MATHÉMATIQUES
VISION
APPRENTISSAGE

Computational Statistics - TD1

Louis Martinez
louis.martinez@telecom-paris.fr

(*The notebook of all implementations can be found here*)

Exercise 1: Lasso

1)

Let $z = Xw - y$. (LASSO) can be reformulated as:

$$\text{minimize } \frac{1}{2}z^T z + \lambda \|w\|_1$$
$$\text{s.t. } Xw - y = z$$

The Lagrangian is:

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

$\mathcal{L}$ is a quadratic form with respect to z, minimized for in v. To minimize with respect to w we use what was done for the previous homework:

$$\begin{aligned}\inf_w\left(\lambda\|w\|_1 - \left(-\left(X^T v\right)^T w\right)\right) &= -\lambda \sup_w\left(-\frac{1}{\lambda}\left(X^T v\right)^T w - \|w\|_1\right) \\ &= -\lambda f_*\left(-\frac{1}{\lambda}X^T v\right) \qquad \text{With } \ f_* \ \text{ the dual of } \ \|.\|_1\end{aligned}$$

This add the constraint $\left\|X^T v\right\|_\infty \leq \lambda$

Thus the dual problem is

$$\text{maximize } -\frac{1}{2}v^T I v - v^T y$$
$$\text{s.t. } \ \left\|X^T v\right\|_\infty \leq \lambda$$

The inequality constraint can be rewritten as a linear system by writing that

$$\forall i \in [\![1, d]\!], -\lambda \leq \left(X^T v\right)_i \leq \lambda$$

with $\left(X^T v\right)_i$ the i-th row of $X^T v$, leading to

$$\forall i \in [\![1, d]\!], -X_i^T v \leq \lambda \quad \text{and} \quad X_i^T v \leq \lambda$$

We thus have $2d$ inequalities, boiling down to

$$\begin{pmatrix} X^T \\ -X^T \end{pmatrix} v = \begin{pmatrix} \lambda I_d \\ \lambda I_d \end{pmatrix}$$

We find the dual problem under its expected formulation with

$$A = \begin{pmatrix} X^T \\ -X^T \end{pmatrix} \in \mathbb{R}^{2d \times n} \qquad B = \begin{pmatrix} \lambda I_d \\ \lambda I_d \end{pmatrix} \in \mathbb{R}^{2d} \qquad Q = \frac{1}{2}I_n \qquad p = y$$

2)

We first define some utility functions for the rest of the exercise

```
def f0(Q, p, v):
  return v@Q@v + p@v


def f(Q, p, A, b, t, v):
    in_log = (b - A@v)
    if np.any(in_log<0):
        return np.inf
    return t*(v@Q@v) + t*p@v - np.sum(np.log(in_log))


def grad_f(Q, p, A, b, t, v):
    denom = 1 / (b - (A@v)) # (2p,)
    return 2*t*(Q@v) + t*p + np.einsum('ij,i->j', A, denom)


def hess_f(Q, p, A, b, t, v):
    denom = 1 / (b - A @ v)**2 # (2p,)
    A_matrices = np.einsum('ij,ik->ijk', A, A) # (2p, n, n)
    sum_matrices = np.sum(denom[:,None,None] * A_matrices, axis=0) # (n, n)
    return 2*t*Q + sum_matrices
```

Below are provided the codes for the centering step and the barrier method

```
def centering_step(Q, p, A, b, t, v0, eps):
  v_arr = [v0]
  v = v0.copy()

  grad = grad_f(Q, p, A, b, t, v) # (n,)
  H_inv = np.linalg.inv(hess_f(Q, p, A, b, t, v)) # (n, n)
  delta_v = -H_inv @ grad
  lamb_2 = -grad@delta_v

  # Backtracking line search
  def back_line_search(delta_v, beta=0.9, alpha=0.1, max_iter=500):
      step = 1
      while f(Q, p, A, b, t, v + step*delta_v) > f(Q, p, A, b, t, v) \
            -alpha*step*grad@delta_v:
          step *= beta
      return step

while lamb_2 / 2  > eps:
    best_step = back_line_search(delta_v)
    v += best_step * delta_v
    v_arr.append(v)

    grad = grad_f(Q, p, A, b, t, v) # (n,)
    H_inv = np.linalg.inv(hess_f(Q, p, A, b, t, v)) # (n, n)
    lamb_2 = -grad @ delta_v
    delta_v = -H_inv @ grad

return v_arr
```

```python
def barr_method(Q, p, A, b, v0, eps, mu):
  m = A.shape[0]
  v_arr = [v0]
  v = v0.copy()
  t = 1
  while  m/t >= eps:
      v_center = centering_step(Q, p, A, b, t, v, eps)
      v_arr+=v_center
      t *= mu
      v = v_center[-1]
  return np.array(v_arr)
```

3)

X and y were generated using with normal distribution. We set $\varepsilon = 10^{-7}, n = 20, d = 300$ and $\mu \in \{5, 15, 50, 100, 500, 400, 2000, 5000\}$. The random seed is fixed to 0.

Here is the code used to generate data and plots

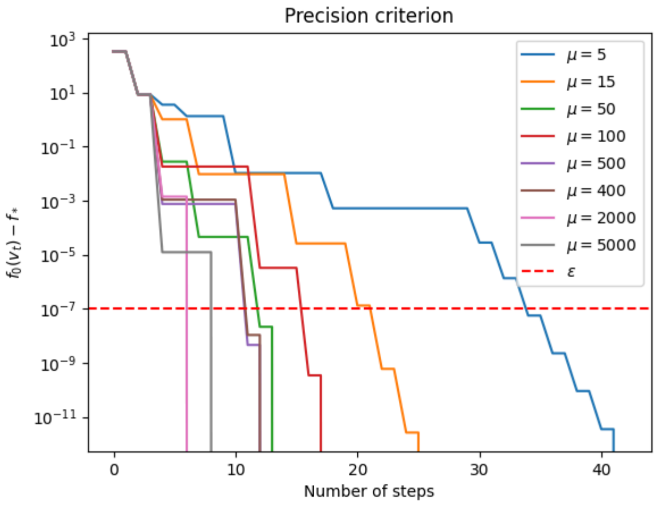
```python
def init_params(n, d, l):
  """Utility function to initialize the problem"""

  X = np.random.normal(size=(n, d))
  Q = 0.5 * np.eye(n)
  p = np.random.normal(size=n) # =y
  A = np.vstack([X.T, -X.T])
  b = l * np.ones(2*d)
  v0 = np.random.normal(size=n)
  eps = 1e-7
  return Q, p, A, b, v0, eps

n = 10
d = 300
lamb = 10
Q, p, A, b, v0, eps = init_params(d, n, lamb)

all_v_arr = []
mu_vals = [5, 15, 50, 100, 500, 400, 2000, 5000]
for mu in tqdm(mu_vals):
  v_arr = barr_method(Q, p, A, b, v0, eps, mu)
  all_v_arr.append(v_arr)

plt.figure()
plt.title('Precision criterion')
for mu, v_arr in zip(mu_vals, all_v_arr):
    f_vals = []
    for v in v_arr:
        f_vals.append(f0(Q, p, v))
    plt.plot(np.array(f_vals)-np.min(f_vals), label=f'$\mu=${mu}')
    plt.ylabel('$f_0(v_t)-f_*$')
    plt.xlabel('Number of steps')
    plt.yscale('log')
plt.axhline(eps, ls='--', c='r', label='$\epsilon$')
plt.legend()
plt.show()
```



We notice that up to a certain value of μ, the higher the faster. However, $\mu = 2000$ requires fewer steps than $\mu = 5000$.

$\boxed{\text{In this case } \mu = 2000 \text{ is an appropriate value}}$
