Convex Optimization - Homework 3

Yassine ZAOUI

December 2, 2024

Question 1

Remark: We can get inspired by the Exercise 2 in HW2, thus, we go a little faster over the calculations.

Starting from the LASSO optimization problem, we define the substitution $u = Xw - y$, with the dual variable v, to formulate the Lagrangian:

$$L(w, u, v) = \frac{1}{2}\|u\|_2^2 + \lambda\|w\|_1 + v^T(Xw - y - u).$$

Maximizing the Lagrangian with respect to w and u, we obtain the dual function:

$$g(v) = -\frac{1}{2}\|v\|_2^2 - y^T v - Ind\left(\frac{X^T v}{\lambda}\right),$$

where $Ind(x)$ is the indicator function defined as:

$$Ind(x) = \begin{cases} 0, & \text{if } \|x\|_\infty \leq 1, \\ +\infty, & \text{otherwise.} \end{cases}$$

This leads us to the dual formulation of the LASSO problem:

$$\max_v -\frac{1}{2}\|v\|_2^2 - y^T v,$$

subject to the constraint:

$$\|X^T v\|_\infty \leq \lambda \quad \Leftrightarrow \quad \begin{bmatrix} X^T \\ -X^T \end{bmatrix} v \leq \begin{bmatrix} \lambda \\ \lambda \end{bmatrix}.$$

Finally, the equivalent quadratic program is expressed as:

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

with the following components:

$$Q = \frac{1}{2} I_n \geq 0, \quad p = y, \quad A = \begin{bmatrix} X^T \\ -X^T \end{bmatrix}, \quad b = \begin{bmatrix} \lambda \\ \vdots \\ \lambda \end{bmatrix}.$$

Question 2

Before diving to the implementation, we recall the expression of the objective function as well as its gradient and Hessian in the case of the approximation of the quadratic program using a logarithmic barrier:

$$f(v) = t\left(v^T Q v + p^T v\right) - \sum_{i=1}^{m} \log\left(b_i - a_i^T v\right).$$

The gradient of $f(v)$ is:

$$\nabla f(v) = 2tQv + tp + \sum_{i=1}^{m} \frac{a_i}{b_i - a_i^T v}.$$

The Hessian of $f(v)$ is:

$$\nabla^2 f(v) = 2tQ + \sum_{i=1}^{m} \frac{a_i a_i^T}{\left(b_i - a_i^T v\right)^2}.$$

For the rest of the Homework, please check the notebook.

```
!pip install cvxpy
```

```
Requirement already satisfied: cvxpy in c:\users\yassi\anaconda3\envs\yourenvname\lib\site-packages (1.6.0)
Requirement already satisfied: osqp>=0.6.2 in c:\users\yassi\anaconda3\envs\yourenvname\lib\site-packages (from cvxpy) (0.6.7.post3
Requirement already satisfied: clarabel>=0.5.0 in c:\users\yassi\anaconda3\envs\yourenvname\lib\site-packages (from cvxpy) (0.9.0)
Requirement already satisfied: scs>=3.2.4.post1 in c:\users\yassi\anaconda3\envs\yourenvname\lib\site-packages (from cvxpy) (3.2.7)
Requirement already satisfied: numpy>=1.20 in c:\users\yassi\anaconda3\envs\yourenvname\lib\site-packages (from cvxpy) (1.26.4)
Requirement already satisfied: scipy>=1.1.0 in c:\users\yassi\anaconda3\envs\yourenvname\lib\site-packages (from cvxpy) (1.13.1)
Requirement already satisfied: qdldl in c:\users\yassi\anaconda3\envs\yourenvname\lib\site-packages (from osqp>=0.6.2->cvxpy) (0.1.7
```

We start by importing necessary python libraries for the homework as follows:

```
# Import libraries
import numpy as np
import matplotlib.pyplot as plt
import cvxpy as cp
import time
```

We use the time library to have each time different data

```
np.random.seed(int(time.time()))
```

Question 2

For this question, we define the centrering_step and barr_method functions for the log-barrier method as well as the objective functions as well as the gradient and Hessian of the log_barrier approximation objective function.

```
# Define the functions
def f_0(Q, p, v):
    """Compute the quadratic objective f(v) = v^T Q v + p^T v"""
    return v @ Q @ v + p @ v

def f(Q, p, A, b, v, t):
    """Log-barrier approximation of the objective function."""
    Av = A @ v
    if np.any(b - Av <= 0):
        return np.inf  # Handle infeasibility
    return t * (v @ Q @ v + p @ v) - np.sum(np.log(b - Av))

def grad(Q, p, A, b, v, t):
    """Compute the gradient of the log-barrier function."""
    Qv = Q @ v
    Av = A @ v
    z = 1 / (b - Av)
    return 2 * t * Qv + t * p + A.T @ z

def H(Q, p, A, b, v, t):
    """Compute the Hessian of the log-barrier function."""
    Av = A @ v
    z = (1 / (b - Av))**2
    s = np.einsum("ij,ik,i->jk", A, A, z)  # Efficient Hessian computation
    return 2 * t * Q + s

def centering_step(Q, p, A, b, t, v0, eps, max_iters=50):
    """Centering step for the barrier method."""
    V = [v0]
    v = np.copy(v0)
    alpha = 0.1
    beta = 0.7
    for _ in range(max_iters):
        # Compute gradient and Hessian
        g = grad(Q, p, A, b, v, t)
        H_inv = np.linalg.pinv(H(Q, p, A, b, v, t))
        delta_v = -H_inv @ g

        # Newton decrement
        decrement = -g @ delta_v / 2
        if decrement < eps:  # Convergence check
            break

        # Backtracking line search
        u = 1
        while f(Q, p, A, b, v + u * delta_v, t) > f(Q, p, A, b, v, t) - alpha * u * g @ delta_v:
            u *= beta
```

```
        # Update
        v += u * delta_v
        V.append(v)
    return V

def barr_method(Q, p, A, b, v0, eps, mu=10, max_iters=50):
    """Barrier method for solving the quadratic program."""
    V = [v0]
    t = 1  # Initial barrier parameter
    for _ in range(max_iters):
        if A.shape[0] / t < eps: # the stopping criterion m/t < epsilon
            break
        v_new = centering_step(Q, p, A, b, t, V[-1], eps)
        V.extend(v_new)
        t *= mu  # Update the barrier parameter
    return V
```

Question 3

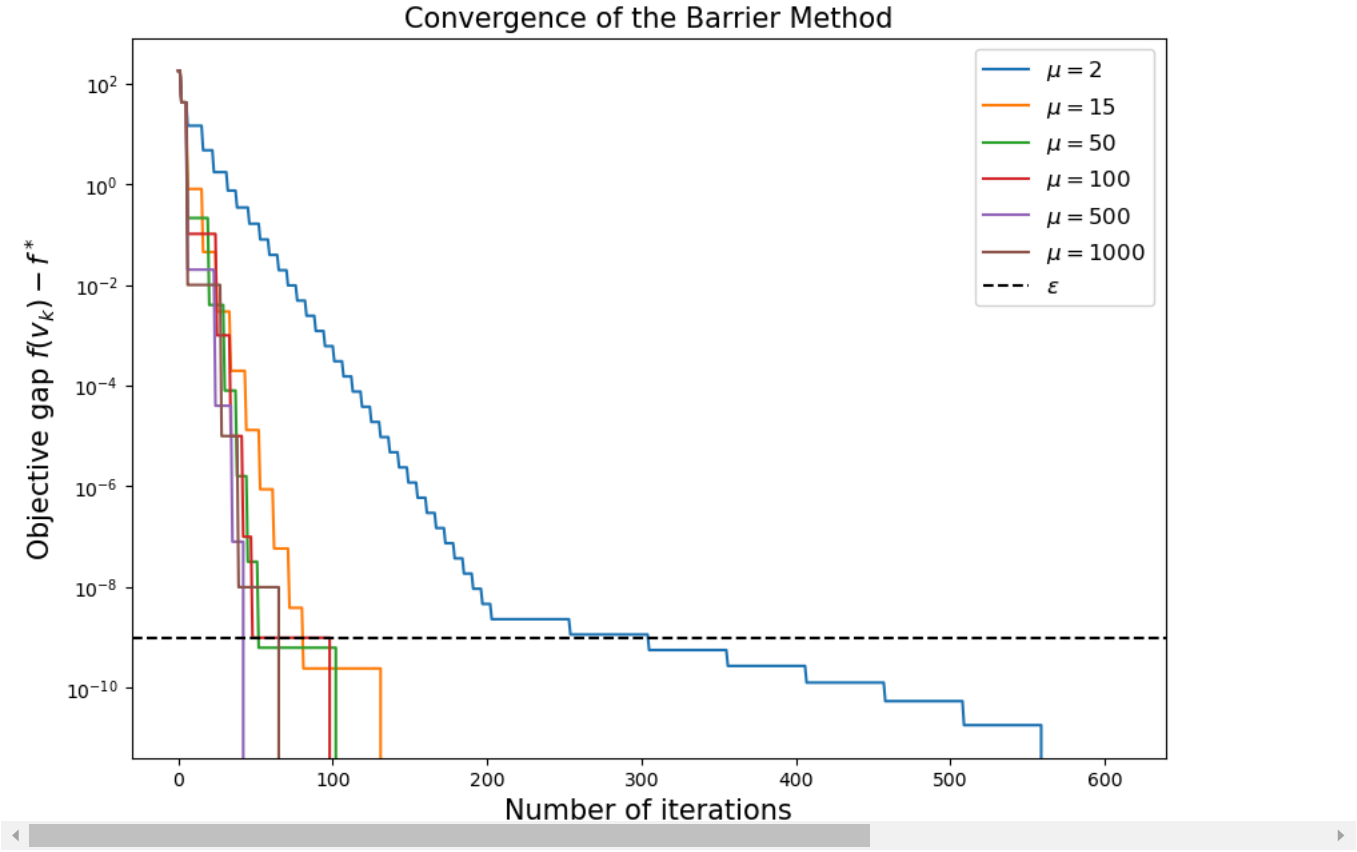
We start by defining our parameters and generate random matrices X and observations y. We compute as well the corresponding QP parameters (Q, p, A, b) and choose arbitrary the zero matirx as the initialization for the algorithm as well as the minimal accepted precision epsilon.

```
# Generate parameters and data
Lambda = 10
d = 500
n = 10
X = np.random.normal(size=(n, d))  # Feature matrix
y = np.random.normal(size=n) + 15  # Observations

# Define QP components
Q = np.eye(n) / 2
p = y
A = np.vstack([X.T, -X.T])
b = Lambda * np.ones(2 * d)

#Initialization and precision
v0 = np.zeros(n)
eps = 1e-9
```

Then, we apply the log_barrier method for different values of mu in [2,15,50,50,10,500,1000]. We plot as well the convergence of the method for each value of mu.

```
# Barrier method with different mu values
fig, ax = plt.subplots(figsize=(10, 7))
for mu in [2, 15, 50, 100,500,1000]:
    V = barr_method(Q, p, A, b, v0, eps, mu)
    x = [f_0(Q, p, v) for v in V]
    ax.plot(np.array(x) - np.min(x), label=f"$\mu = {mu}$")
    print(f"mu = {mu}, iterations = {len(V)}, optimal value = {f_0(Q, p, V[-1]):.6f}")
# Plot convergence of the barrier method
ax.set_yscale("log")
ax.set_xlabel("Number of iterations", fontsize=15)
ax.set_ylabel("Objective gap $f(v_k) - f^*$", fontsize=15)
ax.set_title("Convergence of the Barrier Method", fontsize=15)
ax.axhline(eps, color="k", linestyle="--", label="$\epsilon$")
ax.legend(fontsize=12)
plt.show()
```

```
mu = 2, iterations = 610, optimal value = -179.234121
mu = 15, iterations = 182, optimal value = -179.234121
mu = 50, iterations = 153, optimal value = -179.234121
mu = 100, iterations = 149, optimal value = -179.234121
mu = 500, iterations = 93, optimal value = -179.234121
mu = 1000, iterations = 116, optimal value = -179.234121
```



According to the plot, as expected, as μ increases, the number of needed iterations decreases drastically since with bigger μ values, t tends faster to ∞. We need to pinpoint, the total iterations depend on the balance between outer iterations (updates to t) and the number of centering steps required at each t. Large μ values reduce the number of outer iterations (fewer updates to t), but increase the complexity of each centering step. Thus, taking all of this into account, it sees plausible to say that for $\mu \in \{50, 100\}$ we have the rather satisfying compromise between convergence rate and computational compelxity.

Remark: For $\mu = 2$, for some random initialization, we find its optimal value slightly different at fourth decimal digit.

Just to be sure of the optimal value and that the approximation $t \to \infty$ gives us indeed the initial problem, let's see the optimal value using cp library

```
# Solve the problem using CVXPY for comparison
v = cp.Variable(n)
qp_prob = cp.Problem(cp.Minimize(cp.quad_form(v, Q) + p.T @ v), [A @ v <= b])
f_star = qp_prob.solve()
print(f"Optimal value (CVXPY): {f_star:.6f}")
print(f"Optimal point (CVXPY): {v.value}")
```

```
Optimal value (CVXPY): -179.234121
Optimal point (CVXPY): [-1.86305781 -1.02584351 -1.21457526 -1.46414434 -1.39939465 -1.12953611
 -0.88918431 -1.10977058 -1.32130842 -1.48897359]
```

So, it looks like we did implement well the needed method since we found the exact same value at least for a 10^{-6} precision.

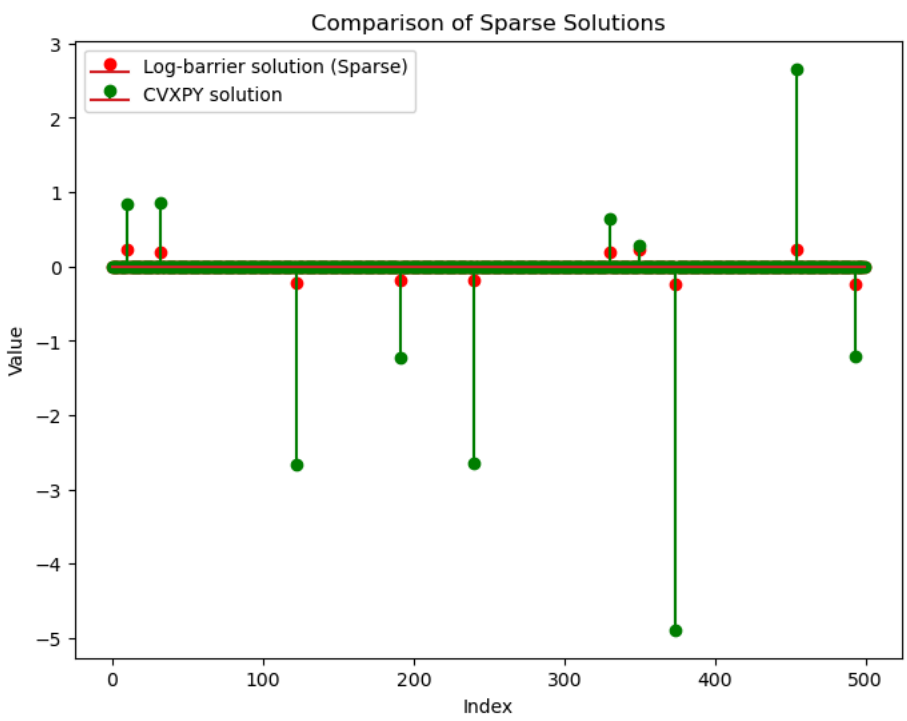
Now, let's recover the sparse w from the log-barrier solution so that we can compare its sparsity with the solution we found.

```
# Recover sparse w from log-barrier solution
signs = np.zeros(d)
signs[np.isclose(X.T @ V[-1], Lambda)] = -1
signs[np.isclose(X.T @ V[-1], -Lambda)] = +1
w_star = signs * np.abs(np.linalg.pinv(X) @ (V[-1] + y))

# Solve the primal LASSO using CVXPY for direct comparison
w = cp.Variable(d)
lasso_prob = cp.Problem(cp.Minimize(cp.norm2(y - X @ w)**2 + Lambda * cp.norm1(w)))
```

```
lasso_prob.solve(solver=cp.SCS)
print(f"Optimal value (LASSO, CVXPY): {lasso_prob.value:.4f}")
```

```
Optimal value (LASSO, CVXPY): 183.5830
```

```
# Plot comparison of sparse solutions
plt.figure(figsize=(8, 6))
plt.stem(w_star, linefmt="r-", markerfmt="ro", label="Log-barrier solution (Sparse)")
plt.stem(w.value, linefmt="g-", markerfmt="go", label="CVXPY solution")
plt.legend()
plt.title("Comparison of Sparse Solutions")
plt.xlabel("Index")
plt.ylabel("Value")
plt.show()
```



As we see, the sparsity is basically identical in terms of the indices but the values are in general different and even too different. In fact, we can only recover the signs but not the true values that's why we have such a plot.

Double-cliquez (ou appuyez sur Entrée) pour modifier