

Exercice 1**Que faire avec ChatGPT (et ses amis) ?****8 points**

Voici une conversation entre un *data scientist* et chat GPT à propos d'un problème des régression par morceau. A chacun des quatre étape de la conversation proposez si vous l'estimez judicieux :

1. des améliorations du prompt proposé
2. une analyse critique de la réponse proposée par l'IA générative
3. une proposition d'autres prompts

Le premier prompt suivant à été adressé à chatGPT

Soit X une matrice de données de n individus ordonnés et p variables associé à un vecteur y de n réponses. Peux tu proposer une méthode permettant d'estimer les paramètres d'un modèle linéaire par morceau permettant d'expliquer y connaissant X ?

Sa réponse a à été la suivante (en italique) :

Soit $X \in \mathbb{R}^{n \times p}$ une matrice de covariables associée à n individus ordonnés (par exemple par temps, position, ou rang), et $y \in \mathbb{R}^n$ le vecteur des réponses. On cherche un modèle linéaire par morceaux (segmentation) où les coefficients peuvent changer à certains points de rupture le long de l'ordre des individus.

On suppose qu'il existe K segments contigus, définis par des ruptures

$$1 = \tau_0 < \tau_1 < \dots < \tau_{K-1} < \tau_K = n,$$

tels que, pour tout $k \in \{1, \dots, K\}$ et tout $i \in \{\tau_{k-1}, \dots, \tau_k\}$,

$$y_i = x_i^\top \beta^{(k)} + \varepsilon_i, \quad \varepsilon_i \sim \mathcal{N}(0, \sigma^2) \text{ (optionnel).}$$

Ici x_i^\top désigne la i -ème ligne de X , et $\beta^{(k)} \in \mathbb{R}^p$ le vecteur de paramètres du segment k .

Définissons le coût d'ajustement sur un intervalle contigu $[a, b]$:

$$C(a, b) = \min_{\beta \in \mathbb{R}^p} \sum_{i=a}^b (y_i - x_i^\top \beta)^2.$$

La solution interne est la régression des moindres carrés sur le sous-échantillon :

$$\hat{\beta}_{a:b} = (X_{a:b}^\top X_{a:b})^{-1} X_{a:b}^\top y_{a:b} \quad (\text{si inversible ; sinon pseudo-inverse ou ridge}).$$

On estime ensuite simultanément les ruptures et les paramètres via

$$\min_{K, \tau_1, \dots, \tau_{K-1}} \left\{ \sum_{k=1}^K C(\tau_{k-1}, \tau_k) + \lambda(K-1) \right\},$$

où $\lambda > 0$ contrôle le compromis entre qualité d'ajustement et complexité (nombre de ruptures).

Pour un K fixé, on peut minimiser $\sum_{k=1}^K C(\tau_{k-1}, \tau_k)$ par programmation dynamique. Dans la version pénalisée, on introduit :

$$F(t) = \min_{\substack{K \geq 1 \\ 1 = \tau_0 < \dots < \tau_K = t}} \left\{ \sum_{k=1}^K C(\tau_{k-1}, \tau_k) + \lambda(K-1) \right\}, \quad F(1) = 0.$$

La récurrence de Bellman s'écrit, pour $t \geq 2$,

$$F(t) = \min_{s \in \{1, \dots, t-1\}} \{F(s) + C(s+1, t) + \lambda\},$$

(avec la convention qu'on ne paie pas λ pour le premier segment, gérable par initialisation). On conserve en outre l'argmin

$$\hat{s}(t) = \arg \min_{s < t} \{F(s) + C(s+1, t) + \lambda\}$$

pour reconstruire les ruptures par backtracking depuis $t = n$.

Une fois les ruptures $\hat{\tau}_1, \dots, \hat{\tau}_{\hat{K}-1}$ obtenues, on estime les coefficients segment par segment :

$$\hat{\beta}^{(k)} = \hat{\beta}_{\hat{\tau}_{k-1} : \hat{\tau}_k}, \quad k = 1, \dots, \hat{K}.$$

Choix de λ (ou de K) Plusieurs stratégies sont possibles :

- Critère d'information (type BIC) : par exemple $\lambda \propto p \log n$ pour pénaliser chaque rupture.
- Validation croisée sur blocs contigus (afin de respecter l'ordre).
- Contraintes pratiques : taille minimale de segment $b - a + 1 \geq m_{\min}$.

Remarques pratiques

- Si $X_{a:b}^\top X_{a:b}$ est mal conditionnée, utiliser une version ridge :

$$\hat{\beta}_{a:b}^{(\text{ridge})} = (X_{a:b}^\top X_{a:b} + \gamma I)^{-1} X_{a:b}^\top y_{a:b}.$$

- La complexité naïve est $O(n^2)$ évaluations de $C(a, b)$; on peut accélérer via des techniques de pruning (PELT, etc.) lorsque certaines conditions sont remplies.

Résumé. La méthode consiste à (i) définir un coût de régression linéaire sur tout intervalle contigu, (ii) résoudre une segmentation optimale pénalisée par programmation dynamique pour estimer les ruptures, puis (iii) estimer les coefficients par moindres carrés sur chaque segment.

Le deuxième prompt suivant à été adressé à chatGPT

Et pourquoi pas utiliser le fused lasso pour traiter ce problème ?

Sa réponse a à été la suivante (en italique) :

Oui, le fused lasso est une approche naturelle et souvent très efficace pour estimer un modèle linéaire par morceaux lorsque les individus sont ordonnés. L'idée centrale est de faire varier les coefficients β le long de l'indice $i \in \{1, \dots, n\}$, tout en pénalisant la variation totale afin d'induire des paliers (coefficients constants par segments).

Au lieu d'un nombre de segments K explicite, on introduit une séquence de vecteurs de paramètres

$$\beta_1, \beta_2, \dots, \beta_n \in \mathbb{R}^p,$$

et on modélise

$$y_i = x_i^\top \beta_i + \varepsilon_i, \quad i = 1, \dots, n.$$

Sans contrainte, ce modèle est non identifiable (trop de paramètres). Le fused lasso régularise en imposant que la suite (β_i) soit par morceaux constante.

La version la plus standard est :

$$\min_{\beta_1, \dots, \beta_n \in \mathbb{R}^p} \frac{1}{2} \sum_{i=1}^n (y_i - x_i^\top \beta_i)^2 + \lambda \sum_{i=2}^n \|\beta_i - \beta_{i-1}\|_1.$$

- *Le terme d'ajustement impose la proximité aux données.*
- *La pénalité $\sum_{i=2}^n \|\beta_i - \beta_{i-1}\|_1$ (norme ℓ_1 sur les différences) favorise des différences exactement nulles pour beaucoup de i , donc des plateaux : $\beta_i = \beta_{i-1}$ sur de longs intervalles.*
- *Les indices où $\hat{\beta}_i \neq \hat{\beta}_{i-1}$ jouent le rôle de ruptures.*

Comparaison avec la segmentation pénalisée à ruptures explicites La segmentation classique cherche

$$\min_{\tau, \{\beta^{(k)}\}} \sum_{k=1}^K \sum_{i=\tau_{k-1}}^{\tau_k} (y_i - x_i^\top \beta^{(k)})^2 + \lambda(K-1).$$

Le fused lasso remplace la recherche combinatoire des ruptures par une optimisation convexe (ou quasi-convexe) en introduisant (β_i) et en pénalisant les sauts.

Avantages du fused lasso :

- **Problème convexe** (au moins pour la perte quadratique) : minimisation globale, stabilité numérique, existence d'algorithmes efficaces.
- **Ruptures sélectionnées automatiquement** : pas besoin de fixer K .
- **Chemin de régularisation** : en faisant varier λ , on obtient une hiérarchie de segmentations (du très lisse au très fragmenté).
- **Extensions simples** : pertes robustes (Huber), contraintes, pondérations, etc.

Limites / points d'attention :

- La pénalité ℓ_1 sur les différences induit un **biais de rétrécissement** sur l'amplitude des sauts.
- Les **ruptures peuvent différer par variable** : avec $\|\cdot\|_1$ sur les différences, certaines composantes de β_i peuvent sauter à des positions différentes, ce qui peut être souhaité (effets variables) ou non (on veut des ruptures communes).
- Le coût mémoire/temps peut croître si n et p sont grands (mais des méthodes spécialisées existent).

Si l'on souhaite que les ruptures soient partagées entre toutes les composantes (un vrai modèle par morceaux avec segments communs), on remplace la norme ℓ_1 par une norme de groupe :

$$\min_{\beta_1, \dots, \beta_n} \frac{1}{2} \sum_{i=1}^n (y_i - x_i^\top \beta_i)^2 + \lambda \sum_{i=2}^n \|\beta_i - \beta_{i-1}\|_2.$$

Ici, $\|\cdot\|_2$ favorise des vecteurs de différence entièrement nuls, donc des sauts simultanés sur les p coefficients.

On peut aussi combiner sélection de variables et segmentation :

$$\min_{\beta_1, \dots, \beta_n} \frac{1}{2} \sum_{i=1}^n (y_i - x_i^\top \beta_i)^2 + \lambda_1 \sum_{i=2}^n \|\beta_i - \beta_{i-1}\|_2 + \lambda_2 \sum_{i=1}^n \|\beta_i\|_1,$$

ce qui impose à la fois (i) peu de ruptures et (ii) parcimonie des coefficients.

Comment en déduire une segmentation et des paramètres par segment Une fois la solution $\{\hat{\beta}_i\}$ obtenue, on définit l'ensemble des ruptures estimées :

$$\hat{\mathcal{J}} = \{i \in \{2, \dots, n\} : \hat{\beta}_i \neq \hat{\beta}_{i-1}\}.$$

En ordonnant $\hat{\mathcal{J}} = \{\hat{\tau}_1 + 1, \dots\}$, on récupère des segments contigus, et les paramètres par segment sont simplement la valeur constante prise par $\hat{\beta}_i$ sur ce segment.

Conclusion Le fused lasso fournit une alternative élégante à la programmation dynamique :

- la **segmentation** est induite par une pénalité de variation totale,
- l'estimation se fait via une **optimisation convexe** (perte quadratique),
- la structure par morceaux émerge naturellement via les différences $\hat{\beta}_i - \hat{\beta}_{i-1}$.

C'est donc une méthode particulièrement adaptée lorsque l'ordre des individus est essentiel et que l'on souhaite une estimation automatique et régularisée des ruptures.

Le troisième prompt suivant à été adressé à chatGPT

Peux tu proposer une fonction python permettant d'estimer les paramètres β du fused lasso ?

On considère le problème (version ℓ_1 sur les différences) :

$$\min_{\beta_1, \dots, \beta_n \in \mathbb{R}^p} \frac{1}{2} \sum_{i=1}^n (y_i - x_i^\top \beta_i)^2 + \lambda \sum_{i=2}^n \|\beta_i - \beta_{i-1}\|_1.$$

En empilant β_i dans une matrice $B \in \mathbb{R}^{n \times p}$ (ligne $i = \beta_i^\top$), on résout numériquement via une méthode proximale type ADMM.

On introduit la variable auxiliaire $Z = DB$ où $D \in \mathbb{R}^{(n-1) \times n}$ est l'opérateur de différence première

$$(DB)_i = B_{i+1, \cdot} - B_{i, \cdot}, \quad i = 1, \dots, n-1,$$

et on reformule :

$$\min_{B, Z} \frac{1}{2} \sum_{i=1}^n (y_i - x_i^\top B_{i, \cdot})^2 + \lambda \|Z\|_{1,1} \quad \text{s.c.} \quad Z = DB,$$

où $\|Z\|_{1,1} = \sum_{i,j} |Z_{ij}|$. ADMM alterne :

$$B \leftarrow \arg \min_B \frac{1}{2} \sum_{i=1}^n (y_i - x_i^\top B_{i, \cdot})^2 + \frac{\rho}{2} \|DB - Z + U\|_F^2,$$

$$Z \leftarrow \text{soft}(DB + U, \lambda/\rho),$$

$$U \leftarrow U + DB - Z,$$

avec $\rho > 0$ et soft le seuillage doux élément-par-élément.

Fonction Python (ADMM) — dépendance : numpy uniquement *Sortie. La fonction renvoie $B \in \mathbb{R}^{n \times p}$; la ligne i correspond à $\widehat{\beta}_i^T$.*

```

import numpy as np

def fused_lasso_admm(X, y, lam, rho=1.0, max_iter=2000, tol=1e-4,
                    ridge=1e-8, verbose=False):
    """
    Estimate fused lasso coefficients beta_i (i=1..n) for ordered data.

    Solves:
    min_{B in R^{n x p}} 0.5 * sum_i (y_i - x_i^T beta_i)^2
    + lam * sum_{i=2..n} ||beta_i - beta_{i-1}||_1

    Parameters
    -----
    X : (n, p) array
    y : (n,) array
    lam : float, regularization parameter (lambda)
    rho : float, ADMM penalty parameter
    max_iter : int
    tol : float, stopping tolerance (primal/dual residual)
    ridge : float, small ridge added to stabilize per-row solves
    verbose : bool

    Returns
    -----
    B : (n, p) array, rows are beta_i^T
    info : dict, diagnostics
    """
    X = np.asarray(X, dtype=float)
    y = np.asarray(y, dtype=float).reshape(-1)
    n, p = X.shape
    if y.shape[0] != n:
        raise ValueError("y must have length n (number of rows of X).")
    if lam < 0:
        raise ValueError("lam must be >= 0.")
    if rho <= 0:
        raise ValueError("rho must be > 0.")

    # Difference operator D: (n-1) x n, applied to B row-wise: DB = B[1:] - B[:-1]
    # ADMM variables
    B = np.zeros((n, p))
    Z = np.zeros((n - 1, p))
    U = np.zeros((n - 1, p))

    # Precompute per-row scalars for the data term:
    # For row i: minimize 0.5 (y_i - x_i^T b)^2 + (rho/2) ||b - v||^2
    # Closed-form: (x x^T + rho I) b = x y_i + rho v
    # Use Woodbury for speed: (rho I + x x^T)^{-1} = (1/rho)I - (1/rho^2) x x^T / (1 + (x^T
    # x)/rho)
    xnorm2 = np.sum(X * X, axis=1) # ||x_i||^2

    def soft_threshold(A, kappa):
        return np.sign(A) * np.maximum(np.abs(A) - kappa, 0.0)

    # Helper: apply Dt to W (n-1,p) -> (n,p)
    # Dt W has: first row = -W0, middle rows = W_{i-1} - W_i, last row = W_{n-2}
    def Dt(W):
        out = np.zeros((n, p))
        out[0] = -W[0]
        out[1:-1] = W[:-1] - W[1:]
        out[-1] = W[-1]
        return out

    # Because B-update couples rows through DB, we use a few iterations of
    # conjugate gradient on the normal equations:
    # (Diag(x_i x_i^T) + rho * Dt D) B = X*y (row-wise) + rho * Dt (Z - U)
    # We implement matrix-vector products without forming large matrices.

    # Build X*y contribution as (n,p) where row i is x_i * y_i
    XY = X * y[:, None]

    def apply_A(Bmat):

```

```

# A(B) = data Hessian part + rho * Dt(D(B))
# data Hessian part: for each i, x_i x_i^T B_i (rank-1 operator)
# which equals x_i * (x_i^T B_i)
proj = np.sum(X * Bmat, axis=1) # (n,)
data_part = X * proj[:, None] # (n,p)
# fused part: rho * Dt(D(B))
DB = Bmat[1:] - Bmat[:-1] # (n-1,p)
fused_part = rho * Dt(DB) # (n,p)
return data_part + fused_part + ridge * Bmat # add tiny ridge for stability

def cg_solve(B0, rhs, cg_max=200, cg_tol=1e-6):
    Bk = B0.copy()
    R = rhs - apply_A(Bk)
    P = R.copy()
    rsold = np.sum(R * R)
    if rsold < cg_tol**2:
        return Bk
    for _ in range(cg_max):
        AP = apply_A(P)
        alpha = rsold / (np.sum(P * AP) + 1e-30)
        Bk = Bk + alpha * P
        R = R - alpha * AP
        rsnew = np.sum(R * R)
        if rsnew < cg_tol**2:
            break
        P = R + (rsnew / (rsold + 1e-30)) * P
        rsold = rsnew
    return Bk

history = {"primal": [], "dual": []}

for it in range(1, max_iter + 1):
    B_prev = B.copy()
    Z_prev = Z.copy()

    # B-update: solve linear system via CG
    rhs = XY + rho * Dt(Z - U)
    B = cg_solve(B, rhs, cg_max=200, cg_tol=1e-6)

    # Z-update: soft-threshold
    DB = B[1:] - B[:-1]
    Z = soft_threshold(DB + U, lam / rho)

    # U-update
    U = U + DB - Z

    # Diagnostics (primal and dual residuals)
    r = DB - Z
    s = rho * Dt(Z - Z_prev) # mapped to (n,p)
    r_norm = np.linalg.norm(r)
    s_norm = np.linalg.norm(s)

    history["primal"].append(r_norm)
    history["dual"].append(s_norm)

    # Stopping rule (standard ADMM)
    if r_norm < tol and s_norm < tol:
        if verbose:
            print(f"Converged at iter {it}: primal={r_norm:.3e}, dual={s_norm:.3e}")
            break

if verbose and (it % 50 == 0 or it == 1):
    print(f"Iter {it:4d}: primal={r_norm:.3e}, dual={s_norm:.3e}")

info = {
    "n_iter": it,
    "primal_res": history["primal"][-1] if history["primal"] else None,
    "dual_res": history["dual"][-1] if history["dual"] else None,
    "history": history,
}
return B, info

```

Notes d'utilisation

- La sortie $B[i, :]$ correspond à $\widehat{\beta}_{i+1}^\top$ si l'on indexe Python à partir de 0.
- Les ruptures estimées sont les indices i tels que $\|\widehat{\beta}_i - \widehat{\beta}_{i-1}\|_1 > 0$ (ou $> \epsilon$ numériquement).
- Le paramètre ρ affecte la vitesse de convergence ; typiquement on teste $\rho \in \{0.5, 1, 2, 5\}$.
- Pour imposer des ruptures communes (group fused lasso), il suffit de remplacer le seuillage doux élément-par-élément par un seuillage de groupe (norme ℓ_2 par ligne) :

$$Z_i \leftarrow \left(1 - \frac{\lambda/\rho}{\|DB_i + U_i\|_2}\right)_+ (DB_i + U_i),$$

où $(\cdot)_+ = \max(\cdot, 0)$.

Le quatrième prompt suivant à été adressé à chatGPT

Pourquoi ne pas utiliser cvxpy ?

Utiliser `cvxpy` est tout à fait pertinent pour ce problème, car l'estimation fused lasso s'écrit comme un programme convexe (perte quadratique + pénalité de variation totale). L'intérêt principal est de bénéficier :

- d'une **formulation très compacte** et lisible ;
- de **solveurs robustes** (OSQP, SCS, ECOS, etc.) ;
- d'une **prototypage rapide** et d'extensions faciles (contraintes, pertes robustes, pondérations).

On reprend la formulation matricielle avec $B \in \mathbb{R}^{n \times p}$:

$$\min_{B \in \mathbb{R}^{n \times p}} \frac{1}{2} \sum_{i=1}^n (y_i - x_i^\top B_{i,\cdot})^2 + \lambda \sum_{i=2}^n \|B_{i,\cdot} - B_{i-1,\cdot}\|_1.$$

On peut l'écrire avec les opérations `sum_squares` et `norm1` de `cvxpy`.

```
import numpy as np
import cvxpy as cp

def fused_lasso_cvxpy(X, y, lam, solver="OSQP", verbose=False):
    """
    Solve fused lasso:
    min_B 0.5 * sum_i (y_i - x_i^T beta_i)^2 + lam * sum_i ||beta_i - beta_{i-1}||_1
    where beta_i are row-vectors of B (n x p).

    Parameters
    -----
    X : (n, p) array
    y : (n,) array
    lam : float >= 0
    solver : str, e.g. "OSQP", "SCS", "ECOS"
    verbose : bool

    Returns
    -----
    B_hat : (n, p) array, estimated beta_i^T in each row
    prob_value : float, optimal objective value
    """
    X = np.asarray(X, dtype=float)
    y = np.asarray(y, dtype=float).reshape(-1)
    n, p = X.shape
    if y.shape[0] != n:
        raise ValueError("y must have length n.")
    if lam < 0:
        raise ValueError("lam must be >= 0.")

    # Decision variable: B (n x p)
    B = cp.Variable((n, p))

    # Data-fit term: 0.5 * ||y - sum_j X_{ij} B_{ij} ||^2
    # In vector form: predictions = diag(X B^T) = sum over columns X*B row-wise
    yhat = cp.sum(cp.multiply(X, B), axis=1)
    loss = 0.5 * cp.sum_squares(y - yhat)

    # Fused penalty: sum_{i=2..n} ||B_i - B_{i-1}||_1
```

```

diffs = B[1:, :] - B[:-1, :]
penalty = lam * cp.sum(cp.norm1(diffs, axis=1)) # sum over rows of L1 norms

problem = cp.Problem(cp.Minimize(loss + penalty))
problem.solve(solver=solver, verbose=verbose)

if problem.status not in ("optimal", "optimal_inaccurate"):
    raise RuntimeError(f"Solver failed with status: {problem.status}")

return B.value, problem.value

```

Avantages.

- **Clarté** : la formulation mathématique est presque identique au code.
- **Flexibilité** : ajout de contraintes (bornes, monotonie, etc.) et pertes robustes (Huber) trivial.
- **Solveurs** : choix du solveur selon la structure (QP, conique, etc.).

Limites.

- **Scalabilité** : pour n très grand (par ex. 10^5) et/ou p important, `cvxpy` peut devenir coûteux (car il construit un problème générique et délègue au solveur).
- **Contrôle fin** : une implémentation dédiée (ADMM, PELT, algorithmes spécialisés TV) peut être plus rapide et plus mémoire-efficace.

Exercice 2

Revue de code

6 points

1. On a demandé à un data scientist d'écrire un programme python pour estimer les paramètres d'un modèle linéaire à l'aide de la régression du LAD L0 avec un paramètre $\lambda = 0,1$,

$$\widehat{\beta}_0 = \arg \min_{\beta \in \mathbf{R}^p} \|y - X\beta\|_1 + \lambda \|\beta\|_0$$

en utilisant CVX et une reformulation du problème comme un MILP. La réponse suivante à été produite (par Gemini). Qu'en pensez vous ?

```

import cvxpy as cp
import numpy as np

def solve_lad_l0(X, y, lambda=0.1, M=10):
    """
    Estime les paramètres d'un modèle linéaire via LAD + L0.
    Optimisation: min ||y - Xb||_1 + lambda * ||b||_0
    """
    n, p = X.shape

    # --- Variables de decision ---
    beta = cp.Variable(p)
    # Variables binaires pour la norme L0
    z = cp.Variable(p, boolean=True)
    # Variables auxiliaires pour la norme L1 (LAD)
    e = cp.Variable(n)

    # --- Objectif ---
    # somme(e_i) + lambda * somme(z_j)
    objective = cp.Minimize(cp.sum(e) + lambda * cp.sum(z))

    # --- Contraintes ---
    constraints = [ # Contraintes pour la LAD (L1)
        e >= y - X @ beta,
        e >= -(y - X @ beta),
        # Contraintes Big-M pour la norme L0
        beta <= M * z,
        beta >= -M * z ]

    # --- Resolution --- On utilise GLPK_MI ou ECOS_BB
    problem = cp.Problem(objective, constraints)
    problem.solve(solver=cp.GLPK_MI)
    return beta.value, z.value

# --- Exemple d'utilisation ---

```

```

if __name__ == "__main__":
    # Generation de donnees synthetiques
    np.random.seed(42)
    n, p = 50, 10
    X = np.random.randn(n, p)
    # On cree un vecteur beta vrai tres creux (seulement 3 coefficients non nuls)
    true_beta = np.zeros(p)
    true_beta[1], true_beta[4], true_beta[8] = 2.5, -3.0, 1.5

    # Ajout de bruit laplacien (pour favoriser la LAD)
    y = X @ true_beta + np.random.laplace(0, 0.5, size=n)

    # Parametres
    lambda_val = 0.1
    big_M = 10.0 # A ajuster selon la magnitude attendue de vos coefficients

    beta_est, support = solve_lad_l0(X, y, lambda=lambda_val, M=big_M)

```

Exercice 3

L'exam selon NotebookLM

6 points

Comme NotebookLM, répondez aux questions suivantes en 5 à 15 lignes.

1. **Optimisation** : Quelle méthode proposez vous pour commencer à résoudre un problème d'optimisation convexe et pourquoi ?
2. **Le Lasso** :
 - a) Citer cinq façons de résoudre le Lasso,

$$\widehat{\beta}_L = \arg \min_{\beta \in \mathbf{R}^p} \frac{1}{2} \|y - X\beta\|^2 + \lambda \|\beta\|_1$$

indiquez les avantages et les inconvénients de chacune de ces méthodes

- b) Dans le Lasso, quel sont les effets possibles de l'augmentation du paramètre de régularisation λ sur les coefficients β_L .

3. **Le Lasso et ses variantes** : parmi ces cinq méthodes,
 - Ridge : fonction de perte quadratique et pénalité L2

$$\widehat{\beta}_R = \arg \min_{\beta \in \mathbf{R}^p} \frac{1}{2} \|y - X\beta\|^2 + \frac{\lambda}{2} \|\beta\|^2.$$

— Lasso : fonction de perte quadratique et pénalité L1

$$\widehat{\beta}_L = \arg \min_{\beta \in \mathbf{R}^p} \frac{1}{2} \|y - X\beta\|^2 + \lambda \|\beta\|_1,$$

— Elastic Net : Combine les pénalités L2 (Ridge) et L1 (Lasso)

$$\widehat{\beta}_E = \arg \min_{\beta \in \mathbf{R}^p} \frac{1}{2} \|y - X\beta\|^2 + \lambda \|\beta_j\|_1 + \frac{\mu}{2} \|\beta\|^2,$$

— Adaptive Lasso : Utilise des poids pour pénaliser différemment chaque coefficient.

$$\widehat{\beta}_A = \arg \min_{\beta \in \mathbf{R}^p} \frac{1}{2} \|y - X\beta\|^2 + \lambda \sum_{j=1}^p w_j |\beta_j|,$$

— Square Root Lasso : une version où la fonction de perte liée aux données n'est pas au carré :

$$\widehat{\beta}_S = \arg \min_{\beta \in \mathbf{R}^p} \frac{1}{2} \|y - X\beta\| + \lambda \|\beta\|_1,$$

en vous basant sur vos connaissances et les résultats sur Kaggle, laquelle a priori donne les meilleurs résultats ?

4. **Elastic Net et Lasso** : Comment peut-on transformer un problème Elastic Net en un problème Lasso classique ?
5. **Pénalités convexe ou non ?** : Quel est l'intérêt d'utiliser des pénalités non convexes comme la MCP (Minimax Concave Penalty) par rapport au Lasso ?