Convex Homework 3

Julien Delavande

November 2024

Question 1: Deriving the dual

We consider:

$$\min_{w \in \mathbb{R}^d} \frac{1}{2}\|Xw - y\|_2^2 + \lambda \|w\|_1 \quad (\lambda > 0).$$

We rewrite it as:

$$\min_{w} \frac{1}{2}\|z\|_2^2 + \lambda \|w\|_1, \quad \text{where } z = Xw - y.$$

The Lagrangian $\mathcal{L}$ is computed as follows. Let $v \in \mathbb{R}^d$:

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

The function $d(v)$ is derived by:

$$d(v) = \inf_{w,z} \mathcal{L}(w, z, v).$$

Since $\mathcal{L}$ is quadratic in z, we set $\nabla_z \mathcal{L} = 0$ to obtain:

$$z^* = v.$$

Thus, we have:

$$\inf_z \mathcal{L} = \frac{1}{2}\|v\|_2^2 - v^\top y.$$

We then compute:

$$d(v) = \inf_w \left[\frac{\lambda}{2}\|w\|_1 + v^\top Xw \right] - \frac{1}{2}\|v\|_2^2 - v^\top y.$$

Using the dual property of $\|w\|_1$, we obtain:

$$d(v) = \begin{cases} -\frac{1}{2}\|v\|_2^2 - v^\top y & \text{if } \|X^\top v\|_\infty \leq \lambda, \\ -\infty & \text{otherwise.} \end{cases}$$

The dual problem is therefore:

$$\max_v d(v) = \min_v \frac{1}{2}\|v\|_2^2 + v^\top y,$$

subject to $\|X^\top v\|_\infty \leq \lambda$.

Finally, we deduce that the dual problem can be written as:

$$\min_{v \in \mathbb{R}^m} \frac{1}{2} v^\top Q v + p^\top v, \quad \text{with } Av \leq b,$$

where:

$$Q = \frac{1}{2} I_m, \quad p = y, \quad A = \begin{bmatrix} X^\top \\ -X^\top \end{bmatrix}, \quad b = \lambda \begin{bmatrix} \mathbf{1} \\ \mathbf{1} \end{bmatrix}.$$

Question 2

```python
import numpy as np

def centering_step(Q, p, A, b, t, v0, eps):
    """
    Implements Newton's method to solve the centering step in the barrier method for QP.

    Parameters:
    Q   : numpy.ndarray
        Positive definite matrix in the quadratic term of the objective function (n x n).
    p   : numpy.ndarray
        Coefficient vector in the linear term of the objective function (n,).
    A   : numpy.ndarray
        Constraint matrix (m x n).
    b   : numpy.ndarray
        Constraint vector (m,).
    t   : float
        Barrier method parameter.
    v0  : numpy.ndarray
        Initial guess for the variables (n,).
    eps : float
        Target precision.

    Returns:
    v_seq : list of numpy.ndarray
        Sequence of variable iterates.
    """
    v = v0.copy()
    v_seq = [v.copy()]
    m, n = A.shape
    alpha = 0.01
    beta = 0.5

    def compute_phi_and_derivatives(v):
        residual = b - A @ v
        if np.any(residual <= 0):
            # Return infinity if outside the domain
            return np.inf, None, None
        phi = t * (0.5 * v.T @ Q @ v + p.T @ v) - np.sum(np.log(residual))
        grad = t * (Q @ v + p) + A.T @ (1 / residual)
        Hessian = t * Q + A.T @ np.diag(1 / (residual ** 2)) @ A

        return phi, grad, Hessian

    while True:
        phi, grad, Hessian = compute_phi_and_derivatives(v)

        # Compute Newton step and decrement
        try:
            # Solve Hessian * delta_v = -grad
            delta_v = -np.linalg.solve(Hessian, grad)
        except np.linalg.LinAlgError:
            # Hessian is singular
            print("Hessian is singular at iteration.")
            break

        lambda_squared = grad.T @ np.linalg.solve(Hessian, grad)
```

```
        # Check stopping criterion
        if lambda_squared / 2 <= eps:
            break

        # Backtracking line search
        step_size = 1.0
        while True:
            v_new = v + step_size * delta_v
            residual_new = b - A @ v_new
            if np.all(residual_new > 0):
                phi_new, _, _ = compute_phi_and_derivatives(v_new)
                if phi_new <= phi + alpha * step_size * grad.T @ delta_v:
                    break
            step_size *= beta

        # Update v
        v = v_new
        v_seq.append(v.copy())

    return v_seq

def barr_method(Q, p, A, b, v0, eps, mu=10, max_iter=50):
    """
    Implements the barrier method to solve a quadratic program (QP).

    Parameters:
    Q, p, A, b : QP parameters.
    v0         : Initial feasible point.
    eps        : Target precision for the duality gap.
    mu         : Multiplicative factor to increase t.
    max_iter   : Maximum number of outer iterations.

    Returns:
    v_seq      : Sequence of variable iterates.
    gap_seq    : Sequence of duality gaps.
    """
    t = 1.0       # Initial barrier parameter
    m = len(b)    # Number of inequality constraints
    v = v0.copy()
    v_seq = [v.copy()]
    gap_seq = []

    for _ in range(max_iter):
        # Centering step
        v_seq_center = centering_step(Q, p, A, b, t, v, eps)
        v = v_seq_center[-1]
        v_seq.extend(v_seq_center[1:])

        # Duality gap
        duality_gap = m / t
        gap_seq.append(duality_gap)

        if duality_gap <= eps:
            break
        else:
            t *= mu

    return v_seq, gap_seq
```

Question 3: Impact of Barrier Parameter μ

The objective of this experiment is to analyze the effect of the barrier method parameter μ on the optimization process and the sparsity of the solution w. For the given problem, we solve the LASSO dual formulation using the barrier method and vary μ to observe its influence on the weight values and convergence.

Precision Criterion and Gap Analysis

For different values of μ (2, 15, 50, and 100), we plot the precision criterion, represented by the gap $f(v_t) - f^*$, in semilog scale. Here, f^* is approximated as the best objective value found during the experiments. The results are shown in Figure 1.

```
import numpy as np

def find_feasible_point(A, b):
    """
    Finds a feasible point v0 such that A v0 < b.

    Parameters:
    A : Constraint matrix (m x n).
    b : Constraint vector (m,).

    Returns:
    v0 : Feasible point (n,).
    """
    m, n = A.shape
    # Introduce slack variable s
    c = np.zeros(n + 1)
    c[-1] = 1  # Objective is to minimize s

    # Constraints: A v - s <= b
    A_aug = np.hstack([A, -np.ones((m, 1))])
    bounds = [(None, None)] * n + [(0, None)]  # s >= 0

    from scipy.optimize import linprog
    res = linprog(c, A_ub=A_aug, b_ub=b, bounds=bounds, method='highs')
    if res.success and res.x[-1] <= 1e-8:
        v0 = res.x[:-1]
        return v0
    else:
        raise ValueError("No feasible point found.")

import matplotlib.pyplot as plt


# Generate random data
np.random.seed(42)
n = 100
d = 20
X = np.random.randn(n, d)
true_w = np.zeros(d)
sparse_indices = np.random.choice(range(d), size=5, replace=False)
true_w[sparse_indices] = np.random.randn(5)
y = X @ true_w + 0.1 * np.random.randn(n)
```

```
lambda_reg = 10

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

# Find feasible point
v0 = find_feasible_point(A, b)
eps = 1e-6
mu_values = [2, 15, 50, 100]
results = {}

for mu in mu_values:
    print(f"Running barrier method with mu = {mu}")
    v_seq, gap_seq = barr_method(Q, p, A, b, v0, eps, mu=mu)
    results[mu] = {
        'v_seq': v_seq,
        'gap_seq': gap_seq,
        'final_v': v_seq[-1]
    }

def dual_objective(v):
    return 0.5 * np.linalg.norm(v)**2 + y.T @ v

# Find f*
f_values = []
for mu in mu_values:
    final_v = results[mu]['final_v']
    f_val = dual_objective(final_v)
    f_values.append(f_val)
f_star = min(f_values)
print(f"Best objective value found (f*): {f_star}")

# Compute f(v_t) - f*
for mu in mu_values:
    v_seq = results[mu]['v_seq']
    f_gap = []
    for v in v_seq:
        f_val = dual_objective(v)
        f_gap.append(f_val - f_star)
    results[mu]['f_gap'] = f_gap

# Plot convergence
plt.figure(figsize=(12, 6))
for mu in mu_values:
    f_gap = results[mu]['f_gap']
    plt.semilogy(f_gap, label=f' = {mu}')
plt.xlabel('Iteration')
plt.ylabel('Objective Gap f(v_t) - f*')
plt.title('Convergence of Barrier Method for Different  Values')
plt.legend()
plt.grid(True)
plt.show()

# Compute w and plot
plt.figure(figsize=(12, 6))
for mu in mu_values:
```

```
    final_v = results[mu]['final_v']
    w = -X.T @ final_v
    results[mu]['w'] = w
    plt.plot(w, label=f' = {mu}')
plt.xlabel('Feature Index')
plt.ylabel('Weight Value')
plt.title('Weights w Obtained for Different  Values')
plt.legend()
plt.grid(True)
plt.show()
```

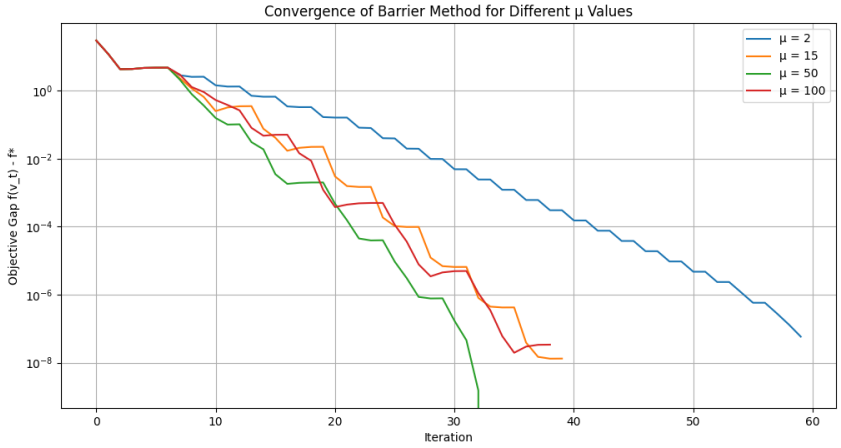

Figure 1: Precision criterion $f(v_t) - f^*$ in semilog scale for different μ values. Smaller μ values converge more gradually but are more stable, while larger μ values converge faster but may introduce instability.

Impact on Weight Values w

We also examine the effect of μ on the weight values w. The weights w are computed for the final iterate of v for each μ. The plot of w values is shown in Figure 2.

Discussion and Appropriate Choice of μ

From the experiments, we observe the following:

- Smaller values of μ (e.g., $\mu = 2$) lead to a more gradual convergence. However, they ensure numerical stability and a smooth optimization trajectory.
- Larger values of μ (e.g., $\mu = 100$) result in faster convergence but may introduce instability and reduced precision in the final solution.
- The weight values w remain consistent across all μ, indicating that the optimization process is robust to μ variations.

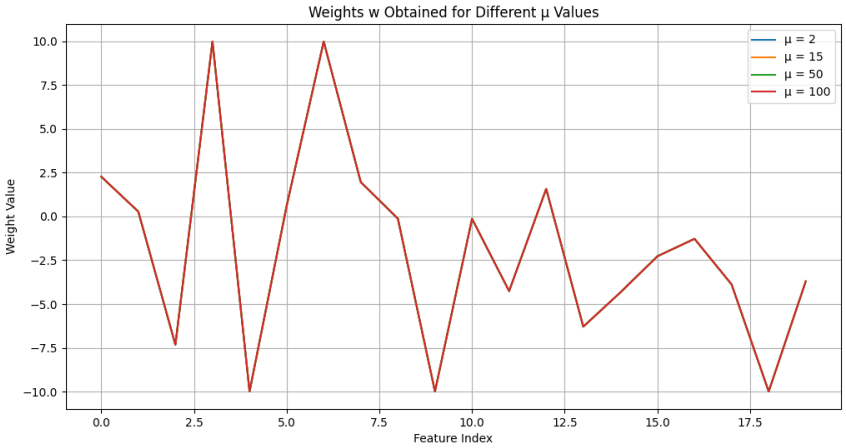

Figure 2: Weight values w obtained for different μ values. The solution converges to the same values across all μ, demonstrating the robustness of the method.