HW3

LEVY
JAROD

1] We want to write the LASSO problem as a general Quadratic Problem:

$$\underset{w}{\text{Minimize}} \ \frac{1}{2}\|Xw - y\|_2^2 + \lambda \|w\|_1$$

Let's introduce $z = Xw$. The problem becomes:

$$\underset{w,z}{\text{Minimize}} \ \frac{1}{2}\|z - y\|_2^2 + \lambda\|w\|_1$$

$$\text{subject to} \quad z = Xw$$

With the Lagrangian, we obtain: μ is the dual variable

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

$$= \frac{1}{2}\|z - y\|_2^2 - \mu^T z + \lambda\|w\|_1 + \mu^T X w$$

Let's consider 2 subproblems.

- $L_1(w,\mu) = \lambda\|w\|_1 + \mu^T X w = \lambda|w| + \mu^T X w$

For each i: $L_{1,i}(w_i, \mu) = (\lambda + \text{sgn}(w_i)\mu^T X)|w_i|$

This problem is minimized if $\|\mu^T X\|_\infty \leq \lambda$ and we obtain:

$$g_1(\mu) = \begin{cases} 0 & \text{if } \|\mu^T X\|_\infty \leq \lambda \\ -\infty & \text{otherwise} \end{cases}$$

- $L_2(z,\mu) = \frac{1}{2}\|z - \cancel{y}\|_2^2 - \mu^T z$

$\nabla_z L_2 = z - y - \mu = 0 \Rightarrow z = y + \mu$. It reaches a minimum at $z = y + \mu$

So $g_2(\mu) = \frac{1}{2}\|\mu\|_2^2 - \|\mu\|_2^2 - \mu^T y$

We reinjected the minimum in L_2 to obtain this form.

Combining the 2 subproblems, we obtain the following form:

$$\underset{\mu}{\text{Maximize}} \ -\frac{1}{2}\|\mu\|_2^2 - \mu^T y$$

$$\text{Subject to} \quad \|\mu^T X\|_\infty \leq \lambda$$

We can rewritten it as:

$$\underset{\mu}{\text{Minimize}} \ \frac{1}{2}\mu^T\mu + y^T\mu$$
$$\text{subject to} \ \|X\mu\|_\infty \leq \lambda$$

By keeping the factor $\frac{1}{2}$, we achieve the form of a general Quadratic problem with: $\begin{cases} Q = I \\ p = y \\ \mu = v \end{cases}$

Moreover, the constraint can be seen as $2N$ inequality constraints:

$\begin{cases} X\mu \leq \lambda 1_N \\ (-X)\mu \leq \lambda 1_N \end{cases}$. By identifying: $\begin{cases} A = [X \, ; \, -X] \\ b = \lambda 1_{2N} \end{cases}$

So we have the form required at the end: $\square$

$$\text{Minimize} \ \frac{1}{2}v^TQv + p^Tv$$
$$\text{subject to} \ Av \leq b$$

$v \in \mathbb{R}^N$
$Q \succeq 0$

```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings('ignore')
```

Utils methods

Here are some methods that are useful to code this solver of the LASSO problem using Barrier method

```
def objective_function(v, Q, p):
  """
  Calculate the value of the quadratic function considered in the problem
  Parameters:
    v (numpy array): The variable vector.
    Q (numpy array): The quadratic coefficient matrix, should be symmetric.
    p (numpy array): The linear coefficient vector.

  Returns:
    float: The value of the quadratic function.
  """

  return np.dot(v.T, np.dot(Q, v)) + np.dot(p.T, v)


def barrier_function(v, Q, p, A, b, t0):
  """
  Calculate the value of the objective function tf(v) + phi(v) for barrier method
  Parameters:
    v (numpy array): The variable vector.
    Q (numpy array): The quadratic coefficient matrix, should be symmetric.
    p (numpy array): The linear coefficient vector.
    A (numpy array): Coefficient matrix for inequality constraints.
    b (numpy array): Right-hand side vector for inequality constraints.
    t0 (float): Scaling factor for the objective function.

  Returns:
    float: The value of the modified objective function.

  """
  quadratic_linear_term = np.dot(v.T, np.dot(Q, v)) + np.dot(p.T, v)
  barrier_term = -sum([np.log(b[i] - np.dot(A[i], v)) for i in range(b.shape[0])])
  return t0 * quadratic_linear_term + barrier_term


def line_search(Q, p, A, b, v, t, df, dx, t0, alpha=0.1, beta=0.7):
  """
  Perform backtracking line search in the context of barrier method
  Parameters:
    Q, p: Parameters of the quadratic function.
    A, b: Parameters for the inequality constraints.
    v (numpy array): Current point.
    t (float): Initial step size.
    df (numpy array): Gradient of the function at v.
    dx (numpy array): Search direction.
    t0 (float): Barrier parameter.
    alpha (float): Parameter controlling the sufficiency of decrease (default: 0.01).
    beta (float): Reduction factor for step size (default: 0.5).

  Returns:
    numpy array: The next point in the iteration.
  """

  new_v = v + t * dx

  # Check if new point is feasible
  if ((b-A.dot(new_v)) > 0).all():
    # Armijo condition for sufficient decrease
    if barrier_function(new_v, Q, p, A, b, t0) <= barrier_function(v, Q, p, A, b, t0) + alpha * t * np.dot(df.T, dx):
      return new_v

  # If the point is not feasible or does not satisfy decrease condition, reduce step size and try again
  return line_search(Q, p, A, b, v, beta*t, df, dx, t0, alpha, beta)
```

Centering step method

We want to implement the Newton method to solve the centering step given the inputs of the quadratic problem and use a backtracking line search with appropriate parameters.

```
def centering_step(Q, p, A, b, t, v0, eps, num_iter=0):
  """
  Perform the centering step in the barrier method for quadratic optimization

  Parameters:
    Q, p: Parameters of the quadratic function.
    A, b: Parameters for the inequality constraints.
    t (float): Barrier parameter.
    v0 (numpy array): Current point.
    eps (float): Tolerance for the stopping criterion.
    numb_iter (int): Current iteration number (default: 0).

  Returns:
    tuple: The next point in the iteration and the number of iterations.
  """

  # Gradient computation
  grad = t * (2 * np.dot(Q, v0) + p)
  for i in range(b.shape[0]): # Log contribution
    grad += A[i, np.newaxis].T / (b[i] - np.dot(A[i], v0))

  # Hessian computation
  hess = 2 * t * Q
  for i in range(b.shape[0]): # Log contribution
    hess += (np.outer(A[i, np.newaxis].T, A[i, np.newaxis].T)) / ((b[i]-np.dot(A[i], v0))**2)

  # Newton step
  dx = -1 * np.dot(np.linalg.inv(hess), grad)

  # Stopping criterion
  l2 = np.dot(grad.T, np.dot(np.linalg.inv(hess), grad))
  if l2 / 2 <= eps:
    return v0, num_iter

  # Line search to find next point
  v1 = line_search(Q, p, A, b, v0, t=1, df=grad, dx=dx, t0=t)

  # Recursive call with updated point and iteration count
  return centering_step(Q, p, A, b, t, v1, eps, num_iter + 1)
```

Barrier method

Write a function that implements the barrier method to solve QP using precedent function. The function should outputs the sequence of variables iterates.

```
def barr_method_util(Q, p, A, b, v0, eps, t, mu, num_iter=0, num_newton=[], v_seq=[], f_seq_true=[]):
  '''
    Recursive barrier method for solving quadratic optimization problems with inequality constraints.

    Parameters:
      Q, p: Parameters of the quadratic function.
      A, b: Parameters for the inequality constraints.
      v0 (numpy array): Starting point for optimization.
      eps (float): Tolerance for the stopping criterion.
      t (float): Initial barrier parameter.
      mu (float): Scaling factor for the barrier parameter.
      num_iter (int): Accumulated number of iterations across all recursive calls.
      num_newton (list): Stores the number of Newton iterations per barrier parameter.
      v_seq (list): Sequence of points obtained during optimization.
      f_seq_true (list): Sequence of true function values at each point in v_seq.

    Returns:
      tuple: Optimized point, list of Newton iterations, sequence of points, sequence of true function values.
  '''

  # Centering step
  v_center, num_iter_inter = centering_step(Q, p, A, b, t, v0, eps)

  # Store results
```

```
  num_newton.append(num_iter_inter)
  v_seq.append(v_center)
  f_seq_true.append(objective_function(v_center, Q, p)[0][0])

  # Stopping criterion (dual gap)
  if b.shape[0] / t < eps:
    return v_center, num_newton, v_seq, f_seq_true
  else: # Increase barrier method and recurse
    t = mu * t
    return barr_method_util(Q, p, A, b, v_center, eps, t, mu, num_iter + num_iter_inter, num_newton, v_seq, f_seq_true)


def barr_method(Q, p, A, b, v0, eps, mu):
    '''
    Entry function for the barrier method in quadratic optimization problems with inequality constraints.

    Parameters:
      Q, p: Parameters of the quadratic function.
      A, b: Parameters for the inequality constraints.
      v0 (numpy array): Initial guess for the optimization.
      eps (float): Tolerance for the stopping criterion.
      mu (float): Scaling factor for the barrier parameter.

    Returns:
      tuple: Final optimized point, list of Newton iterations per step, sequence of points, sequence of function values.
    '''
    # Initialization
    num_newton = [0]
    v_seq = [v0]
    f_seq_true = [objective_function(v0, Q, p)[0][0]]

    # Call the recursive method
    return barr_method_util(Q, p, A, b, v0, eps, 1, mu, 0, num_newton, v_seq, f_seq_true)
```

FUNCTION TEST

Test of the function on randomly generated matrices X and observations y with lambda = 10.

```
lambd= 10
eps = 10e-6
n, d = 100, 20
mu = 5

# Identify the different parameter to the derivation of Q.1
X = np.random.rand(n,d)
y = np.random.rand(n,1)
Q = 0.5*np.eye(n)
p = -y
A = np.vstack((X.T,-X.T))
b = lambd*np.ones((2*d,1))

v0 = np.zeros((n,1))
mu_list = [2, 5, 15, 30, 50, 100, 150, 200, 300]
w_center_list = []
f_true_list = []
```

Plot 1

```
# Plot of the norm of f(mu) - f(mu*) vs Number of Centering Steps
"""
This plot visualizes how the difference in the objective function values (from its final value) changes
with the number of centering steps for different values of mu
"""


plt.figure(figsize=(10, 6))
for mu in mu_list:
    v_center, num_newton, v_seq, f_seq_true = barr_method(Q, p, A, b, v0, eps, mu)
    w_center = np.linalg.lstsq(X,-v_center-y)[0]
    w_center_list.append(w_center)
    f_true_list.append(f_seq_true[-1])

    if mu in mu_list:
        sns.lineplot(x=range(len(num_newton)), y=np.array(f_seq_true) - f_seq_true[-1], label='mu = ' + str(mu))
```

```
plt.yscale('log')
plt.ylabel("Norm of f(mu) - f(mu)*")
plt.xlabel("Number of Centering Steps")
plt.title("Objective Function Convergence for Different Values of mu")
plt.legend()
plt.show()
```

Plot 2

```
# Plot of the nom of w(mu) - w(mu*) vs mu
"""
This plot shows the norm of the difference between the solution obtained for each mu
and the solution for the best mu (the mu that yields the lowest objective function value)
"""
mu_min = np.argmin(f_true_list)
plt.figure(figsize=(10, 6))
w_diff_norm = [np.linalg.norm(w - w_center_list[mu_min]) for w in w_center_list]
sns.lineplot(x=mu_list, y=w_diff_norm)

plt.ylabel("Norm of w(mu) - w(mu*)")
plt.xlabel("mu")
plt.title("Solution Difference Norm for Various mu")
plt.show()
```

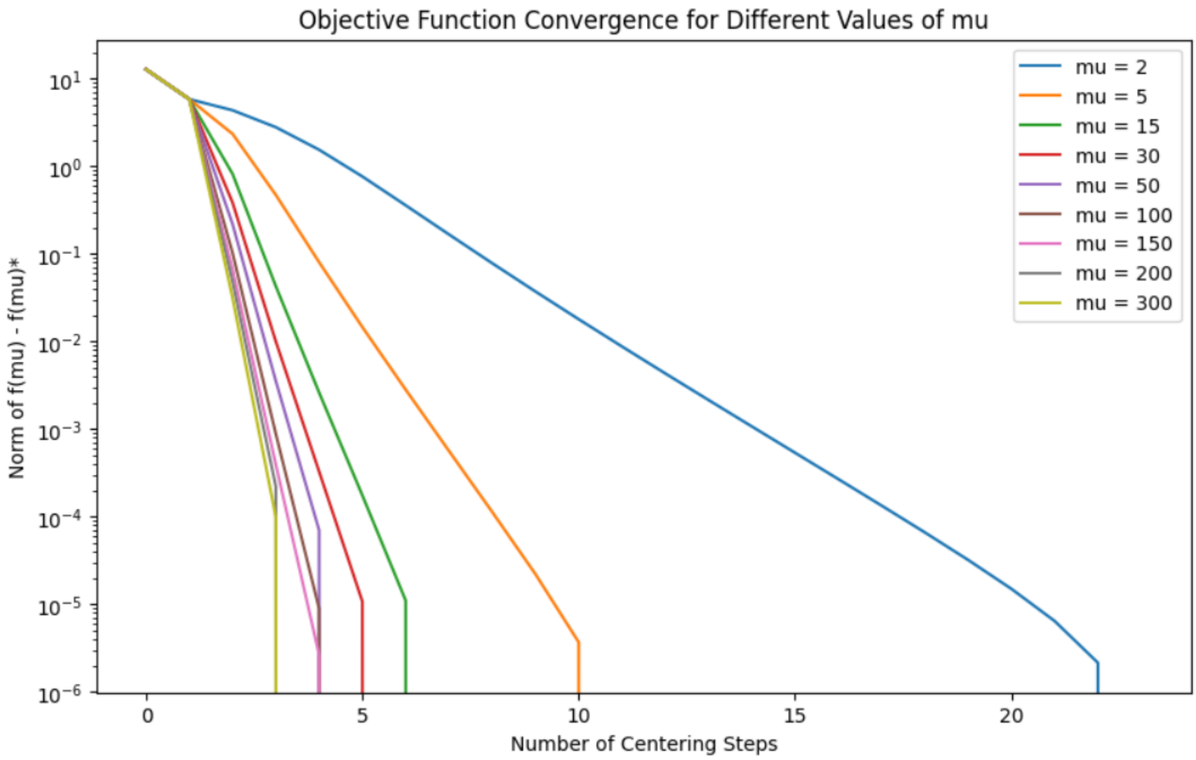
Analysis

- The choice of mu can significantly impact the convergence rate of the barrier method. Small mu leads to slow convergence (more centering steps) but potentially more accurate solutions.
- Plot 1 shows that lower values of mu take more steps to converge to the optimal objective function value. Higher value show a rapid decrease but they also exhibit more steps with no improvement. An intermediate value of mu may provide a balance between fast convergence and solution stability.
- Plot 2 shows significant variation in the solution difference norm as mu changes. Choice of mu should avoid peaks where the norm diverges significantly from zero. Since there is no clear pattern of convergence as mu increases, a heureustic approach may be necessary to choose the best mu.

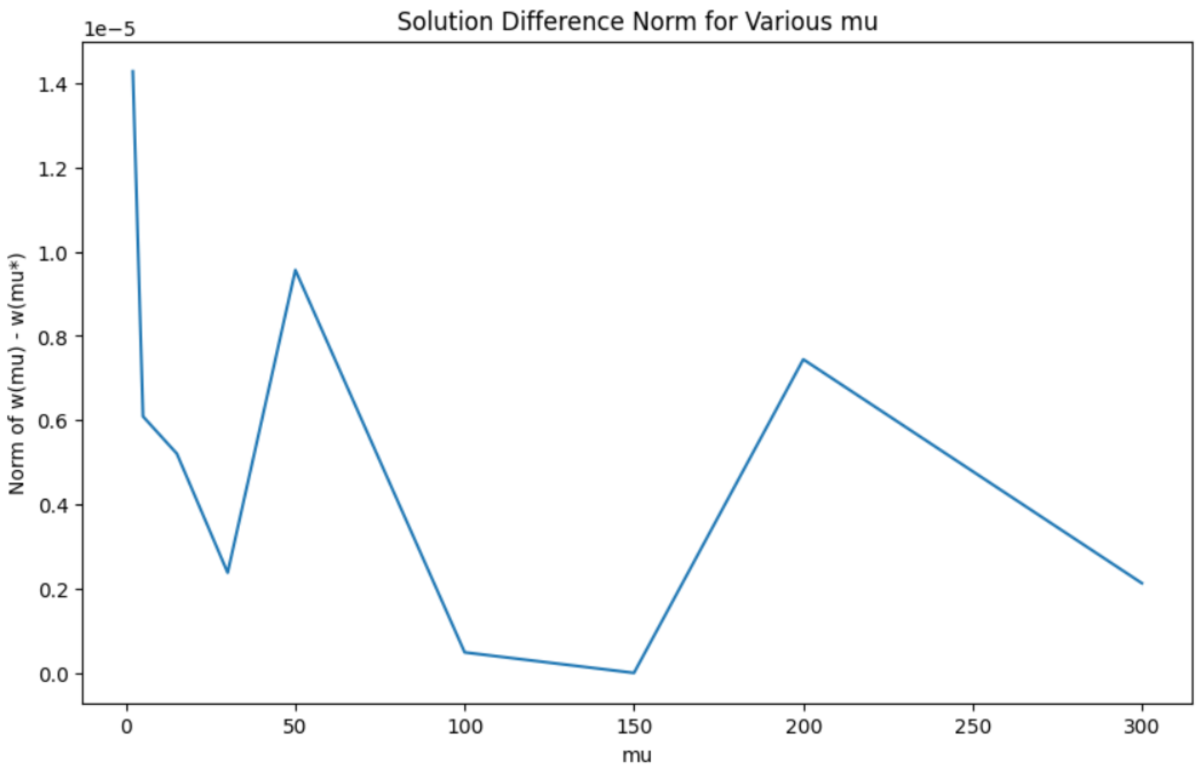
Using the observations of the 2 plots, we advised to take an intermediate value of mu to have a reasonable convergence rate and solution stability: **mu = 30**.

It shows a rapid initial decrease in the objective function value indicating efficient progress and the convergence does not seem to stall as it does with higher values where the curve flattens out.

PLOTS



Objective Function Convergence for Different Values of mu
mu = 2
mu = 5
mu = 15
mu = 30
mu = 50
mu = 100
mu = 150
mu = 200
mu = 300
Norm of f(mu) - f(mu)*
Number of Centering Steps



Solution Difference Norm for Various mu
1e−5
Norm of w(mu) - w(mu*)
mu