Master M2 MVA 2015/2016 - Convex Optimization - Homework 3 : The LASSO problem

By: Lilian Besson (`lilian.besson at ens-cachan.fr`).

Abstract:

- I implemented everything for the first two parts, and got good results for both the sub-gradient and coordinate-descent methods. I tested for several values of n, d, λ on randomly generated LASSO problems and the obtained precision was satisfactory.
- I did the complete proof for part 1, and implemented everything, but failed to finish debugging the centering step for the barrier method (it quickly becomes infeasible and diverges).
- For the optional part 3, I did the theoretical part almost entirely (two justifications are missing) but did not have time to start the implementation.

Attached programs: The programs for this TP are included in the `zip` archive I sent, and are regular **Python** programs. They require the usual scientific Python modules (`numpy`, `matplotlib`), and for some comparison I used the the machine-learning module `scikit-learn` and the optimization toolbox scipy.optimize.

Notations

In this homework, we are exposing different approaches to numerically solve the following optimization problem, called LASSO[1]:

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

The variable is $w \in \mathbb{R}^d$ (d is the number of features), and the data are $X = (x_1^T, \dots, x_n^T) \in \mathbb{R}^{n\times d}$ (the design matrix), and $y = (y_1, \dots, y_n) \in \mathbb{R}^n$ the real values associated to each data point x_i (n is the size of the sample). $\lambda \geqslant 0$ is the regularization parameter[2].

---

1 Second order method for the dual problem

1.1 Dual problem of LASSO

In this first sub-section, we present the dual problem of LASSO, and give a complete proof.

Theorem 1. *The dual problem of LASSO can be written in the form a general Quadratic Program:*

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

in the variable[3] $v \in \mathbb{R}^n$*, and with a PSD matrix* Q *(*$Q \succeq 0$*).*

[1] Which stands for Least Absolute Shrinkage and Selection Operators.

[2] Note that $\lambda = 0$ is allowed, it just makes LASSO equivalent to usual Least Squares, but the methods presented here are less efficient that usual Least Squares methods.

[3] Note the change of dimension, from d for the primal to n for the dual.

Proof. The *primal* problem is

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

The first step is to add a dummy variable $r = Xw - y$ into the l_2-norm:

$$\min_{w,r} \frac{1}{2}\|r\|_2^2 + \lambda\|w\|_1, \text{ subject to } r = Xw - y.$$

Now this is a (convex) optimization problem with *equality constraints*, and we can write its *Lagragian* $\mathcal{L}(w, r, v)$ (v is the vector Lagrange multiplier associated with $r = Xw - y$):

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

As usual, we try to group the terms with w and r (optimization variables):

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

So from here, we derive the dual function $g(v) = \inf_{w,r} \mathcal{L}(w, r, v)$ by separating the two minimization over w and r:

$$g(v) = \Big(\inf_w \lambda\|w\|_1 + v^T Xw\Big) + \Big(\inf_r \frac{1}{2}\|r\|_2^2 - v^T r\Big) - (y^T v).$$

For the minimization over w, we use the conjugate function of the l_1-norm, which is the dual norm $\|\cdot\|_\infty$. The inequality constraint of the QP appears:

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

For the minimization over r, we use the conjugate function of the l_2-norm squared. The quadratic term of the QP appears:

$$\inf_r \Big(\frac{1}{2}\|r\|_2^2 - v^T r\Big) = -\sup_r \Big(v^T r - \frac{1}{2}\|r\|_2^2\Big) = -\frac{1}{2}v^T v$$

Finally, reassembling all these terms, we have that:

$$g(v) = -\Big(\frac{1}{2}v^T v\Big) - (y^T v) \text{ subject to } \|X^T v\|_\infty \leqslant \lambda.$$

We can conclude by rewriting the maximization problem of $g(v)$ as a minimization problem (QP):

- with[4] a *quadratic term* $v^T Q v = \frac{1}{2}v^T v$, so let $Q \overset{\text{def}}{=} \frac{1}{2}\mathbb{I}_n$ is simply diagonal (so is indeed PSD),
- with a *linear term* $y^T v$, so let $p \overset{\text{def}}{=} y$,
- and with $2n$ *inequality constraints* $X^T v \leqslant \lambda \mathbf{1}_n$ and $(-X)^T v \leqslant \lambda \mathbf{1}_n$, so[5] let $A = [X^T; -X^T]$, and let $b = \lambda \mathbf{1}_{2n}$ is a constant vector of size $2n$.

And so the dual problem is $\min_v \big(v^T Q v + p^T v\big)$ subject to $Av \preceq b$, as wanted. □

[4] These expressions are exactly the one used in the `lasso` function, which compute `Q,p,A,b,v0` from `X,y` in order to use the `barr_method` function.

[5] A is called the *extended design matrix* (X and $-X$ horizontally stacked).

Note: To go from the dual solution v^* to the primal solution w^*, we have to solve this linear system:

$$w = (X^T X)^{-1}(Xy - Xv).$$

Remark that it looks very similar to the normal equations closed form solution to regular Least Squares problem: $w_{LS} = (X^T X)^{-1}(Xy)$, we just add a term v (dual variable).

Also note that, as usual, the dimension of the problem variable changes, it goes from $w \in \mathbb{R}^d$ to $v \in \mathbb{R}^n$ (and usually $n \ll d$) so it's a useful reduction of dimension.

1.2 Implementation of the barrier method to solve QP

In `linesearch.py`, we implemented[6] Armijo's backtracking line search, and a few utility functions to check that Q is PSD `isPSD`, and to check that v_0 (resp. x_0) is strictly feasible (resp. a valid starting point): `satisfyInEqConstraint` (resp. `isInDomain`).

In `second_order_method.py`, we started by implementing Newton's method[7] in a general setting with blackbox `f`, and `gradientf` for Δf and `hessianf` for $\Delta^2 f$, the function `newton_method` returns `x, x_seq, nbstep`: the optimal solution x_t^*, the sequence of values, and the number of Newton steps.

For the Newton method, we have to inverse a linear system: $(\Delta^2 f(x))\delta = -\Delta f(x)$.
Below is included the key part of our implementing of Newton's method, showing what happens at each time step:

```
# Newton's method loop
while lambdax_sq > 2*eps and nbstep < nbstepMax:
    gradientfx = gradientf(x)
    # We solve Hf(x) delta = -Df(x)
    Hx = hessianf(x)
    # 1. Solve the linear system Hf(x) \ -Df(x)
    delta = - la.lstsq(Hx, gradientfx)[0]
    lambdax_sq = np.dot(gradientfx, -delta)
    # 2. Line search (backtracking line search with Armijo rule)
    t = line_search(f, x, delta, gradientfx, alpha, beta)
    # 3. Update the point x
    x += t * delta
    x_seq.append(x.copy())  # Add a point to our sequence
    nbstep += 1  # Just to be cautious, no infinite loops
# End of the while loop
```

Then for `centering_step`, we use the Newton's method on the QP function:

- $f_t(v) = tf(v) + \phi(v) = t\Big(v^T Q v + p^T v\Big) + \phi(v)$ (defined as `f`), with $\phi(v) = \sum \log(b_i - A_i v)$ the log-barrier function for inequality constraints $A_i v \leqslant b_i$ (representing $Xv \leqslant \lambda$ and $Xv \geqslant \lambda$).
- This gives an exact formula for the gradient (defined as `gradientf`): $\Delta f_t(v) = t\Big(2Qv^T + p\Big) - \sum_i \frac{1}{A_i v - b_i} A_i$,

[6] All the given files are well commented and documented, and they should be clear to understand and work with if needed, however they are kinda long.

[7] Remark: I wrote that part a while ago (10 days before the DM3 was out, along with rank 1 methods and DFS and BFGS), and I tested them on simple 1-D, 2-D and 10-D cost functions and they seemed to work well.

- and for the Hessian (defined as `hessianf`): $\Delta^2 f_t(v) = tQ + \sum_i \frac{1}{(A_i v - b_i)^2} A_i \cdot A_i^T$.

This function returns `v_seq, nbstep`: the sequence of dual variables iterates (from v_t to v_{t+1}) and the number of Newton steps (added).

Then we wrote `barr_method` thanks to the centering step. The initial value for the barrier method parameter $t(0)$ is not cleverly guessed, we chose $t(0) = 1$ every time[8].

Finally, in order to be able to solve a LASSO problem from the primal X, y data and not the dual data Q, p, A, b, we wrote a wrapper function `lasso_barr_method`, which uses what we computed in the proof above to get the dual data from the primal data. It then call the `barr_method`. Note that the `barr_method` also return a sequence of dual values v: `v_seq`, which is the concatenation of all the sequences returned by internal Newton's steps (which is why we observed in class an error plot constant by parts, during each Newton's steps the global objective does not change).

We are not sure about the starting value v_0 though. The main issue I faced was to find a started point v_0 for `barr_method`. I should have spent more time on the *phase one method* but ran out of time. From what I understood, choosing $v_0 = 0$ works, because $Av = 0 \prec b$ for the LASSO dual, indeed $b = [\lambda, \dots, \lambda]$ of size $2n$ and $\lambda > 0$.

Note: we added two counters, one for counting the total number of inner steps (Newton steps), and the number of outer step (centering), in order to check what we saw in class: a big μ leads to a lot of inner steps and few outer steps, while a small μ leads to few inner steps but a lot of outer steps.

The requested experiments are done in the demo function `demo_lasso`.

1.3 Testing on randomly generated LASSO problems

To test our barrier method for the LASSO problem, we chosed to generate random matrices X and observations y (and $\lambda = 10 =$ `lmbda` was asked). In order to be realistic and to be able to check the correctness of the learned (primal) parameter w, I implemented a function `random_X_y_w` (in the `helpers.py` file) where I chosed to generate a matrix X of integers (in $[-10, 10]$) and a "true" w of integers (in $[-20, 20]$). Then we got w' artificially sparse, by removing 2/3 of its values, and finally the observations are produced as $y = Xw'$: only in this context it is meaningful to try to learn $\hat{w} \simeq w'$ (and because we started by generated w' we can compare them). In the experiments (see the `second_order_method.txt` output file for more details), we added one line `Estimation of the error: || w - our_w || = ...` (estimating the distance between the "true" w and the learned one). We worked with $n = 10, d = 20$, but the methods scaled to much bigger dimensions ($n = 50, d = 800$ worked as well, just slower).

To plot the required graph, we wrote `plot_dual_objectives(f, v_seq, name, n, d, mu, lmbda)`. We should have plotted both the function values $((f(v_t))_t)$ and (dual) gap $((f^\star - f(v_t))_t)$.

To chose a good value for the barrier parameter μ, we compared the performance for different values ($\mu = 2, 15, 50, 100$, keeping $\lambda = 10$). The basic idea is that if $\mu \gg 1$, there will be a lot of inner steps (Newton) and few outer steps (centering), but if $\mu \ll 1 or \simeq 1$ it will be the opposite. My guess was $\mu = 15$ for the best value.

> Unfortunately, I failed to conclude the code for this part.
> I literally spent hours trying to conclude and debug my centering steps for no result.
> In practice the inequality constraints for the barrier (the log-barrier function $\phi(v)$) become unsatisfied very quickly, and as soon as it appeared, the next centering fail (it tries to minimize

[8] The slides said that there is "several heuristics for choice of t(0)" but none what discussed except $t(0) = 1$, and a quick lookup on Internet didn't help

f_t with initial value $+\infty$).
Please take some time to check on the submitted programs that I am not lying: I really implemented every piece of the required functions, but something is just not working. Everything seems to work, every piece is here, has been checked several times.

---

2 First order methods for the primal problem

In this section, we work on two methods that try to directly minimize the primal problem. In high-dimension settings, when $n \ll d$, the dual method should work better (because it minimize a cost in a really smaller dimension n).

2.1 Sub-gradient

I implemented the sub-gradient descent on the primal LASSO problem, as a function `subgrad(X,y,lmbda,eps)`. To compute *a* sub-gradient at each step, I wrote `subgrad_objective_from_data(X, y, lmbda)`, which compute a sub-gradient $g(w) \in \text{subgrad}(f)(w)$ with this piece of code, which implements that formula[9] (with a small tolerance `tol` $= 10^{-8}$ when comparing w_i to 0):

$$g(w) = X^T(Xw - y) + \sum_{w_i \neq 0} \text{sign}(w_i)e_i + \sum_{w_i \neq 0} \beta_i e_i \quad \text{for } \beta_i \in [-1, 1] \text{ any values}$$

```
def subgrad_f(w):
    """ Subgradient oracle w --> ONE subgradient of f at w."""
    r = np.dot(X, w) - y
    non_zero_part = np.sign(w) * (np.abs(w) >= tol)
    # For that part, we have the choice (random signs really works ↩
       better)
    beta = (2 * np.random.randint(0, 2, np.shape(w)) - 1)
    # We can also try random_values in [-1, 1]
    # beta = np.random.uniform(-1, 1, np.shape(w))
    zero_part = beta * np.ones_like(w) * (np.abs(w) < tol)
    return np.dot(X.T, r) + lmbda * (non_zero_part + zero_part)
```

For the β_i values, I tried both random signs ($\in \{-1, 1\}$) and random values ($\in [-1, 1]$ uniformly). Performance appeared to not be modified, so I kept the simple model (random signs).

I also tried the 4 step-size strategies seen in class:

- constant step-size $\alpha_k = h$,
- constant step-length $\alpha_k = \frac{h}{\|r_k\|}$ (ie. normalized gradient descent),
- sigma step-size sequence (non-negative, square summable but not summable), starting with value $\alpha_0 = 0.1$ (`sigma_step_size`), for instance $\alpha_k = \frac{0.1}{/} k + 1$,

[9] See [Nesterov], Ex.5 page 133 for the proof.

- a non-summable diminishing[10] positive step-size sequence (`decreasing_step_size`).

The only strategy which works was the constant step-size strategy. The other one diverged clearly (the primal vector w become filled with `nan` very quickly). I tried to fix it, but ran out of time. Even with a very small initial value (order of magnitude of r_k to emulate the behavior of normalized gradient descent), none of the 3 other strategies worked.

For the experiments (see the `first_order_methods.txt` output file for more details), , we used the same kind of random X, y from part 1. But our implementation worked way better in this part, so we could try with higher dimensions (up-to $d = 800$ and $n = 200$ it worked fine, also for $d = 800$ but small $n = 5$ or $n = 10$).

For this constant step-length strategy, we plotted below the (primal) objective as a function of iterations $((i, f(w_i))_{i=1...T}$), see Figure 1:

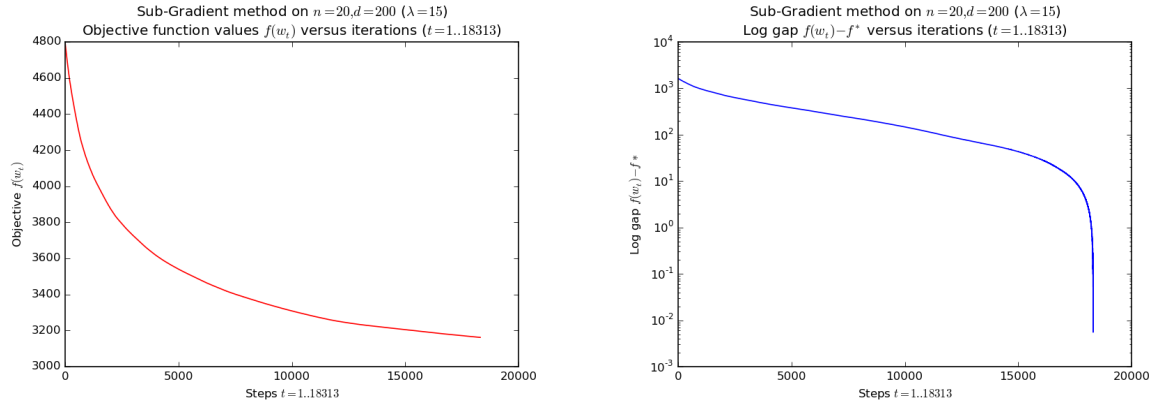

Figure 1: Objectives $f(w_i)$ and gap $f^\star - f(w_i)$ for the Sub-Gradient method, $n = 20, d = 200, \lambda = 15$.

Another similar result, for different dimensions, is also plotted, see Figure 2. The primal gap is less satisfactory that the one obtained for smaller n (we do not even obtain a gap of $\varepsilon = 10^{-3}$), and I think this is logical since the whole point of LASSO is to work well when $n \ll d$.

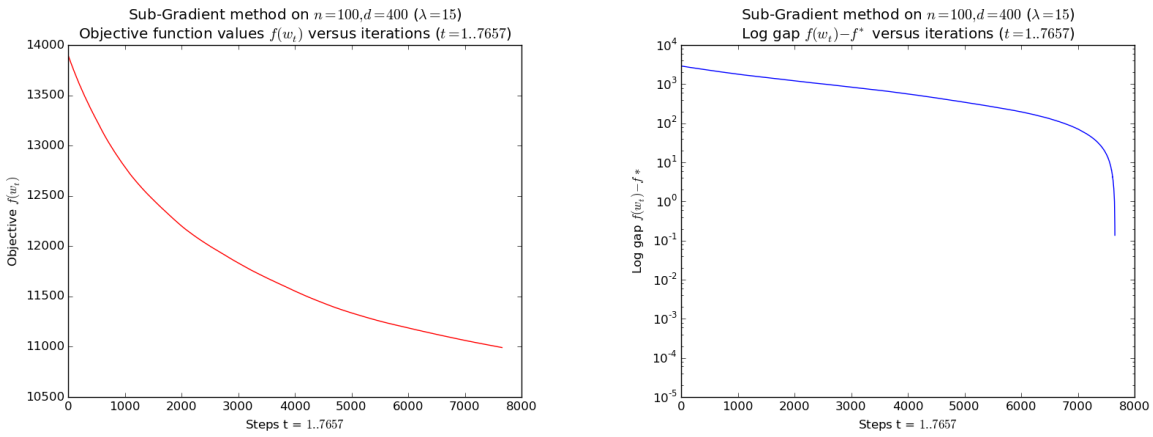

Figure 2: Objectives $f(w_i)$ and gap $f^\star - f(w_i)$ for the Sub-Gradient method, $n = 100, d = 400, \lambda = 15$.

We can be happy about these results, I guess. A comparison with coordinate-descent is done below.

[10] I did not tried with a sequence other than the $\frac{1}{k+1}$ one.

2.2 Coordinate descent gradient

I implemented the sub-gradient descent on the primal LASSO problem, as a function `coord_descent(X,y,lmbda,ep` For each step, we use `onedimension_minimization` to optimize the (primal) cost function according to only one dimension, ie. by optimizing $f_i : t \mapsto f(w_1, \dots, w_{i-1}, t, w_{i+1}, \dots, w_d)$, with a cycle on the coordinates $i = 1, \dots, i = d$ (`i = nbstep % d` in Python), In order to be efficient and win some time here, I first used a one-dimension minimizer from `scipy.optimize` (`minimize_scalar`).

Afterward I computed a closed form formula:

$$\text{Residual: } r = X_i^T \cdot (y - [w_1, \dots, w_{i-1}, 0, w_{i+1}, w_d])$$

$$\text{and } w_i \leftarrow \text{sign}(r) \frac{\max(|r| - \lambda, 0)}{\|X_i\|_2^2}.$$

```
# 2. Improve current point w_k+1 from w_k by changing only its i^th coordinate
# Cheating: w[i] = onedimension_minimization(f, w, i, eps=eps, nbstepMax=nbstepMax)
# Better method (closed form minimizer):
tmp = w.copy()
tmp[i] = 0.
residual = np.dot(X[:, i], y - np.dot(X, tmp).T)
w[i] = np.sign(residual) * np.fmax(np.abs(residual) - lmbda, 0) / np.dot(X[:, i], X[:, i])
```

For the experiments, we used the same kind of random X, y as before, and the same dimensions (n, d) were tried.

For this constant step-length strategy, we plotted below the (primal) objective as a function of iterations $((i, f(w_i))_{i=1\dots T})$, see Figure 3:

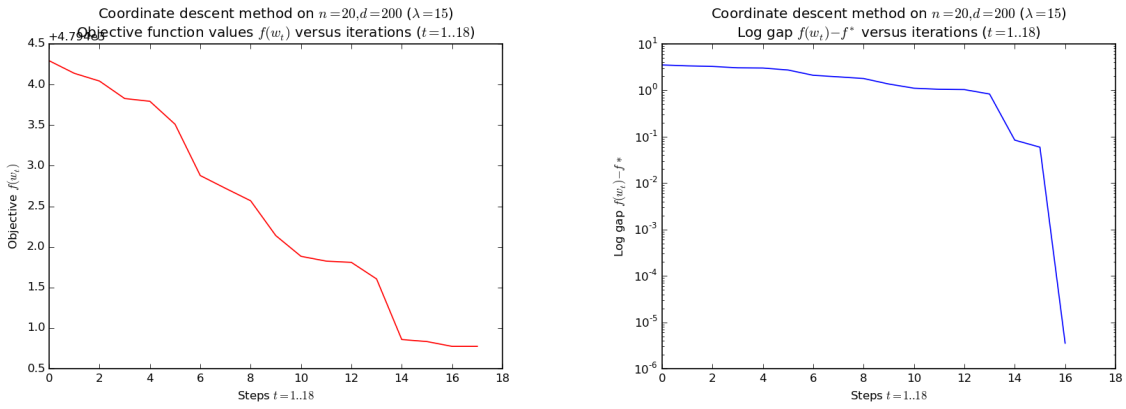

Figure 3: Objectives $f(w_i)$ and gap $f^\star - f(w_i)$ for the Coordinate Descent, $n = 20, d = 200, \lambda = 15$.

Another similar result, for different dimensions, is also plotted, see Figure 4. The primal gap is less satisfactory that the one obtained for smaller n (we do not even obtain a gap of $\varepsilon = 10^{-3}$), and I think this is logical since the whole point of LASSO is to work well when $n \ll d$.

Remark: I think something went wrong in the coordinate-descent, because in some examples it appeared to converge in 4 steps or 17 steps (really weird when we try to learn each coordinate of a

$d = 800$ sized vector w one by one...). Again, the homework was very long and I had to conclude without being able to understand this issue.

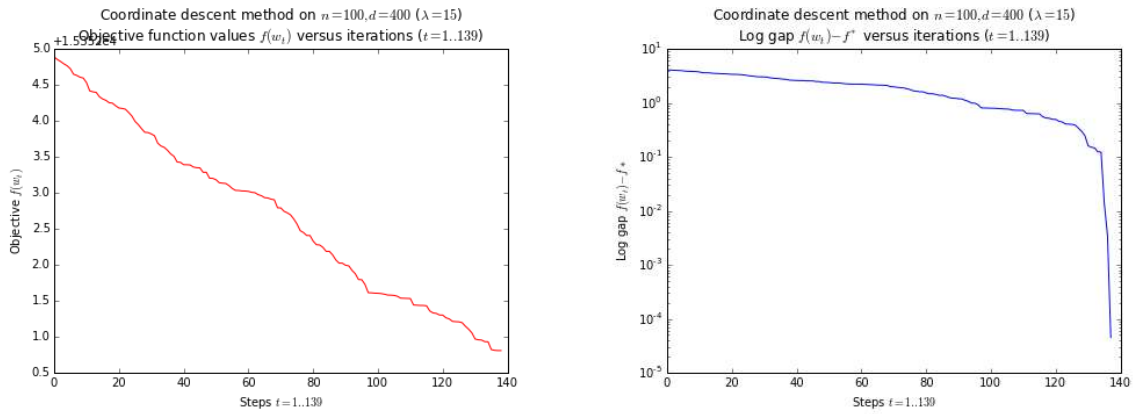

Figure 4: Objectives $f(w_i)$ and gap $f^\star - f(w_i)$ for the Coordinate Descent, $n = 100, d = 400, \lambda = 15$.

We can be quite happy of the results, this method converges in a small number of steps (about 140). And the dual gap becomes small enough (10^{-4} or 10^{-5}).

Comparing sub-gradient and coordinate-descent methods I tried the final question, to compare the two methods. In practice, in terms of iterations, coordinate-descent is clearly better (as we can see with the number of iterations in the graphs above, about 5000 vs 140). In terms of CPU time, coordinate-descent is also better.

Both methods succeed to obtain a gap of 10^{-3}, but sub-gradient failed to be better than 10^{-3}, while the other worked for up-to 10^{-6}. Aiming at 10^{-10} was just a dream, it didn't work at all.

There is many more plots of sub-gradient and coordinate-descent methods, on various size of problems (see the `fig/` folder in the zip file I sent).

I also try to change λ in this part, and I did not observe the required behavior: for both methods, the sparsity of w did not seem to really be controled by the value of λ. Some extra plots for other values of λ ($= 1, 50, 200$) are also included.

---

3 Proximal methods for the primal problem

For this optional part, I am sorry and sad to not have been able to work on it enough.

Question 1

For LASSO (of any size), the "smooth" convex function $f(x)$ is $f(w) = \frac{1}{2}\|Xw - y\|_2^2$, which is indeed convex and differentiable (so it's smooth as excepted). As for its strong convexity, it obviously depend on X: if $X = 0$, the function is constant, hence not strongly convex.

I would want to say that f is strictly convex $\Leftrightarrow$ X is non-singular, but have no proof for this claim. For $n \ll d$, X cannot be non-singular anymore, and we loose the strong convexity.

However the smoothness always holds: If $f(w) = \frac{1}{2}\|Xw-y\|_2^2$, $\Delta f(w) = Xw$, so $\|\Delta f(w) - \Delta f(v)\| = \|X(w-v)\| \leqslant \|X\|_{op}\|w - v\| (*)$ where $\|X\|_{op}$ is the matrix norm (operator norm induced induced by $\|\cdot\|$ on vectors). Then if $L \overset{\text{def}}{=} \|X\|_{op}$, f is L-smooth from this computation $(*)$ (by definition). In

practice, because $\|w\| = \|w\|_2$ the norm 2, the induced matrix norm is the Fröbenius norm (spectral norm), usually also written $\|X\|_2$.

Conclusion: $L \overset{\text{def}}{=} \|X\|_2$ the spectral norm of X is a good smooth-parameter for f for the LASSO problem.

Question 2

Because we assume h to be convex (but maybe non-differentiable), the minimization problem defining $\text{prox}_{h,P}$ is convex, and so have a solution (ie. the proximal operator is well defined).

For an indicator function of a convex set, $\mathcal{C}$: Let $h = \mathbf{1}_{\mathcal{C}}$: $h_{\mathcal{C}}(x) = 0$ if $x \in \mathcal{C}$, $+\infty$ otherwise.

So the $\text{argmin}_z (\cdots + h(z))$ is only to be considered for $z \in \mathcal{C}$, and then $\frac{P}{2} \text{argmin}_{z \in \mathcal{C}} \|z - x\|_2^2 = \frac{P}{2} \text{dist}_2(x, \mathcal{C})^2$ the distance between x and the convex set.

Conclusion: For an indicator of a convex set $\mathcal{C}$:

$$\text{prox}_{\mathbf{1}_{\mathcal{C}},P}(x) = \frac{P}{2} \underset{z}{\text{argmin}} \, \text{dist}_2(x, z)^2.$$

Note that this distance might not be easy to compute in general, but it's still easier than a general proximal operator (for a general h). If $\mathcal{C}$ is closed, $\text{dist}_2(x, \mathcal{C}) = \|x - \pi_{\mathcal{C}}(x)\|_2^2$ and $\text{argmin}_z \, \text{dist}_2(x, z) = \pi_{\mathcal{C}}(x)$ for $\pi_{\mathcal{C}}$ the projection operator onto $\mathcal{C}$. So

$$\text{prox}_{\mathbf{1}_{\mathcal{C}},P}(x) = \frac{P}{2} \pi_{\mathcal{C}}(x).$$

For $h(x) = \|x\|_1$ For the l^1-norm, the argmin looks very similar to the one in LASSO, with $n = d$, $X = \mathbb{I}_n$ and $w = z$, and with $\lambda = \frac{2}{P} > 0$.

We use another characterization[11] of the proximal operator:

$$w = \text{prox}_{h,P}(x) \Leftrightarrow 0 \in \delta(h(w) + \frac{P}{2}\|w - x\|_2^2) \Leftrightarrow 0 \in \delta h(w) + P(w - x) \Leftrightarrow w \in x - \frac{1}{P}\delta h(w).$$

The i-th coordinate of the set of sub-gradients (the differential) for $\|\cdot\|$ was given above (part 2.1), and it's $\delta(h(w))_i = \delta\|w\|_i = \text{sign}(w)$ if $w_i \neq 0, [-1, +1]$ if $w_i = 0$.

So with this other characterization of $\text{prox}_{h,P}(x)$, and the specific form of $\delta h(u)$, coordinate-wise, we can conclude.

Conclusion: in this case, the proximal operator will be the "soft-threshold" operator (seen in class as the "shrinkage" operator):

$$\text{prox}_{\|\cdot\|,P}(x)_i = P \max(\|x_i\|/P - 1, 0) = \max(|x_i| - P, 0).$$

Question 3

The question should be understood: with x being fixed, for which values of $M > 0$ is $g_{x,M}(z)$ an upper bound on $\phi(x)$ (fixed value) for all z?

To answer it, let's work by equivalence, by starting to developp these two terms:

$$\begin{aligned} \forall z, g_{x,M}(z) \geqslant \phi(x) &\Leftrightarrow \forall z, \Delta f(x)^T(z - x) + \frac{M}{2}\|z - x\|_2^2 + h(z) \geqslant h(x) \\ &\Leftrightarrow \forall z, (-\Delta f(x)^T)(z - x) \leqslant \frac{M}{2}\|z - x\|_2^2 + h(z) - h(x) \end{aligned}$$

[11] See here for a more detailed proof.

I have no idea how to conclude, but my intuition is that choosing $M = P = \frac{1}{L}$ is the best choice.

Then the iteration scheme $x_{t+1} = \text{argmin}_z \, g_{x_t,M}(z)$ becomes:

$$x_{t+1} = \text{prox}_{h,P}(x_t - M\Delta f(x_t)).$$

- For $h = 0$, $g_{x,M} = f(x) + (\Delta f(x))^T(z - x)$ will give the iteration scheme of the **usual gradient descent** on f: $x_{t+1} = x_t - \delta_t \Delta f(x_t)$ (with a step-size δ_t, constant equal to M).
- For $h = \mathbf{1}_{\mathcal{C}}$, similarly, based on the form of the proximal operator (computed above as the projection onto $\mathcal{C}$), the proximal iteration scheme will give the iteration scheme of the usual **projected gradient descent**[12]:

$$x_{t+1} = \pi_{\mathcal{C}}(x_t - \delta_t \Delta f(x_t)).$$

Question 4 and 6

I really did not have the time to implement or work on this part, sorry.

Question 5

We just have to observe that $x^\star$ is an minimizer of the initial problem $\min_x \phi(x) = f(x) + h(x)$ if and only if $x^\star = \text{prox}_{P,h}(x^\star - P\Delta F(x^\star))$. So $x^\star$ is a fixed-point of a certain operator Ψ, namely $x \mapsto \text{prox}_{P,h}(x - P\Delta F(x))$.

And in fact, the proximal iteration scheme derived in question 3 is exactly a fixed point iteration procedure: $x_0 \in \Omega, x_{t+1} \leftarrow \Psi(x_t)$. We do not know from which x_0 it starts, but if the proximal operator is proven to be contractant, the iteration scheme will converge (quadratically) from any starting point.

I have no idea how we could prove the contraction property for Ψ though.

Conclusion

The last homework was interesting, thanks, but it was very lengthy and quite time consuming, so many points were left unconcluded (or untouched for the end of optional part 3).

References

- Course and slides by Alexandre D'Aspremont: Newton's method with equality constraints (pages 19 and 41), Barrier methods (page 13), First Order methods part 1 and part 2.
- Wolfe conditions (on Wikipedia) for the Armijo rule.

[12] Which can be solved with Uzawa's method for simple convex domains.