Convex Optimization - Homework 3

Oliver Jack
MVA

November 12, 2024

1.

Let us reformulate our problem:

$$\min_{v,w} \quad \frac{1}{2}||v||_2^2 + \lambda||w||_1$$

$$\text{subject to} \quad v = Xw - y$$

Then the Lagrangian is given by:

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

where $v, \mu \in \mathbb{R}^n, w \in \mathbb{R}^d$.
Dual function:

$$g(\mu) = \inf_v \left(\frac{1}{2}||v||_2^2 + \mu^T v\right) + \inf_w \left(\lambda||w||_1 - (X^T\mu)^T w\right) + \mu^T y$$

Set

$$f : \mathbb{R}^n \to \mathbb{R}, \quad v \mapsto \frac{1}{2}||v||_2^2 + \mu^T v$$

Then:

$$\nabla_v f(v) = 0 \iff v = -\mu$$

As seen in exercise 2 of sheet 2:

$$\begin{aligned}\inf_w(\lambda||w||_1 - (X^T\mu)w) &= -\lambda \sup_w \left(\left(\frac{X^T\mu}{\lambda}\right) w - ||w||_1\right) \\ &= \begin{cases} 0 & \text{if } ||\frac{X^T\mu}{\lambda}||_\infty \leq 1 \iff ||X^T\mu||_\infty \leq \lambda \\ -\infty & \text{otherwise} \end{cases}\end{aligned}$$

So:

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

Dual problem:

$$\max_{\mu} \quad -\frac{1}{2}||\mu||_2^2 + \mu^T y$$
$$\text{subject to} \quad ||X^T \mu||_\infty \leq \lambda$$

This is equivalent to saying:

$$\min_{v} \quad \frac{1}{2}v^T v - y^T v$$
$$\text{subject to} \quad |X^T v| \preceq \lambda \mathbf{1}_d \iff \begin{pmatrix} X^T \\ -X^T \end{pmatrix} v \preceq \lambda \mathbf{1}_{2d}$$

This can be rewritten as:

$$\min_{v} \quad v^T Q v + p^T v$$
$$\text{subject to} \quad Av \preceq b$$

where

$$Q = \frac{1}{2} I_n, \quad p = -y, \quad A = \begin{pmatrix} X^T \\ -X^T \end{pmatrix}, \quad b = \lambda \mathbf{1}_{2d}$$

2.

The centering problem can be written as:

$$\min_{v} \quad f_t(v) = t(v^T Q v + p^T v) - \sum_{i=1}^{2d} \log(b_i - (Av)_i)$$

To iteratively minimize f_t wrt. t using the Newton method, we first need to compute its gradient and Hessian wrt. v:

$$\nabla_v f_t(v) = t(Q + Q^T)v + tp + \sum_{i=1}^{2d} \frac{(A_{i,.})^T}{b_i - (Av)_i}$$

where $A_{i,.}$ is the i-th row of A. This can be written in vectorized form as:

$$\nabla_v f_t(v) = t(Q + Q^T)v + tp + A^T \frac{1}{b - Av}$$

where

$$\frac{1}{b - Av} = \begin{pmatrix} \frac{1}{b_1 - (Av)_1} \\ \vdots \\ \frac{1}{b_{2d} - (Av)_{2d}} \end{pmatrix}$$

Next:

$$\nabla_v^2 f_t(v) = t(Q + Q^T) + \sum_{i=1}^{2d} \frac{(A_{i,.})^T A_{i,.}}{(b_i - (Av)_i)^2}$$

In matrix form, this can be written as:

$$\nabla_v^2 f_t(v) = t(Q + Q^T) + A^T \text{diag}\left(\left(\frac{1}{(b_i - (Av)_i)^2}\right)_{i=1,\ldots,2d}\right) A$$

To compute the Newton step and decrement, we can then use a linear solver to solve

$$\nabla_v^2 f_t(v) \Delta v_{nt} = -\nabla_v f_t(v)$$

and subsequently

$$\lambda^2(v) = -\nabla_v f_t(v) \Delta v_{nt}$$

3.

Our implemented functions can be tested by randomly generating matrices X and observations y with $\lambda = 10$. In this particular case, I chose $n = 10$ and $d = 10000$, while the coefficients of X and y are sampled random variables following a uniform distribution on the interval $[-1, 1]$. The plot below describes the evolution of the gap $f(v_t) - f^*$ for $\epsilon = 0.01$, $\alpha = 0.1$ and $\beta = 0.5$ (backtracking line search parameters), while considering different values for $\mu \in \{2, 15, 50, 100, 200, 500\}$.

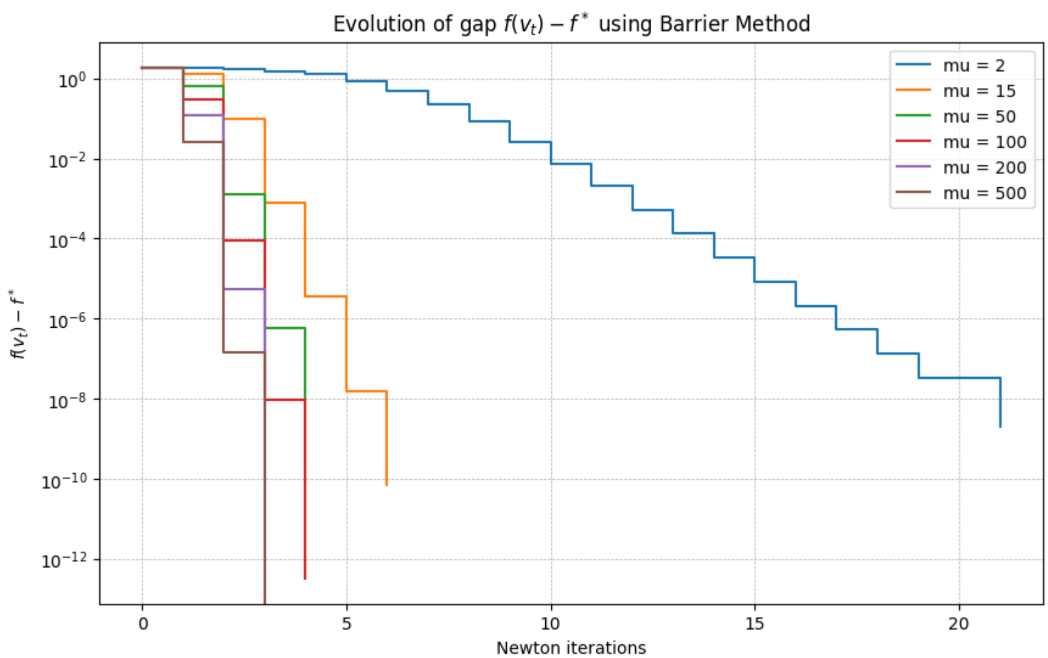

Figure 1: Evolution of the gap $f(v_t) - f^*$ using the Barrier Method

Based on this plot, we can observe that for small values of μ, the algorithm requires fewer Newton steps (inner iterations) for each outer iteration, but this increases the overall number of outer iterations needed. On the other hand, for larger values of μ, the algorithm takes more inner iterations but requires fewer outer iterations to reach the desired gap precision. Since there seems to be a trade-off between inner and outer iterations, the choice of $\mu = 50$ or $\mu = 100$ may offer a good balance between the different types of iterations.

Code

```python
import numpy as np
import matplotlib.pyplot as plt


def original_objective(Q, p, v):
    return v.T @ Q @ v + p.T @ v


def barrier_objective(Q, p, A, b, t, v):

    residual = b - A @ v
    if np.any(residual <= 0):
        return np.inf  # Return infinity if the point is infeasible

    # Barrier term
    barrier_term = -np.sum(np.log(residual)) / t

    # Barrier objective function
    f = t * original_objective(Q, p, v) + barrier_term
    return f


def centering_step(Q, p, A, b, t, v0, eps, alpha=0.1, beta=0.5):
    v = v0
    v_seq = [v0]

    while True:
        # Compute gradient & Hessian
        residual = b - A @ v
        gradient = t * (Q + Q.T) @ v + t * p + A.T @ (1 / residual)
        hessian = t * (Q + Q.T) + A.T @ np.diag(1 / residual**2) @ A

        # Compute Newton step & decrement
        newton_step = np.linalg.solve(hessian, - gradient)
        newton_decrement = - gradient.T @ newton_step

        # Break condition for centering step
        if newton_decrement / 2 <= eps:
            break

        # Backtracking line search
        s = 1
        current_obj = barrier_objective(Q, p, A, b, t, v)
        # Break condition for backtracking line search step
        while barrier_objective(Q, p, A, b, t, v + s * newton_step) > current_obj + alpha * s * gradient.T @ newton_step:
            s *= beta

        # Keep track of updated v values
        v = v + s * newton_step
        v_seq.append(v)
```

```python

    return v_seq


def barr_method(Q, p, A, b, v0, eps, t=1, mu=10):
    v = v0
    v_seq = []
    m = b.size

    while True:
        # Run centering step for current v
        v_center_seq = centering_step(Q, p, A, b, t, v, eps)

        # Keep last value v obtained via centering step
        if len(v_center_seq) > 0:
            v = v_center_seq[-1]
            v_seq.append(v)

        # Break condition for barrier method
        if m / t < eps:
            break

        # Increase t by factor mu
        t *= mu

    return v_seq


# Initialization of parameters
lam = 10
mus = [2, 15, 50, 100, 200, 500]
n = 10
d = 10000

X = np.random.uniform(-1, 1, (n, d))
y = np.random.uniform(-1, 1, n)

# Reformulating problem to general QP
Q = 0.5 * np.identity(n)
p = -y
A = np.vstack([X.T, -X.T])
b = np.full(2*d, lam)
eps = 0.01

# Check if v0 satisfies the feasibility condition A @ v0 <= b
v0 = np.zeros(n)
while not np.all(A @ v0 <= b):
    v0 = np.random.uniform(-1, 1, n)

all_results = {}
f_star = np.inf

# Run barrier method for different values of mu
for mu in mus:
```

```python
    print(f"Running Barrier Method with mu = {mu}")
    v_seq = barr_method(Q, p, A, b, v0, eps, mu=mu)
    f_values = [original_objective(Q, p, v) for v in v_seq]

    all_results[mu] = f_values

    # Keep track of minimal value found for objective function f0 (f^*)
    f_star = min(f_star, min(f_values))

# Plot evolution of gap for different values of mu using matplotlib
plt.figure(figsize=(10, 6))

for mu, f_values in all_results.items():
    gaps = [f - f_star for f in f_values]
    plt.step(range(len(gaps)), gaps, label=f'mu = {mu}', where='post')

plt.yscale('log')
plt.xlabel('Newton iterations')
plt.ylabel(r'$f(v_t) - f^*$')
plt.title('Evolution of gap $f(v_t) - f^*$ using Barrier Method')
plt.legend()
plt.grid(True, which="both", linestyle='--', linewidth=0.5)

plt.show()
```