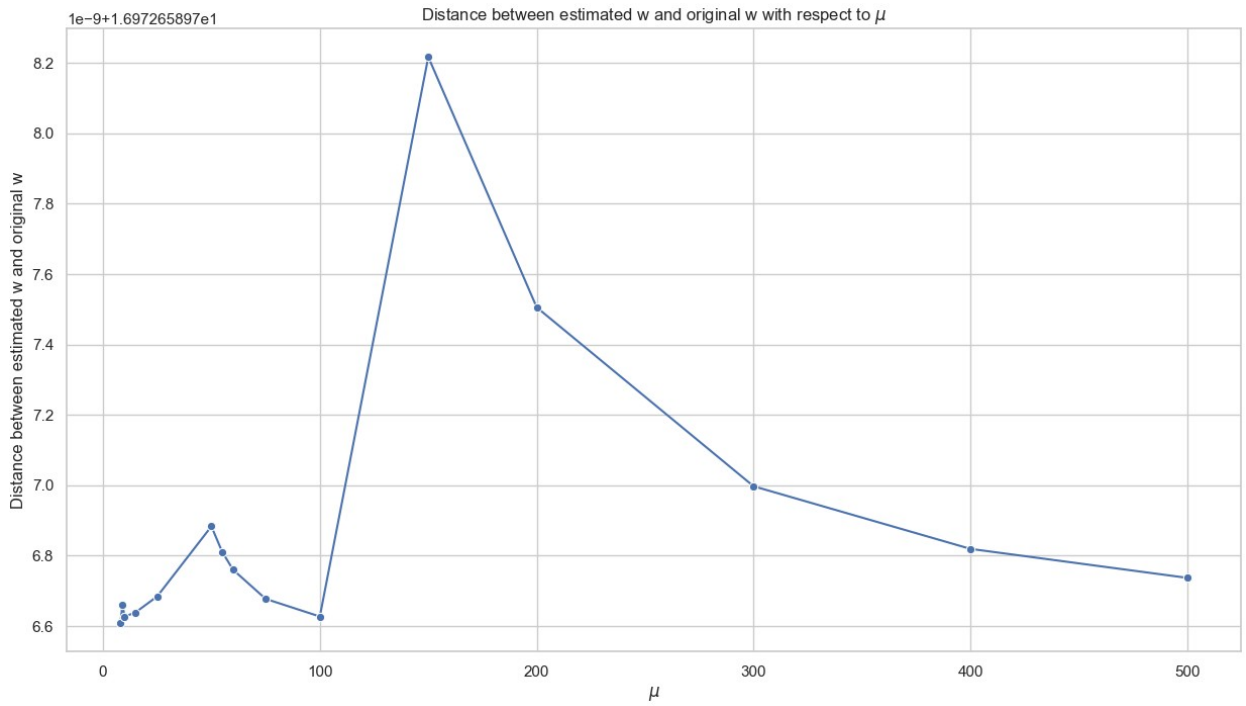
3.e - Evolution of w

```
# repeat on different values of mu
# check the inpact on w
```

```
mu_list = [8,9,10, 15,25, 50,55,60,75,100, 150,200,300,400,500]
distances = []
solutions = []

for mu in mu_list :

    v_seq  = barr_method (Q, p, A, b, v0, eps)
    # After solving KKT
    # gradient of the Lagrangian respect to z = 0
    # z* = nu *
    # Xw* - y = nu*
    # w* = (X.TX)^-1 X.T(- y - nu*)

    w_est = (np.linalg.pinv(X) @ (y + v_seq[-1])) /
(np.linalg.norm(np.linalg.pinv(X) @ (y + v_seq[-1])))
    w_norm = w_true / np.linalg.norm(w_true)
    w_est_r = (np.linalg.pinv(X) @ (y + v_seq[-1]))
    distance = np.linalg.norm(w_est - w_norm)
    distances.append(distance)
    print(f"Distance for μ={mu}: {distance}")
    # Solution of the problem
    # 0.5 * ||y - Xw||^2 + lambda * ||w||_1
    # give me the python script for the solution
    solution = 0.5 * np.linalg.norm(y - X @ w_est_r)**2 +
regularization_lambda * np.linalg.norm(w_est_r, 1)
    solutions.append(solution)
    print(f"Solution for μ={mu}: {solution}")
    print("")

# Convert the lists to a DataFrame
data = pd.DataFrame({'mu': mu_list, 'Distance': distances})
results = pd.DataFrame({'mu': mu_list, 'Distance': distance,
'Solution': solutions})

# Plot using seaborn
sns.set_theme(style="whitegrid")
plt.figure(figsize=(15, 8))
sns.lineplot(x='mu', y='Distance', data=data, marker='o')

# Labeling the plot
plt.ylabel("Distance between estimated w and original w")
plt.xlabel('$\mu$')
plt.title('Distance between estimated w and original w with respect to
$\mu$')

# Display the plot
plt.show()
```

```
# display results
results
```

```
Distance for μ=8: 16.972658976607757
Solution for μ=8: 852179.7842404708

Distance for μ=9: 16.972658976659176
Solution for μ=9: 852179.7846625737

Distance for μ=10: 16.972658976626906
Solution for μ=10: 852179.7843976622

Distance for μ=15: 16.972658976638503
Solution for μ=15: 852179.7844928706

Distance for μ=25: 16.972658976684
Solution for μ=25: 852179.784866385

Distance for μ=50: 16.972658976883114
Solution for μ=50: 852179.7865009058

Distance for μ=55: 16.972658976810294
Solution for μ=55: 852179.7859031388

Distance for μ=60: 16.972658976759753
Solution for μ=60: 852179.7854882329

Distance for μ=75: 16.972658976677057
Solution for μ=75: 852179.7848094081

Distance for μ=100: 16.972658976626906
Solution for μ=100: 852179.7843976619

Distance for μ=150: 16.9726589782169
Solution for μ=150: 852179.7974511241

Distance for μ=200: 16.97265897750529
Solution for μ=200: 852179.7916087824

Distance for μ=300: 16.97265897699698
Solution for μ=300: 852179.7874356804

Distance for μ=400: 16.97265897681906
Solution for μ=400: 852179.7859750948

Distance for μ=500: 16.97265897673671
Solution for μ=500: 852179.7852990524
```



	mu	Distance	Solution
0	8	16.972659	852179.784240
1	9	16.972659	852179.784663
2	10	16.972659	852179.784398
3	15	16.972659	852179.784493
4	25	16.972659	852179.784866
5	50	16.972659	852179.786501
6	55	16.972659	852179.785903
7	60	16.972659	852179.785488
8	75	16.972659	852179.784809
9	100	16.972659	852179.784398
10	150	16.972659	852179.797451
11	200	16.972659	852179.791609
12	300	16.972659	852179.787436
13	400	16.972659	852179.785975
14	500	16.972659	852179.785299

Observations

- The distance between the estimated w and the original w is almost always the same, meaning that mu value will always give the same convergence result.

3.f - Find the optimal value with cvxpy

```
w = cp.Variable((d,1))
objec = cp.Minimize(0.5*cp.sum_squares(X @ w - y) +
regularization_lambda*cp.norm(w,1))
prob = cp.Problem(objec)
```

```
result = prob.solve()
print("Optimal value",result)
```

```
Optimal value 4786.621570636406
```

Observations

- What we observe is that the optimal value can also be given by cvxpy.
- To conclude, with all the observations, we can say that the barrier method is a working method and that we have to be aware on which parameters to choose, mu can be choosen between the range [40 - 120] based on our observations.
- This homework has been an interessting exploration into the field of constrained optimization. It has shown its efficiency in handling optimization problems with inequality constraints. Throughout this practical, we observed how the method went through the feasible region by incorporating barrier functions into the objective function.
