Convex Optimization - Homework 3

Gabriel Watkinson

Report plots, comments and theoretical results in a pdf file. Send your code together with the requested functions and a main script reproducing all your experiments. You can use Matlab, Python or Julia.

Given $x_1, ..., x_n \in \mathbb{R}^d$ data vectors and $y_1, ..., y_n \in \mathbb{R}$ observations, we are searching for regression parameters $w \in \mathbb{R}^d$ which fit data inputs to observations y by minimizing their squared difference. In a high dimensional setting (when $n \ll d$) a ℓ_1 norm penalty is often used on the regression coefficients w in order to enforce sparsity of the solution (so that w will only have a few non-zeros entries). Such penalization has well known statistical properties, and makes the model both more interpretable, and faster at test time.

From an optimization point of view we want to solve the following problem called LASSO (which stands for Least Absolute Shrinkage Operator and Selection Operator)

$$\text{minimize} \quad \frac{1}{2} \|Xw - y\|_2^2 + \lambda \|w\|_1 \tag{LASSO}$$

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

1 Question 1

Derive the dual problem of LASSO and format it as a general Quadratic Problem as follows

$$\begin{array}{ll} \text{minimize} & v^T Q v + p^T v \\ \text{subject to} & Av \preceq b \end{array} \tag{QP}$$

in variable $v \in \mathbb{R}^n$, where $Q \succeq 0$.

We can repose the LASSO problem as the following problem

$$\begin{array}{ll} \min\limits_{w \in \mathbb{R}^d,\, z \in \mathbb{R}^n} & \frac{1}{2} \|z\|_2^2 + \lambda \|w\|_1 \\ \text{subject to} & z = Xw - y \end{array} \tag{LASSO'}$$

The associated Lagragian is

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

We can then calculate the dual function

$$\begin{aligned} g(v) &= \inf_{w,z} \frac{1}{2} z^T z + \lambda \|w\|_1 - v^T (y - Xw + z) \\ &= -v^T y + \inf_z \{\frac{1}{2} z^T z - v^T z\} + \lambda \inf_w \{\|w\|_1 + \frac{1}{\lambda} v^T X w\} \\ &= \begin{cases} -v^T y - \frac{1}{2} v^T v & \text{if } \left\|X^T v\right\|_\infty \leq \lambda \\ -\infty & \text{otherwise} \end{cases} \end{aligned}$$

We can obtain the dual problem

$$\begin{array}{ll} \text{maximize} & -v^T y - \frac{1}{2} v^T v \\ \text{subject to} & \left\|X^T v\right\|_\infty \leq \lambda \end{array} \tag{LASSO*}$$

which can be simplified as follows

$$\begin{array}{ll} \min\limits_{v \in \mathbb{R}^n} & \frac{1}{2} v^T v + y^T v \\ \text{subject to} & -X^T v \leq \lambda \cdot \mathbb{1}_d \\ & X^T v \leq \lambda \cdot \mathbb{1}_d \end{array} \tag{LASSO*}$$

Finally, we can rewrite it as

$$\begin{array}{ll} \text{minimize} & v^T Q v + p^T v \\ \text{subject to} & Av \preceq b \end{array} \tag{QP}$$

with

- $Q = \frac{1}{2} I_n \in \mathbb{R}^{n \times n}$, we have $Q \succeq 0$.
- $p = y \in \mathbb{R}^n$
- $b = \lambda \cdot \mathbb{1}_{2d} \in \mathbb{R}^{2d}$

and

$$A = \begin{pmatrix} X^T \\ -X^T \end{pmatrix} \in \mathbb{R}^{2d \times n}$$

For the next question, we pose $m = 2d$.

2 Question 2 and 3

Implement the barrier method to solve QP.

- Write a function `v_seq = centering_step(Q, p, A, b, t, v0, eps)` which implements the Newton method to solve the centering step given the inputs (Q, p, A, b), the barrier method parameter t (see lectures), initial variable v_0 and a target precision ϵ. The function outputs the sequence of variables iterates $(v_i)_{i=1,\dots,n_\epsilon}$, where n_ϵ is the number of iterations to obtain the ϵ precision. Use a backtracking line search with appropriate parameters.
- Write a function `v_seq = barr_method(Q, p, A, b, v0, eps)` which implements the barrier method to solve QP using precedent function given the data inputs (Q, p, A, b), a feasible point v_0, a precision criterion ϵ. The function outputs the sequence of variables iterates $(v_i)_{i=1,\dots,n_\epsilon}$, where n_ϵ is the number of iterations to obtain the ϵ precision.
- Test your function on randomly generated matrices X and observations y with $\lambda = 10$. Plot precision criterion and gap $f(v_t) - f^*$ in semilog scale (using the best value found for f as a surrogate for f^*). Repeat for different values of the barrier method parameter $\mu = 2, 15, 50, 100, \dots$ and check the impact on w. What would be an appropriate choice for μ ?

2.1 Notations

We can write

$$A = \begin{pmatrix} \hline a_1^T \\ \hline \vdots \\ \hline a_m^T \end{pmatrix}$$

with $a_i \in \mathbb{R}^n$.

The barrier problem can be written as

$$\begin{array}{ll} \min\limits_{v \in \mathbb{R}^n} & t(v^T Q v + p^T v) + \phi(v) \\ \text{subject to} & \forall i \in [1, \cdots, m],\ a_i^T v - b_i \leq 0 \end{array} \tag{Barrier}$$

We pose $\forall i \in [1, \cdots, m],\ f_i(v) = a_i^T v - b_i \in \mathbb{R}$. With this notation, $\phi(v) = -\sum_{i=1}^{m} \log(-f_i(v))$.

From there, we can look at

$$\begin{aligned} \nabla \phi(v) &= \sum_{i=1}^{m} \frac{1}{-f_i(v)} \nabla f_i(v) \\ &= \sum_{i=1}^{m} \frac{1}{b_i - a_i^T v} a_i \end{aligned}$$

and

$$\begin{aligned}\nabla^2\phi(v) &= \sum_{i=1}^{m} \frac{1}{f_i(v)^2} \nabla f_i(v) \nabla f_i(v)^T \\ &= \sum_{i=1}^{m} \frac{1}{(b_i - a_i^T v)^2} a_i a_i^T\end{aligned}$$

since $\nabla^2 f_i(v) = 0$.

Lastly, to get w^* from the dual problem, we can use the fact that the Lagrangian is minimized by the optimal point, the gradient of the Lagrangian with respect to z vanishes. Which gives us:

$$\begin{aligned}\nabla_z \mathcal{L}(w^*, z^*, v^*) &= \nabla_z \left[\frac{1}{2}(z^*)^T z^* + \lambda \left\| w^* \right\|_1 - (v^*)^T (y - X w^* + z^*) \right] \\ &= 0 \\ &= z^* - v^* \\ &= X w^* - y - v^*\end{aligned}$$

Given v^*, X and y, we can get w^* by resolving the least squares :

$$w^* = X^\dagger (y + v^*)$$

2.2 Summary

In the first section, we will display plots resuming the optimization process, to have a quick look at the results.

In a second section, we will look at some details of the implementation on generated data, and compare to results from CVXPY.

Lastly, in a third section, we will explain the implementation details.

2.3 Code for the testing

I decided to use Python to implement the testing on the data.

The following code (in the third section), is only a part of the total code I wrote to during this homework. I only kept the essential for readability reasons.

You can find the complete code on this GitHub repository: https://github.com/gwatkinson/HW3.

2.4 Results

In this section, we will display plots resuming the optimization process.

The code needed to generate the plot is not in this notebook. I wanted to get more familiar with Object-Oriented Programming with Python, so I made a package that implements convex optimization for this problem with classes. The plots come from there.

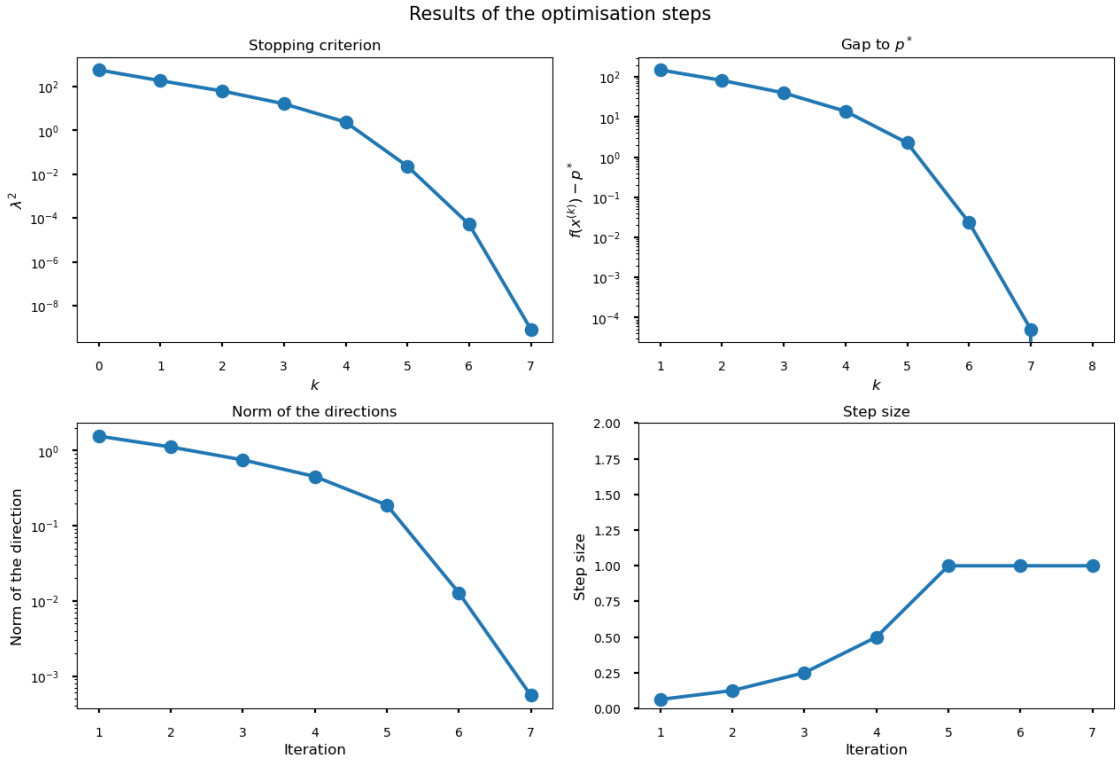
I checked that the outputs were the same for a given dataset.

You can view the entire code of the package on this GitHub repository: https://github.com/gwatkinson/HW3.

And how the plots are generated in the notebook : https://github.com/gwatkinson/HW3/blob/main/test.ipynb, in the last section.

2.4.1 Centering step

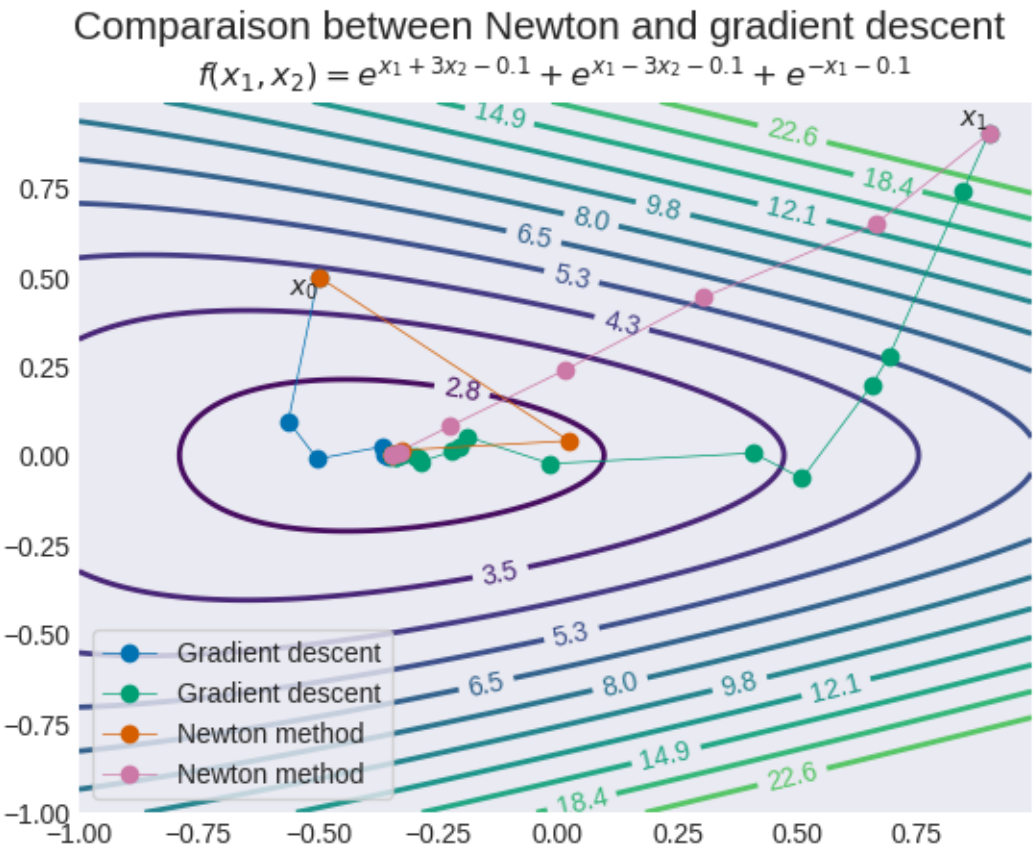
These are the results obtained by a Newton method on the objective function with $\lambda = 10$, $\mu = 10$, $t = 100$, $\alpha = 0.1$ and $\beta = 0.5$



We clearly see the two parts of the method, as described in the course.

Other example On another note, I also used this method and a basic gradient descent on a 2D function, given as an example in the course, to plot the descent.

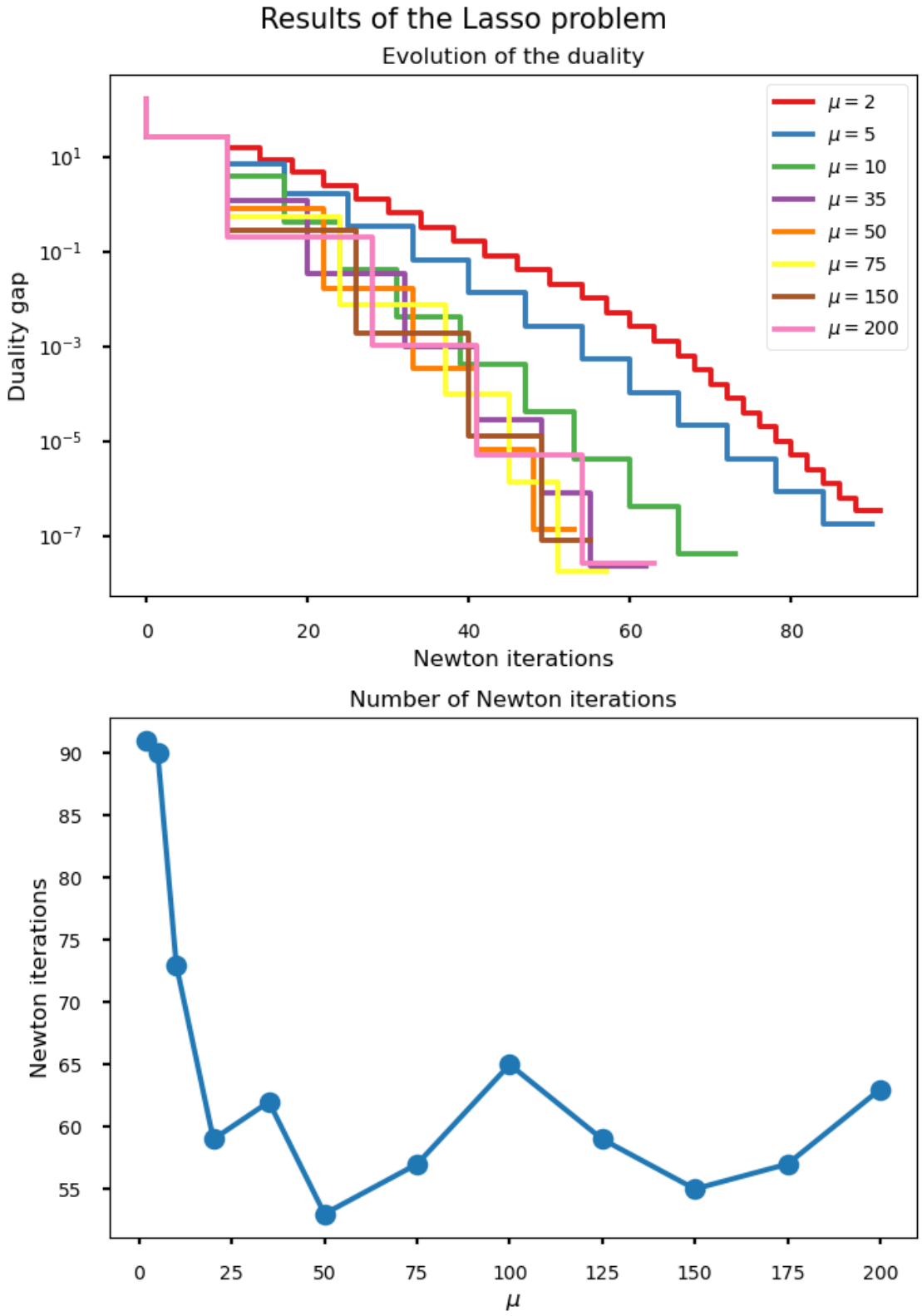
This is just a visual way to confirm that the method is working as intended.



2.4.2 Barrier method

This section presents results of the barrier method on the QP problem.

We plot the dual gap between the optimal value of the dual problem and the value of the LASSO function.



We can see that the barrier method converged successfully. Furthermore, we observe that for small values of $\mu = 2, 5$, many centering steps are needed, which also increases the number of Newton

iterations.

From this graph, we can see that the choice of μ doesn't change significatively the number of iterations when $\mu \geq 20$, as the number of newton steps is around 60.

However, we notice a minimum around 50, so $\mu = 50$ looks like an appropriate choice.

2.5 Test if the algorithms on the data

This section tests the preceding function on generated data.

2.5.1 Generating the data

This cell generates random data for a regression model. It was taken from the CVXPY documentation, and adapted for our needs.

It defines β coefficients, then remove a portion of them to create a sparse problem. Then, it generates a random X dataset. Finally, we have

$$y = X\beta + \mathcal{N}(0, \sigma^2)$$

We can then define the other matrices of the problem Q, p, A and b.

[ ]:
```
def generate_data(N=100, D=50, ld=10, sigma=1, density=0.2):
    "Generates data matrix X and observations Y."
    np.random.seed(1)
    beta_star = np.random.randn(D)
    idxs = np.random.choice(range(D), int((1-density)*D), replace=False)
    for idx in idxs:
        beta_star[idx] = 0
    X = np.random.randn(N,D)
    y = X.dot(beta_star) + np.random.normal(0, sigma, size=N)
    Q = 0.5*np.identity(N)
    p = y
    A = np.vstack((X.T, -X.T))
    b = ld * np.ones(2*D)
    return X, y, beta_star, Q, p, A, b
```

[63]:
```
# Generate the data
N = 100
D = 50
M = 2*D
ld = 10

X, y, beta_star, Q, p, A, b = generate_data(
    N=N,
    D=D,
    ld=ld,
    sigma=1,
    density=0.2
```

```
)

v0 = np.zeros(N)
t0 = 1
```

2.6 Running the optimization on this dataset

[64]:
```
# Optimise with the barrier method
"""
`vs` contains the variables iterates.
`l_iterates` contains the other iterates generated during the centering steps.
"""
vs, ts, l_iterates = barr_method(
    Q, p, A, b, v0, eps=1e-10, t0=t0, mu=20, alpha=0.4, beta=0.5, verbose=False
)
```

[65]:
```
# The final optimal point
v_star = vs[-1]
v_star
```

[65]:
```
array([ 0.31552562, -0.11703727,  0.01728237, -0.61737851, -0.99928907,
        1.46258562, -0.42576194,  0.8568421 , -0.28038062, -0.56092876,
       -0.65274369,  0.29339506,  0.31064405, -0.89736969, -1.7106127 ,
       -1.21882627,  0.7451865 ,  1.78745409, -0.35292253, -1.38315793,
       -0.26489221,  0.00563829,  0.3942802 , -0.00680659, -0.92309693,
        0.06081644,  0.10674194,  1.11850177,  1.64766325,  0.70049316,
        0.98543705, -0.52862293, -0.26416066, -1.50249411,  1.8282942 ,
        0.3445711 ,  0.21621665, -1.34963551,  0.18589437,  1.17286416,
       -0.1218349 ,  1.1707134 ,  0.61834599, -0.16522791,  0.31948091,
        1.71364731,  0.81019719,  0.26572859,  0.38634092,  0.30676617,
        0.10183189, -0.35887503, -1.8900325 ,  0.95092626,  0.13033911,
       -2.07744882, -1.19539063, -2.35701503,  1.03831051, -0.12284202,
       -0.11956244,  0.87448636, -0.02591839, -1.93634463, -1.33856515,
       -1.25210765,  0.31492665, -1.22922936,  0.95031641, -0.89167381,
       -0.64247143, -1.29265585, -0.16343546,  0.56648747,  1.06346221,
       -1.40619349, -1.49697813,  0.13200218, -0.88739194, -0.80867504,
        0.6736527 , -0.40367335,  0.48625451, -0.05944894,  0.1951453 ,
        1.70126284,  1.40729699, -1.19444094,  0.06878788, -0.69743594,
       -0.32995683, -0.12279312, -0.31907041, -0.72057493,  0.23273406,
       -2.01483994,  1.09135349, -0.32760838,  1.08556586, -0.71555631])
```

We see that the found point is inside the constraints and that $\|Xv\|_\infty \leq \lambda$.

[66]:
```
(v_star @ A.T - b).max()
```

[66]:
```
-1.0622613899613498e-12
```

[67]:
```
# Value of the quadratic function evaluated at the optimal point.
f(v_star, Q, p)
```

[67]:
```
-129.945114757688
```

We can then look at the optimal w^* obtained by

$$w^* = X^\dagger(y + v^*)$$

[68]:
```
w_star = np.linalg.lstsq(X, y+v_star, rcond=None)[0]

# Print the coefs of w_star >= 1e-5
np.where(w_star >= 1e-5, w_star, 0)
```

[68]:
```
array([0.        , 0.01118423, 0.06979384, 0.        , 0.        ,
       0.        , 0.13301761, 0.        , 0.20999201, 0.0149517 ,
       0.        , 0.        , 0.        , 0.        , 0.09410407,
       0.        , 0.        , 0.        , 0.        , 0.10858152,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.05109365, 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.45558223,
       0.        , 0.07117971, 0.        , 1.44966444, 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ])
```

We obtain a sparse vector, which is a good sign as it is the objective of a LASSO regression.

2.6.1 Confirmation with CVXPY

In this section, we confirm the results by using a reliable Python library `CVXPY` that implements convex optimization.

[69]:
```
import cvxpy as cp

# Posing the dual problem
x = cp.Variable(N)
dual_prob = cp.Problem(
    cp.Minimize(
        cp.quad_form(x, Q) + p.T @ x
        ),
    [A @ x <= b]
)

# Solving the problem
dual_prob.solve()

# Printing the optimal solution
dual_prob.value, x.value
```

[69]:
```
(-129.94511475773066,
 array([ 0.31552562, -0.11703727,  0.01728237, -0.61737851, -0.99928907,
         1.46258562, -0.42576194,  0.8568421 , -0.28038062, -0.56092876,
        -0.65274369,  0.29339506,  0.31064405, -0.89736969, -1.7106127 ,
        -1.21882627,  0.7451865 ,  1.78745409, -0.35292253, -1.38315793,
        -0.26489221,  0.00563829,  0.3942802 , -0.00680659, -0.92309693,
         0.06081644,  0.10674194,  1.11850177,  1.64766325,  0.70049316,
         0.98543705, -0.52862293, -0.26416066, -1.50249411,  1.8282942 ,
         0.3445711 ,  0.21621665, -1.34963551,  0.18589437,  1.17286416,
        -0.1218349 ,  1.1707134 ,  0.61834599, -0.16522791,  0.31948091,
         1.71364731,  0.81019719,  0.26572859,  0.38634092,  0.30676617,
         0.10183189, -0.35887503, -1.8900325 ,  0.95092626,  0.13033911,
        -2.07744882, -1.19539063, -2.35701503,  1.03831051, -0.12284202,
        -0.11956244,  0.87448636, -0.02591839, -1.93634463, -1.33856515,
        -1.25210765,  0.31492665, -1.22922936,  0.95031641, -0.89167381,
        -0.64247143, -1.29265585, -0.16343546,  0.56648747,  1.06346221,
        -1.40619349, -1.49697813,  0.13200218, -0.88739194, -0.80867504,
         0.6736527 , -0.40367335,  0.48625451, -0.05944894,  0.1951453 ,
         1.70126284,  1.40729699, -1.19444094,  0.06878788, -0.69743594,
        -0.32995683, -0.12279312, -0.31907041, -0.72057493,  0.23273406,
        -2.01483994,  1.09135349, -0.32760838,  1.08556586, -0.71555631]))
```

We can see that the found optimal values are really close, less than 5×10^{-11}.

[70]:
```
# Difference between custom implementation and CVXPy
f(v_star, Q, p) - dual_prob.value
```

[70]:
```
4.2660985855036415e-11
```

And the norm of the difference between the points is less than 2×10^{-10}.

[71]:
```
# Difference between custom implementation and CVXPy
np.linalg.norm(v_star - x.value)
```

[71]:
```
1.4601715730472986e-10
```

CVXPY problem on the LASSO problem.

[72]:
```
# Pose the LASSO problem
w = cp.Variable(D)
prob = cp.Problem(
    cp.Minimize(
        cp.norm2(X @ w - y)**2 + ld * cp.norm1(w)
    )
)

# Solve the problem
prob.solve()
```

```
# Optimal solution
prob.value, np.where(w.value>=1e-5, w.value, 0)
```

[72]:
```
(167.183396521373,
 array([0.        , 0.07926927, 0.13443537, 0.        , 0.00273   ,
        0.        , 0.16170766, 0.        , 0.24259922, 0.08228944,
        0.        , 0.        , 0.        , 0.        , 0.18405218,
        0.        , 0.        , 0.        , 0.        , 0.18060557,
        0.        , 0.        , 0.06113215, 0.        , 0.        ,
        0.01934214, 0.03138581, 0.        , 0.09522368, 0.01020272,
        0.        , 0.        , 0.        , 0.10223705, 0.03396465,
        0.        , 0.        , 0.        , 0.        , 0.53192205,
        0.01789768, 0.15245121, 0.        , 1.48752488, 0.        ,
        0.        , 0.00382742, 0.11365801, 0.        , 0.        ]))
```

[73]:
```
np.where(w_star>=1e-5, w_star, 0)
```

[73]:
```
array([0.        , 0.01118423, 0.06979384, 0.        , 0.        ,
       0.        , 0.13301761, 0.        , 0.20999201, 0.0149517 ,
       0.        , 0.        , 0.        , 0.        , 0.09410407,
       0.        , 0.        , 0.        , 0.        , 0.10858152,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.05109365, 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.45558223,
       0.        , 0.07117971, 0.        , 1.44966444, 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ])
```

We notice some differences between the solutions of the dual problem and the solution of the LASSO implemented by CVXPY. But, we still have the larger coefficients that are similar.

2.7 Rapid implementation

In this section, I implemented the mentioned function quite fast, and tested it on generated data.

[1]:
```
import numpy as np

from typing import Callable

f_type = Callable[[np.array], float]
```

2.7.1 Defining the functions

This cell defines the function of the problem, given the necessary matrices.

[2]:
```
def f(v, Q, p):
    """The original objective function f_0."""
    return v.T @ Q @ v + p.T @ v
```

```
def grad_f(v, Q, p):
    """The gradient of f."""
    return 2 * Q @ v + p


def hess_f(v, Q, p):
    """The hessian of f."""
    return 2 * Q


def g(v, A, b, i):
    """The log affine constraints f_i."""
    ai = A[i, :]
    bi = b[i]
    return float(np.nan_to_num(-np.log(bi - ai.T @ v), nan=np.inf))

def grad_g(v, A, b, i):
    """The gradient of the log affine constraints f_i."""
    ai = A[i, :]
    bi = b[i]
    return ai / (bi - ai.T @ v)

def hess_g(v, A, b, i):
    """The hessian of the log affine constraints f_i."""
    ai = A[i, :]
    bi = b[i]
    return ai.reshape(-1, 1) @ ai.reshape(-1, 1).T / (bi - ai.T @ v) ** 2


def phi(v, A, b):
    """The log barrier function phi."""
    m = len(b)
    return np.sum([g(v, A, b, i) for i in range(m)])


def grad_phi(v, A, b):
    """The gradient of the log barrier function."""
    m = len(b)
    return np.sum(np.array([grad_g(v, A, b, i) for i in range(m)]), axis=0)


def hess_phi(v, A, b):
    """The hessian of the log barrier function."""
    m = len(b)
    return np.sum(np.array([hess_g(v, A, b, i) for i in range(m)]), axis=0)

def dual_func(v, y):
```

```
    """The dual function (the inf of the Lagragian)."""
    return - 0.5 * v.T @ v - v.T @ y

def lasso_func(w, X, y, ld):
    """The lasso function."""
    return 0.5 * np.linalg.norm(X @ w - y)**2 + ld * np.linalg.norm(w, 1)

def get_w_star(v_star, X, y):
    return np.linalg.lstsq(X, y + v_star, rcond=None)[0]
```

2.7.2 Algorithms

This cell implements the algorithms used for resolving the convex problem.

We first define the `backtracking_line_search` abd the `newton_method` that works on any function, given the gradient and hessian.

Then, the `centering_step`, just applies the newton method to the constrained quadratic problem.

Lastly, the `barr_method` implements the barrier method for a given μ.

[3]:
```
def newton_step(x: np.array, grad_f: f_type, hess_f: f_type) -> np.array:
    step = -np.linalg.inv(hess_f(x)) @ grad_f(x)
    return step

def newton_decrement(x: np.array, grad_f: f_type, hess_f: f_type) -> float:
    decrement = np.sqrt(grad_f(x).transpose() @ np.linalg.inv(hess_f(x)) @␣
 ↪grad_f(x))
    return decrement


def backtracking_line_search(
    x: np.array, f: f_type, grad_f: f_type, delta_x: np.array,
    alpha: float, beta: float, verbose: bool = False
) -> float:
    assert alpha > 0 and alpha < 1 / 2, "Alpha must be between 0 and 0.5"
    assert beta > 0 and beta < 1, "Beta must be between 0 and 1"

    t = 1
    _ = 0
    while f(x + t * delta_x) >= f(x) + alpha * t * grad_f(x) @ delta_x:
        t *= beta
        _ += 1
        if _ >= 100:
            raise ValueError("Backtracking not converging")

    return t

def newton_method(
```

```
    x0: np.array,
    f: f_type,
    grad_f: f_type,
    hess_f: f_type,
    epsilon: float,
    alpha: float = 0.1,
    beta: float = 0.5,
    verbose=True
):

    x = x0
    decrement = newton_decrement(x, grad_f, hess_f)

    xs = [x0]
    decrements = [decrement]
    steps = []
    ts = []

    _ = 0
    while decrement**2 / 2 > epsilon:
        try:
            if verbose:
                print(f"\tCriterion: {decrement**2:.4e}")
            decrement = newton_decrement(x, grad_f, hess_f)
            step = newton_step(x, grad_f, hess_f)
            t = backtracking_line_search(
                x, f, grad_f, step, alpha, beta, verbose=verbose
            )
            x += t * step

            decrements.append(decrement)
            steps.append(step)
            ts.append(t)
            xs.append(x)

            _ += 1
            if _ >= 20:
                raise ValueError("Newton method not converging")

        except ValueError as e:
            if verbose:
                print(e)
            break

    return x, f(x), {"xs": xs, "decrements": decrements, "steps": steps, "ts":␣
↪ts}
```

```python
def centering_step(Q, p, A, b, t, v0, eps, alpha=0.1, beta=0.5, verbose=True):
    x_opt, f_opt, iterates = newton_method(
        x0=v0,
        f=lambda v: t * f(v, Q, p) + phi(v, A, b),
        grad_f=lambda v: t * grad_f(v, Q, p) + grad_phi(v, A, b),
        hess_f=lambda v: t * hess_f(v, Q, p) + hess_phi(v, A, b),
        epsilon=eps,
        alpha=alpha,
        beta=beta,
        verbose=verbose
    )

    return x_opt, f_opt, iterates

def barr_method(Q, p, A, b, v0, eps, t0=1, mu=15, alpha=0.1, beta=0.5,␣
 ↪verbose=True):
    t = t0
    m = len(b)
    v = v0
    vs = [v]
    ts = [t]
    l_x_opt = []
    l_f_opt = []
    l_iterates = []

    while m / t >= eps:
        if verbose:
            print(f"t = {t} and m/t = {m/t: .2e}")
        x_opt, f_opt, iterates = centering_step(
            Q, p, A, b, t, v, eps, alpha, beta, verbose=verbose
        )
        v = iterates["xs"][-1]
        t *= mu

        vs.append(v)
        ts.append(t)
        l_iterates.append(iterates)

    return vs, ts, l_iterates
```