Emmarius DELAR
emmarius.delar@dauphine.eu
21 Mars, 2025

Non-convex inverse problems:
project Sparse PCA

Each function is implemented in the `utils.py` script. Each figure was generated by a script based on the functions previously defined in `utils.py`, and the name of the script associated with each figure will be explicitly mentioned.

To determine and solve the optimization problem, I used the code provided for the practical sessions of the Non-Convex Inverse Problems lecture[1].

1 Question 1.

Write a function which selects a random instance of the considered problem.

It is the `random_instance` function from the `utils.py` script.

```python
def random_instance(k:int, m:int, n:int):
    """_summary_

    Args:
        k (int): cardinality of S1 and S2
        m (int): Number of observation
        n (int): dimension of the matrices

    Returns:
        A : 3D-np.array wich contains all A_i matrices of size nxn
        Y : 3D-np.array wich contains all y_i matrices given by <A_i, X>
    """
    n_2 = n**2
    #selects S1, S2 uniformly
    S1 = random.choice(n, size=k, replace=False)
    S1_C = np.setdiff1d(np.arange(n), S1, assume_unique=True) #Complement of S1
    S2 = random.choice(n, size=k, replace=False)
    S2_C = np.setdiff1d(np.arange(n), S2, assume_unique=True) #Complement of S2
    #Sample u,v
    u = np.multiply(0.01*np.random.randn(n), random.randint(1, n_2, n))
    u[S1_C] = np.zeros(n-k)
    v = np.multiply(0.01*np.random.randn(n), random.randint(1, n_2, n))
    v[S2_C] = np.zeros(n-k)
    # Compute X
    u = u.reshape(n,1)
    v = v.reshape(n,1)
    X = np.dot(u, v.T)
    # Compute A = {A1, .., Am}
    A = random.randn(m*n*n)
    A = A.reshape((m,n,n))
    # Compute Y = {y1, ..., ym}
    f1 = lambda A_i : np.trace(A_i @ X)
    Y = np.array([f1(A[k]) for k in range(m)])

    return X, S1, S2, A, Y
```

[1] Course page

2 Question 2.

Write the considered problem under the form of an optimization problem. Is it convex or non-convex? Justify your answer.

Recover $X \in \mathbb{R}^{n \times n}$ such that :

$$\text{rank}(X) = 1 \tag{1}$$

and,

$$S_1, S_2 \subset \{1, \dots, n\}, \quad \text{card}(S_1) = \text{card}(S_2) = k, \tag{2}$$

$$X_{s_1,s_2} = 0, \quad \forall (s_1, s_2) \notin S_1 \times S_2 \tag{3}$$

and,

$$y_i = \langle A_i, X \rangle, \quad \forall i \in [m] \tag{4}$$

with

$$\forall i \in [m], \quad A_i \sim \mathcal{N}(0, I_n), \quad \text{independent}$$

This optimization problem has two non-convex parts: the objective function 1 and the constraints 3. Consequently, the problem is non-convex.

3 Question 3.

3.1 a) If we had Information 1, but not Information 2, which objective function F from the lectures could you use? We call it F_1.

We could use the nuclear norm as the objective function F_1 because considering only Information 1 leads to a classical low-rank matrix recovery problem.

3.2 b) If we had Information 2, but not Information 1, which objective function F from the lectures could you use? We call it F_2.

If we consider only Information 2, we could use the l_1-norm as the objective function F_2 because the problem becomes a compressed sensing problem.

3.3 c) We choose $F = F_1 + \lambda F_2$ for some appropriate $\lambda \in \mathbb{R}^+$. Implement a function that, given $\lambda, A_1, \dots, A_m, y_1, \dots, y_m$, solves the problem.

It is the `solveur_nuc_l1` function from the `utils.py` script.

```
def solveur_nuc_l1(lbda:float, A:np.array, Y:np.array, beta=1):
    m, n, _ = A.shape
    Z = cp.Variable((n, n))
    F = beta*cp.norm(Z, "nuc") + lbda*cp.norm(Z, 1)
    constraints = [cp.trace(A[k] @ Z) == Y[k] for k in range(m)]
    objective = cp.Minimize(F)
    problem = cp.Problem(objective,constraints)
    # Solve it
    problem.solve(solver=cp.SCS)

    return Z.value
```

3.4 d) Write a function which, given X, S_1, S_2, Z, computes the support recovery error of Z.

It is the `support_recovery_error` function from the `utils.py` script.

```
def support_recovery_error(X, S1, S2, Z):
    # --- Find min_X
    min_X = np.min(np.abs(X[S1, :][:, S2]))
    # --- Support S1 x S2
    supp_S1_S2 = [(s1, s2) for s1 in S1.tolist() for s2 in S2.tolist()]
    supp_S1_S2 = set(supp_S1_S2)
    # --- Support of Z
    supp_Z = []
    # NOT in S1xS2
    indx_sup = np.argwhere(np.abs(Z) > (min_X/10))
    supp_Z.extend(idx for idx in indx_sup.tolist() if not tuple(idx) in supp_S1_S2)
    # in  S1xS2
    indx_inf = np.argwhere(np.abs(Z) < (min_X/10))
    supp_Z.extend(idx for idx in indx_inf.tolist() if tuple(idx) in supp_S1_S2)

    return len(supp_Z)
```

3.5 e) Test your code: take for instance $k = 2$, $n = 4$. From which value of m do you expect the support recovery error to be always 0, for whatever value of λ? Check that this is indeed how your code behaves. What happens for slightly smaller values of m?

The plot in Fig.1 is reproducible by launching the script `test_code.py`.

For $m = n^2$, that leads to have an observation per unknow of X, and then we can expect fully recovere X and have a support recovery error equal to 0, whatever the value of λ.

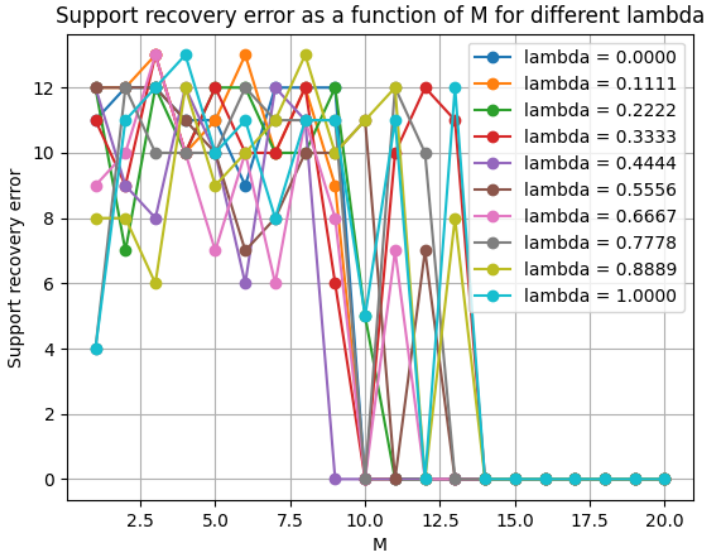

Figure 1: Support recovery error as a function of M

According to Fig.1, this is how the code behaves. Moreover, this figure shows that before reaching $M = n^2$, it can achieve a support recovery error of 0 at a few points where $M < n^2$.

3.6 f) For $k = 3$, $n = 6$, $m = 20$, select a random problem instance. Plot the support recovery error as well as $\|X - Z\|_F$ as a function of λ. Comment the graphs.

The plot in Fig.2 is reproducible by launching the script `support_error_plot.py`.

According to Fig.2, when $\lambda \to 0$, the objective function F is dominated by the nuclear norm, which corresponds to a low-rank solution. As a result, the Frobenius norm remains low, and the support recovery error is small. On the contrary, when $\lambda \gg 1$, the l_1-norm dominates, leading to a sparse solution, which without any constraint on the rank, lead to a high-rank solution.

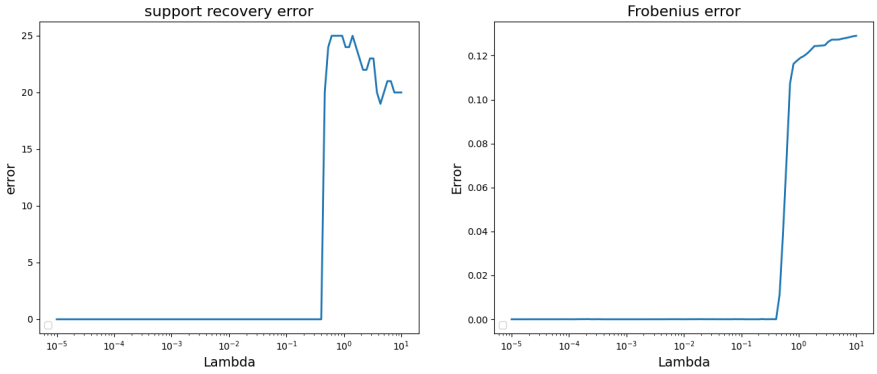

Figure 2: Support recovery error as well as $\|X - Z\|_F$ as a function of λ

3.7 g) Set $n = 20$. For $k = 1, 3, 5, \ldots, 13$, compute approximately the smallest value of m such that Problem (Convex approx.) reaches zero support recovery error for at least one value of λ with probability roughly 50%. Plot this value as a function of k.

The plot in Fig.3 is reproducible by launching the script `plot_smallest_m_F1_F2.py`.

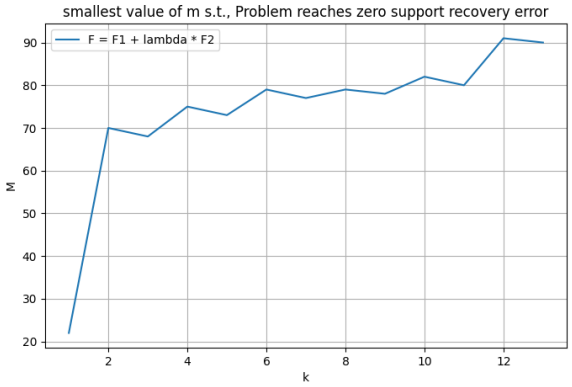

Figure 3: smallest value of m for $F = F_1 + \lambda F_2$

3.8 h) Same question if you use only $F = F_1$ or $F = F_2$ as the objective function. (Note that, in this case, the objective does not depend on λ.)

The plot in Fig.4 is reproducible by launching the script `plot_smallest_m_F1.py`.

As a remark, the algorithm for $F = F_2$ does not reach 50% probability (after 1000 iterations for different values of λ). However, the script is available as `plot_smallest_m_F2.py`.

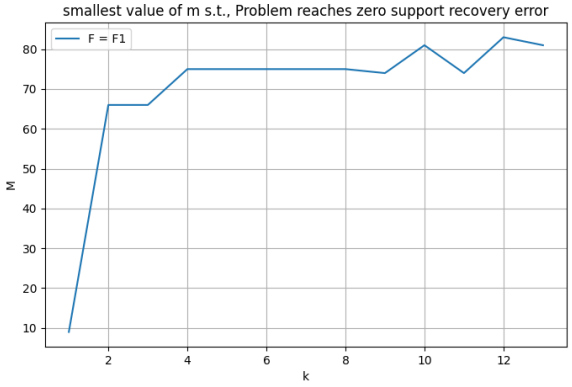

Figure 4: smallest value of m for $F = F_1$

3.9 i) Comment on the plots from the previous two questions.

For $F = F_2$, the optimization is more challenging because it forces the solution to be as sparse as possible without considering the possible structure of S_1 and S_2. Therefore, a large number of observations is needed to converge to a stable solution, i.e., the solution reaches zero support recovery error with a probability of approximately 50%.

On the contrary, when the rank constraint (i.e., F_1) is taken into account, convergence to a solution with zero support recovery error requires fewer observations.

4 Question 4.

4.1 a) Let us set $Z = \frac{1}{m}\sum_{i=1}^{m}\langle A_i, X\rangle A_i$. Compute $\mathbb{E}[Z]$.

$$
\begin{aligned}
\mathbb{E}[Z] &= \mathbb{E}\left[\frac{1}{m}\sum_{i=1}^{m}\langle A_i, X\rangle A_i\right] \\
&= \frac{1}{m}\sum_{i=1}^{m}\mathbb{E}\left[\operatorname{tr}(A_i^T X)A_i\right], \quad \text{since } \operatorname{rank}(X) = 1, \exists u, v \in \mathbb{R}^n \text{ s.t. } X = uv^T \\
&= \frac{1}{m}\sum_{i=1}^{m}\mathbb{E}\left[A_i \operatorname{tr}(v^T A_i^T u)\right] \\
&= \frac{1}{m}\sum_{i=1}^{m}\mathbb{E}\left[A_i (v^T A_i^T u)\right] \\
&= \frac{1}{m}\sum_{i=1}^{m}\mathbb{E}\left[A_i \sum_{k,l}^{n} v_k (A_i)_{k,l} u_l\right]
\end{aligned}
$$

For each component (p, q) of A_i, we have

$$\mathbb{E}\left[(A_i)_{p,q}\sum_{k,l}^{n} v_k(A_i)_{k,l}u_l\right] = \mathbb{E}\left[\sum_{k,l}^{n} v_k(A_i)_{p,q}(A_i)_{k,l}u_l\right] = \sum_{k,l}^{n} v_k\mathbb{E}\left[(A_i)_{p,q}(A_i)_{k,l}\right]u_l$$

Knowing that $(A_i)_{p,q}$ and $(A_i)_{h,\ell}$ are independent $\forall(p,q) \neq (h,\ell)$ and $A_i \sim \mathcal{N}(0, I_n)$, we have :

$$\mathbb{E}\left[(A_i)_{p,q} \cdot (A_i)_{k,l}\right] = \begin{cases} 0, & \text{if } (p,q) \neq (k,l) \\ 1, & \text{if } (p,q) = (k,l) \end{cases}$$

thuse we obtain:

$$\mathbb{E}\left[(A_i)_{p,q}\sum_{k,l}^{n} v_k(A_i)_{k,l}u_l\right] = v_p \times u_q, \quad \forall(p,q) \in \{1,\ldots,n\}^2.$$

Consequently,

$$\begin{aligned}\mathbb{E}[Z] &= \frac{1}{m}\sum_{i=1}^{m}\mathbb{E}\left[A_i\sum_{k,l}^{n} v_k(A_i)_{k,l}u_l\right] \\ &= \frac{1}{m}\sum_{i=1}^{m}\mathbb{E}[X] \\ &= X.\end{aligned}$$

4.2 b) Propose a strategy to approximate S_1, S_2 from Z (knowing k), and implement it.

By the previous question we shown that $\mathbb{E}[Z] = X = uv^T$, for $u, v \in \mathbb{R}^n$. If we extend this form we find:

$$\mathbb{E}[Z] = \begin{bmatrix} u_1v_1 & u_1v_2 & \ldots & u_1v_n \\ u_2v_1 & u_2v_2 & \ldots & u_2v_n \\ \vdots & \vdots & \ddots & \vdots \\ u_nv_1 & u_nv_2 & \ldots & u_nv_n \end{bmatrix}$$

Intuitively, by using the properties of the Law of Large Numbers, we can estimate X by taking the mean of m approximation Z_i, with $m \gg 1$. Hence, we estimate S_1 as the indices of the k rows with the largest l_1-norm and S_2 as the indices of the k columns with the largest l_1-norm.

It is the `support_approximation` function from the `utils.py` script.

```
def support_appoximation(A, Y, k, nb_iter=30):
    m,n,_ = A.shape
    Z = np.zeros((n,n))
    # Estime X
    for _ in range(nb_iter):
        Z += (1/nb_iter)*solveur_nuc_l1(.1, A, Y)
    # S1 : K rows with largest l1-norm
    rows_norm = np.sum(np.abs(Z), axis=1)
    S1 = np.argsort((-1)*rows_norm)[:k]
    # S2 : K rows with largest l1-norm
    cols_norm = np.sum(np.abs(Z), axis=0)
    S2 = np.argsort((-1)*cols_norm)[:k]

    return S1, S2
```

4.3 c) Propose a non-convex algorithm which, given a guess for S_1 and S_2, tries to recover X.

We can reformulate the problem has :
solving :

$$\min_{\mathbf{u},\mathbf{v}\in\mathbb{R}^n} \mathcal{L}(\mathbf{u},\mathbf{v}) = \frac{1}{2m}\sum_{i=1}^{m}\left(y_i - \mathbf{v}^T A_i \mathbf{u}\right)^2$$

such that:

$$u_{s_1} = 0, \quad \forall s_1 \notin S_1 \text{ and } v_{s_2} = 0, \quad \forall s_2 \notin S_2$$

Thus, we can use the following non-convex algortithms based on (S_1, S_2)

Algorithm 1 Non-convex algorithm

Input: $(S_1, S_2, T, \eta, \epsilon)$
Initialisation:
$\mathbf{u}_0 \leftarrow \mathbf{0}_{\mathbb{R}^n}, \quad \mathbf{v}_0 \leftarrow \mathbf{0}_{\mathbb{R}^n}$
for $t = 1$ to T **do**
  $\mathbf{u}_{t+1} \leftarrow \mathbf{u}_t - \eta\nabla_u\mathcal{L}(\mathbf{u}_t, \mathbf{v}_t)$
  $\mathbf{v}_{t+1} \leftarrow \mathbf{v}_t - \eta\nabla_v\mathcal{L}(\mathbf{u}_t, \mathbf{v}_t)$
  Support constraints: $\forall s_1 \notin S_1, \quad u_{s_1} = 0, \quad \forall s_2 \notin S_2, \quad v_{s_2} = 0$
  if $\mathcal{L}(\mathbf{u}_{t+1}, \mathbf{v}_{t+1}) \leq \epsilon$ **then**
    break
  end if
end for
$X = \mathbf{u}_*, \mathbf{v}_*^T$
return X

4.4 d) For $k = 4$, $n = 10$, and various values of m, run the algorithm on several problem instances and print any relevant information on either the reconstructed matrix or the inner behavior of the algorithm. Comment on the results. Try to explain, in the failure cases, why the algorithm fails.

Figure.5, Figure.6 and Fig.7 are reproducible by launching the script `NC_algorithm.py`. This script also contains the implementation of Algorithm 1.

According to Fig. 5, the algorithm is very unstable and highly dependent on the value of M. Moreover, according to Fig. 6, if we have an optimal M, as observed in question 3)e), we can find a learning rate that leads to a low error in terms of the Frobenius norm $lr \approx 0.8$.

4.5 e) Propose an improvement for this non-convex strategy, describe it, and test it.

To improve Algorithm 1, we need to find a better approximation of the supports S_1 and S_2. The performance of our support approximator is based on the law of large numbers, so if $m \ll n$, we cannot obtain a good approximation of the supports and, consequently, do not have a good approximation of X as we can show on Fig.7.

Figure 5: Reconstruction errors as a function of M, for lr=0.01

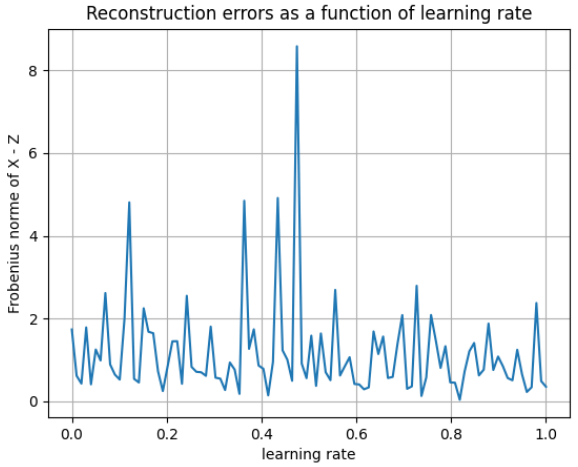

Figure 6: Reconstruction for optimal $m = n^2$ errors as a function of learning rate

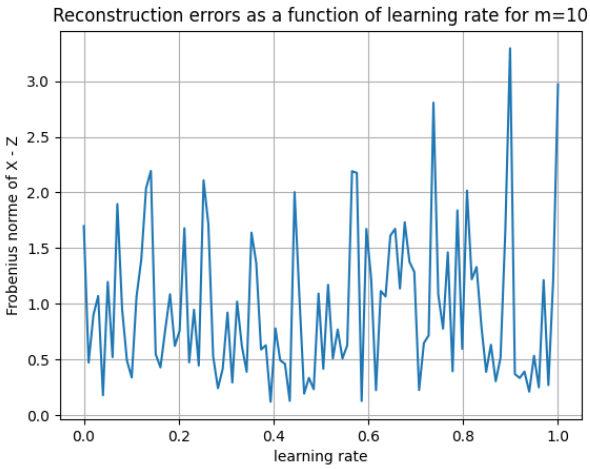

Figure 7: Reconstruction for small $m = 10$ errors as a function of learning rate