Convex Optimization - Homework 3

Samar Rabeh
samar.rabeh@ens-paris-saclay.fr

November , 2024

1 Introduction

The objective of this homework is to solve the LASSO problem. We will approach this by utilizing the QP formulation of the dual LASSO problem. The report is structured as follows: Section 2 outlines the problem setup, while Section 3 presents the theoretical derivation of the QP formulation. In Section 4, we numerically solve the problem using the Barrier method. Lastly, Section 5 discusses and interprets the results obtained.

2 Problem Formulation

We are given $x_1, \ldots, x_n \in \mathbb{R}^d$ as data vectors and $y_1, \ldots, y_n \in \mathbb{R}$ as the corresponding observations. Our goal is to find regression parameters $w \in \mathbb{R}^d$ that best map the data inputs X to the observations y by minimizing the squared differences. In scenarios where the dimensionality is much higher than the number of observations ($n \ll d$), an ℓ_1-norm regularization is typically applied to enforce sparsity in the solution. Formally, the LASSO problem can be expressed as:

$$\min_{w \in \mathbb{R}^d} \frac{1}{2}||Xw - y||_2^2 + \lambda||w||_1$$

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

3 Dual problem of LASSO

3.1 Dual problem

The LASSO problem can be written as:

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

By introducing the auxiliary variable $z = Xw - y$, we can rewrite this as follows:

$$\min_w \frac{1}{2}\|z\|_2^2 + \lambda\|w\|_1$$

The Lagrangian for this formulation is :

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

so:

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

The dual function is derived by minimizing the Lagrangian with respect to z and w:

$$g(v) = \inf_{z,w} L(w, z, v)$$

so we can write:

$$g(v) = \inf_z \left(\frac{1}{2}z^T z - v^T z\right) + \inf_w \left(\lambda\|w\|_1 + v^T Xw\right) - v^T y$$

For the first part, we minimize:

$$\inf_z \left(\frac{1}{2}z^T z - v^T z\right) = -\frac{1}{2}v^T v$$

For the second -part, we apply the conjugate function of $\|\cdot\|_1$:

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

Therefore, the dual function is:

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

The dual problem is written as:

$$\max_v \left(-\frac{1}{2}v^T v - v^T y\right) \quad \text{subject to} \quad \|X^T v\|_\infty \leq \lambda$$

3.2 QP formulation

The dual function can be expressed as:

$$g(v) = \inf_{z,w} L(w, z, v)$$

$$g(v) = -\frac{1}{2}v^T v - v^T y \quad \text{if} \quad \|X^T v\|_\infty \leq \lambda$$

Therefore, the dual optimization problem becomes:

$$\max_v \left(-\frac{1}{2}v^T v - v^T y\right) \quad \text{subject to} \quad \|X^T v\|_\infty \leq \lambda$$

This can be reformulated as a quadratic programming (QP) problem:

$$\min_v \frac{1}{2}v^T Q v + p^T v \quad \text{subject to} \quad Av \preceq b$$

where $Q = \frac{1}{2}I_n$, $A = \begin{pmatrix} X^T \\ -X^T \end{pmatrix} \in \mathbb{R}^{2d \times n}$, $b = \lambda \mathbf{1}_n$, and $p = y \in \mathbb{R}^n$.

4 Resolution with Barrier method

4.1 Barrier method applied to our problem

Given strictly feasible v, $t := t(0) > 0$, $\mu > 1$, tolerance $\epsilon > 0$.

Repeat the following steps:

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

4.2 Centering step - Newton's method

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

Repeat:

- Compute the Newton step and decrement.
- Stopping criterion: Quit if $\|\nabla^2 f(x)\|_2^2 \leq \epsilon$.
- Line search: Choose step size t by backtracking line search.
- Update: $x := x + t\Delta x_{nt}$.

5 Experimental results

In our experiments, we simulate data with dimensions $n = 50$ and $d = 100$, representing a high-dimensional scenario. The regularization parameter is set to $\lambda = 10$ to encourage sparsity in the solutions, and we choose a precision level of $\epsilon = 10^{-6}$. The optimization starts from the trivial feasible point $v_0 = 0$. For solving the problem using Newton's method, we implement backtracking line search with fixed parameters $\alpha = 0.01$ and $\beta = 0.5$.

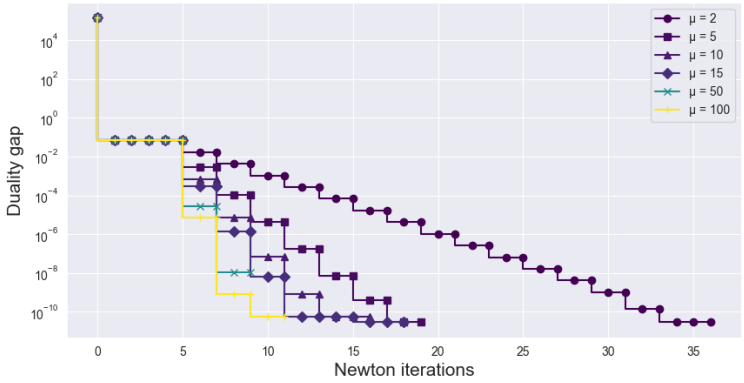


Figure 1: Duality gap over iterations

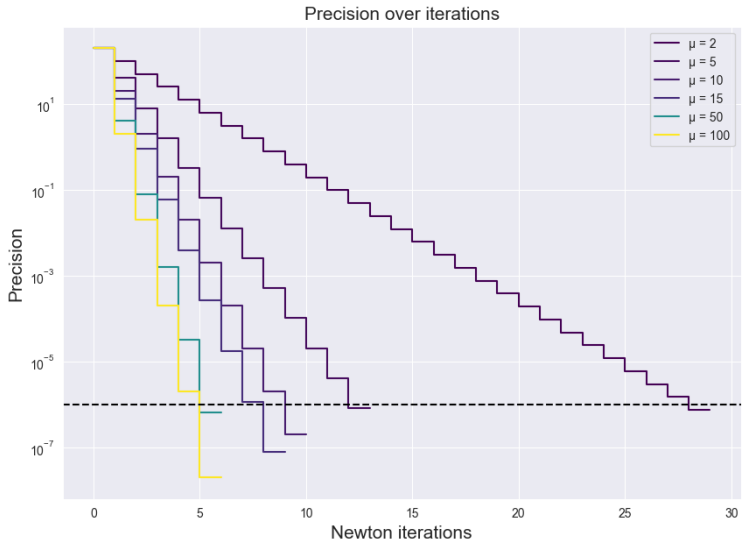


Figure 2: Precision over iterations

5.1 Conclusion

Figure 1 illustrates how the duality gap $f(v_t) - f(v^*)$ evolves as the number of Newton iterations increases, for various values of the barrier method parameter μ in the range of $[2, 5, 10, 15, 30, 50, 100]$.

Figure 2 presents the progression of precision over the cumulative number of Newton steps for the same set of μ values.

From the first plot, it is evident that as μ decreases, the Barrier method converges more slowly, requiring a larger number of iterations to reach convergence (for instance, with $\mu = 2$). This is expected since, with smaller values of μ, each iteration produces smaller updates, leading to a slower reduction in the duality gap. Consequently, more iterations are needed. Therefore, we observe that smaller values of μ correspond to a higher number of outer iterations. On the contrary, for larger values of μ (such as $\mu = 50$ and $\mu = 100$), the method requires fewer outer iterations but longer inner iterations, as the updates are more significant in each step. This results in more iterations for the centering step, which is

carried out using Newton's method.

In conclusion, there is a trade-off between the number of Barrier method iterations and Newton method iterations. Despite different values of μ, we observe that the algorithm consistently reaches nearly the same final solution. Therefore, to minimize computational cost, it would be advantageous to choose a value of μ around $10 - 15$, balancing both outer and inner iterations efficiently.

5.2 Impact of μ on w

Regarding the regression coefficients, we notice that they exhibit significant sparsity, with values falling within the range of $[10^{-13}, 10^{-10}]$ for various choices of μ, as shown in **Figure 3**. These values can be calculated using the formula $w = X^T(v^* + y)$, which follows from the KKT conditions. We do not observe any substantial influence of μ on the coefficients w. Additionally, **Figure 4** illustrates the L_2 norm difference between the computed w and the initial value w_0, confirming that μ has no significant effect on w. The difference remains in the order of 10^{-8} for all tested values of μ.

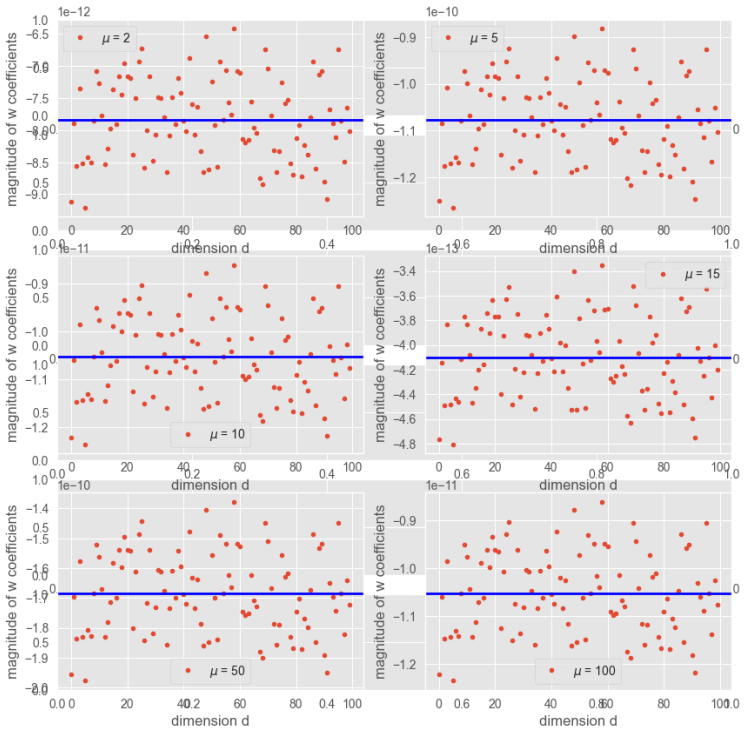


Figure 3: Influence of μ on w magnitude

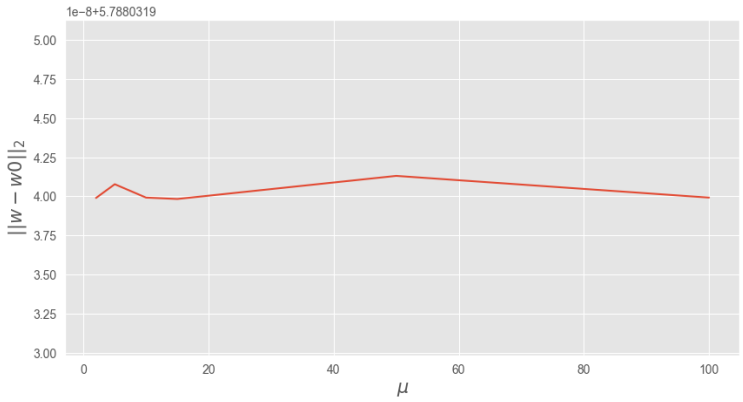


Figure 4: Influence of μ on $\|w - w_0\|_2$

MVA Convex optimization 2024 - Homework 3

RABEH Samar

Utility functions

In [ ]:
```
import numpy as np


def phi(A, b, v):
    return -np.sum(np.log(b-A@v))


def grad_phi(A, b, v):
    return np.sum(A.T / (b-A@v), axis=1)

def check_strict_feasible(A, b, v):
    return np.all(A@v-b < 0)


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


def grad_f(Q, p, v):
    return 2*(Q@v)+p


def hessian_f(Q, p, v):
    return 2*Q

def hessian_phi(A, b, v):
    return A.T@(np.diag(1./(A@v-b)))**2@A


def log_barr_f(Q, p, A, b, t, v):
    if not check_strict_feasible(A, b, v):
        return float("NaN")
    return t*f_(Q, p, v) - phi(A, b, v)


def grad_barr_f(Q, p, A, b, t, v):
    return t*grad_f(Q, p, v) + grad_phi(A, b, v)


def hess_barr_f(Q, p, A, b, t, v):
    return t*hessian_f(Q, p, v) + hessian_phi(A, b, v)
```

Question 2.1.

In [ ]:
```
def centering_step(Q, p, A, b, t, v0, eps, alpha, beta):
    v_seq = [v0]
```

```
    while True:
        grad = grad_barr_f(Q, p, A, b, t, v0)
        hess = hess_barr_f(Q, p, A, b, t, v0)
        step = - np.linalg.inv(hess) @ grad
        decrement_2 = grad.T @ np.linalg.inv(hess) @ grad
        if(decrement_2/2 <=eps):
            break;
        s = backtrack_line_search(v0, step, alpha, beta)
        v0 = v0+s*step
        v_seq.append(v0)
    return np.array(v_seq)


f0 = lambda v: f_(Q, p, v)
f = lambda v: log_barr_f(Q, p, A, b, t, v)
g = lambda v: grad_barr_f(Q, p, A, b, t, v)
def backtrack_line_search(x, step, alpha, beta):
    t=1
    while(f(x+t*step)>=f(x)+alpha*t*g(x).T@step):
        t = beta*t
    return t
```

Question 2.2.

In [ ]:
```
def barr_method(Q, p, A, b, v0, eps, mu, alpha, beta,t):
    m = A.shape[0]
    v_seq = [v0]
    precision = [m/t]
    f_values = []
    Newton_iter = []
    centering_path = [v0]

    while True:
        f_values.append(f0(v0))
        v_seq_newton = centering_step(Q, p, A, b, t, v0, eps, alpha, beta
        v0 = v_seq_newton[-1]
        v_seq.append(v0)
        precision.append(m/t)
        Newton_iter.append(len(v_seq))
        centering_path += [v0]*len(v_seq_newton)
        if (m/t < eps):
            best_f = f0(v0)
            break
        t = mu*t
    return v_seq, precision, f_values, best_f, Newton_iter, centering_pat
```

Question 3.

3.1. Data generation

In [12]:
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style("darkgrid")

# Problem parameters
```

```
n = 50
d = 100
Lambda = 10

X = np.random.random_sample((n, d))
w0 = np.random.random_sample(d)
y = X.dot(w0) + np.random.normal(n)

# Dual parameters
Q = 1/2*np.eye(n)
p = y
A = np.vstack((X.T, -X.T))
b = np.array([Lambda]*2*d)

# Feasible starting point
v = np.zeros(n)
```

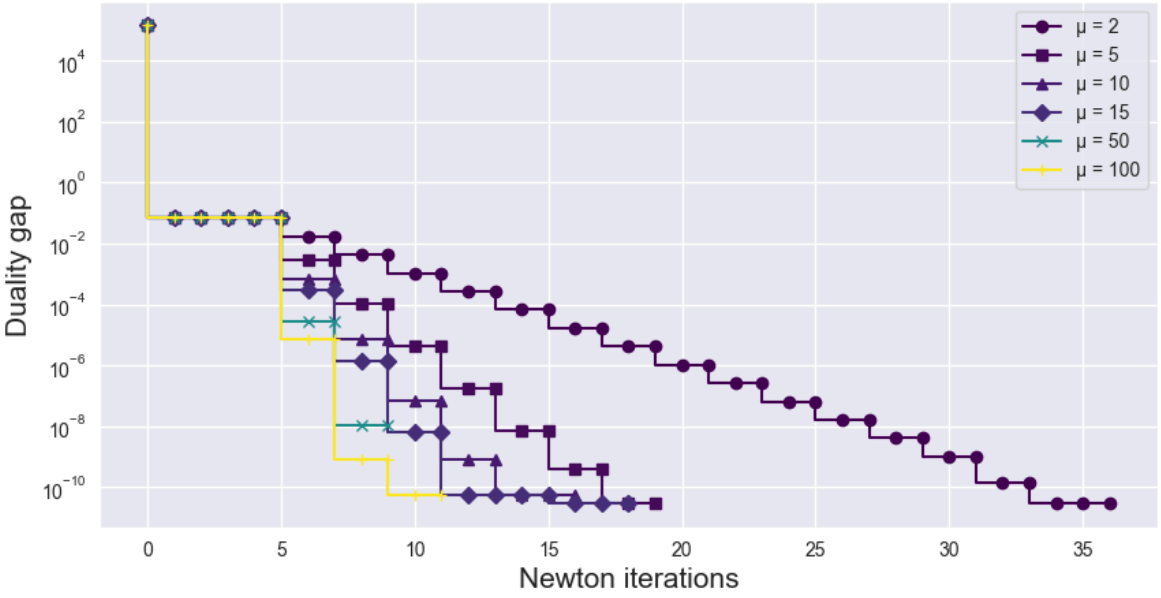
3.2. Parameters

In [ ]:
```
# Parameters of the algorithms
eps = 1e-6
alpha = 0.01
beta = 0.5
t = 1
# Values for test
mus = [2, 5, 10, 15, 50, 100]
```

3.3. Duality gap over iterations

In [21]:
```
import matplotlib.cm as cm


plt.figure(figsize=(10, 5))

labels = []
w_best = []
markers = ['o', 's', '^', 'D', 'x', '+']  # Different markers for each pl
cmap = cm.viridis  # Choose the colormap (viridis)

# Create a normalize object to map mu values to colormap
norm = plt.Normalize(vmin=min(mus), vmax=max(mus))

# Loop over different values of mu
for i, mu in enumerate(mus):
    v_seq, _, f_values, best_f, Newton_iter, centering = barr_method(
        Q, p, A, b, v, eps, mu, alpha, beta, t=1)

    # Compute the duality gap and delta
    duality_gap = np.abs(f_values - np.array(best_f))
    delta = np.array(list(map(f0, centering))) - best_f
    delta = delta[delta > 0]

    # Get a color from the viridis colormap
    color = cmap(norm(mu))

    # Plot the duality gap with a different marker and color for each mu
    plt.step(x=np.arange(0, len(delta)), y=list(delta), label="μ = {}".fo
```

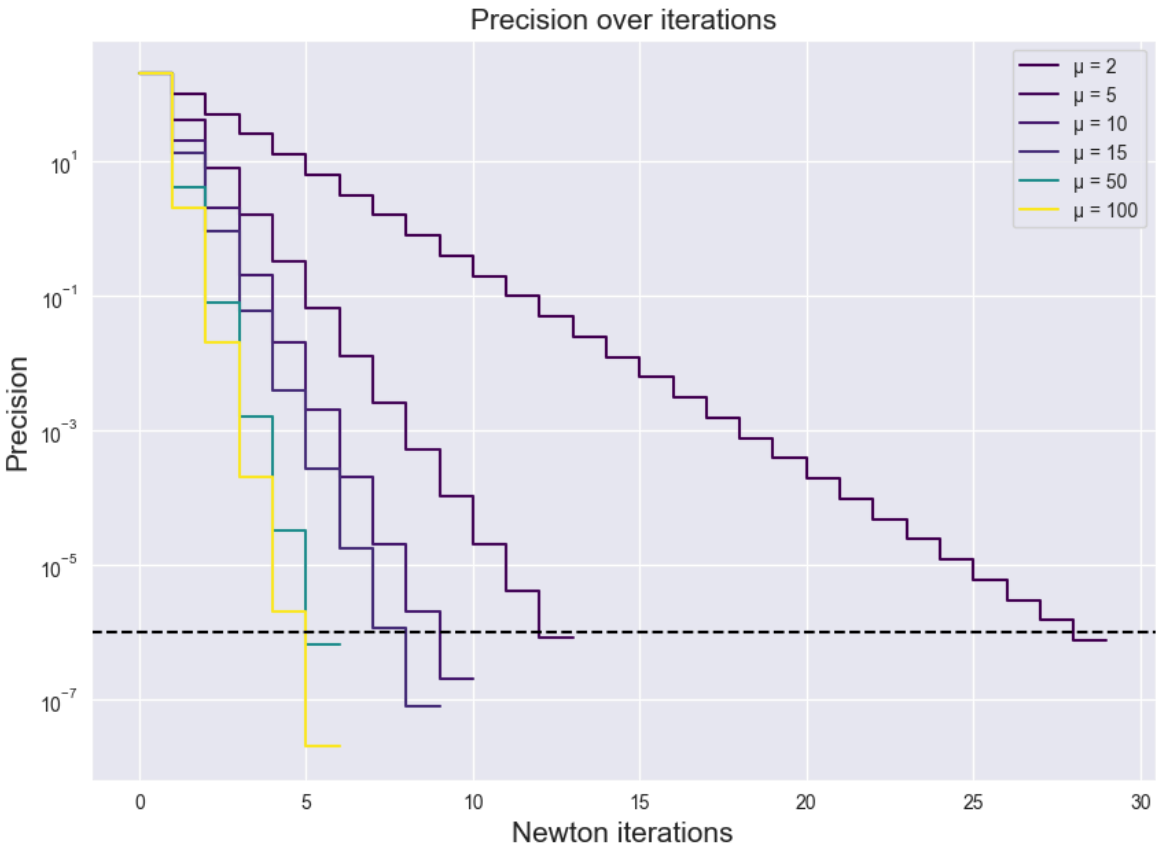
```
            marker=markers[i % len(markers)], linestyle='-', markersize=

    w_best.append(np.dot(np.linalg.pinv(X), v_seq[-1] + y))

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

# Add legend, labels, and title
plt.legend()
plt.xlabel("Newton iterations", fontsize=15)
plt.ylabel("Duality gap", fontsize=15)

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



3.4. Precision over iterations

In [22]:
```
import matplotlib.cm as cm

plt.figure(figsize=(10, 7))

labels = []
cmap = cm.viridis  # Choose the colormap (viridis)
norm = plt.Normalize(vmin=min(mus), vmax=max(mus))  # Normalize for color

# Loop over different values of mu
for i, mu in enumerate(mus):
    precision = barr_method(Q, p, A, b, v, eps, mu, alpha, beta, t=1)[1]

    # Get a color from the viridis colormap
    color = cmap(norm(mu))

    # Plot the precision with a unique color for each mu
    plt.step(range(len(precision)), precision, label="μ = {}".format(mu),

    labels.append(" $\mu$ = {}".format(mu))

# Set the scale for the y-axis to logarithmic
plt.yscale('log')
# Add legend, labels, and title
```

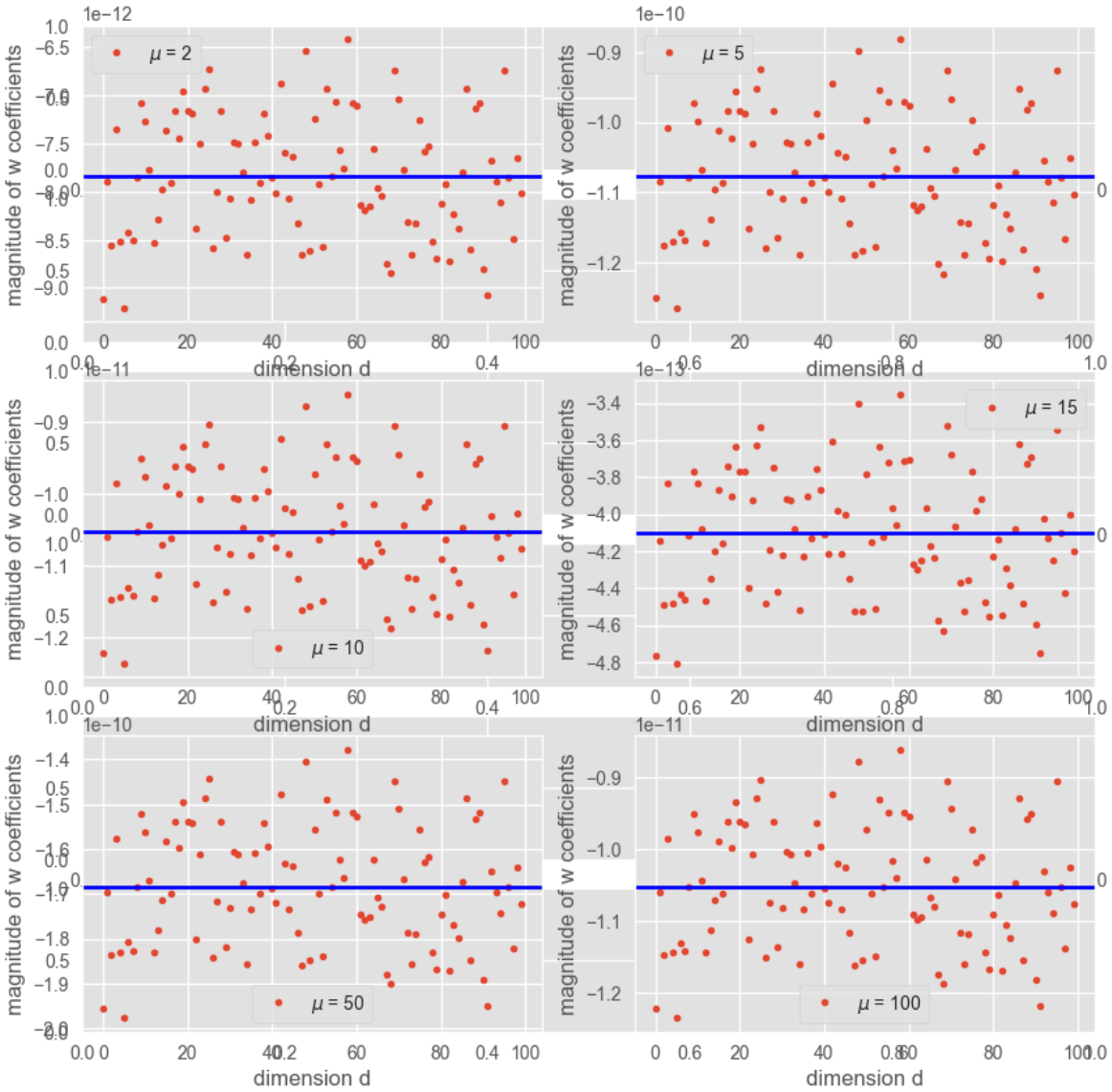
```
plt.legend()
plt.xlabel("Newton iterations", fontsize=15)
plt.ylabel("Precision", fontsize=15)
plt.title('Precision over iterations', fontsize=15)
# Add horizontal line for the epsilon value
plt.axhline(y=eps, c='black', linestyle='--')
# Display the plot
plt.show()
```



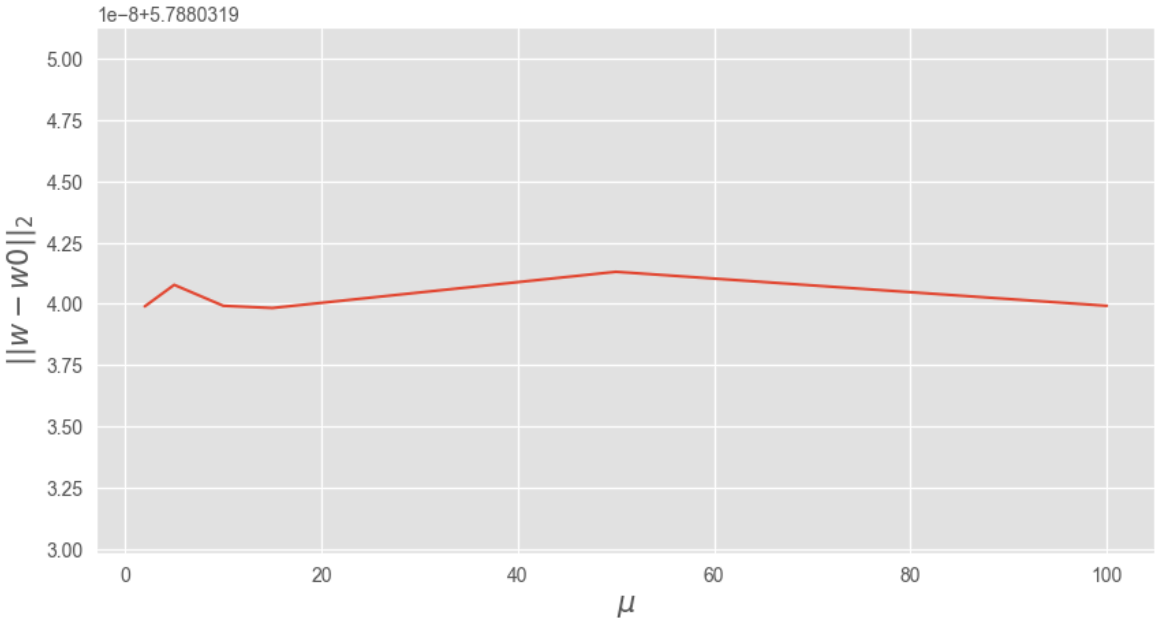
3.5. Impact of μ on w

In [28]:
```
import warnings
import matplotlib.pyplot as plt
import numpy as np
import math

# Suppress warnings
warnings.simplefilter("ignore")

# Create subplots with the correct number of rows and columns
plt.subplots(len(mus), figsize=(10, 10))

# Loop over mus and create subplots for each
for i in range(len(mus)):
    ax = plt.subplot(math.ceil(len(mus) / 2), 2, i + 1)  # Ensure number
    #plt.suptitle('Influence of $\mu$ on magnitude of $w$')
    ax.plot(w_best[i], '.')
    plt.axhline(np.mean(w_best[i]), linestyle='-', color='blue', linewidt
    ax.legend(["$\mu$ = {}".format(mus[i])])
    ax.set_xlabel("dimension d")
    ax.set_ylabel("magnitude of w coefficients")
```

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



In [30]:
```
plt.figure(figsize=(10, 5))

differences = []
for mu in mus:
    v_seq  = barr_method(
        Q, p, A, b, v, eps, mu, alpha, beta,t=1)[0]
    differences.append(np.linalg.norm(np.dot(np.linalg.pinv(X),v_seq[-1]

plt.plot(mus,differences)
plt.ylabel("$||w-w0||_2$",  fontsize=15)
plt.xlabel("$\mu$", fontsize=15)

plt.ylim((min(differences)-1e-8,max(differences)+1e-8));
```



3.6. Number of Newton iterations

In [29]:
```
plt.figure(figsize=(10, 5))
mu_s = [2,5,10,20,30,40,50,65,80,100,120,135,160,175,200]
iter_newton = []
for mu in mu_s:
    Newton_iter  = barr_method(
        Q, p, A, b, v, eps, mu, alpha, beta,t=1)[-1]
    iter_newton.append(sum(Newton_iter))

plt.plot(mu_s,iter_newton,marker='o')
plt.ylabel("Newton iterations",  fontsize=15)
plt.xlabel("$\mu$", fontsize=15)
```

Out[29]:
```
Text(0.5, 0, '$\\mu$')
```