Projet Optimisation et Machine Learning

Fatimetou/Limam/Abeid ——— C16698

Année 2025-2026

Objectifs du projet

- Évaluer les quatre piliers du cours :
  1. Modélisation (Chapitre 1)
  2. Méthodes de gradient déterministes (Chapitre 2)
  3. Passage à l'échelle via stochasticité (Chapitre 3)
  4. Optimisation non lisse par algorithmes proximaux (Chapitre 4)
- Compréhension théorique privilégiée
- Comparaison des algorithmes (convergence, temps CPU)

Formulation du problème

On considère une régression linéaire pour prédire l'année de sortie d'une chanson :

$$f(w) = \frac{1}{2n} \sum_{i=1}^{n} (x_i^\top w - y_i)^2 = \frac{1}{n} \sum_{i=1}^{n} f_i(w)$$

- $x_i \in \mathbb{R}^d$: vecteur des features
- $w \in \mathbb{R}^d$: vecteur des poids à apprendre
- $y_i \in \mathbb{R}$: label (année de sortie)

Détail du MSE

$$f_i(w) = \frac{1}{2}(x_i^\top w - y_i)^2$$

- Erreur prédiction pour l'exemple i : $x_i^\top w - y_i$
- Somme moyenne : $\frac{1}{n}\sum_i f_i(w) \rightarrow$ fonction à minimiser

Régularisation L2 (Ridge)

$$F(w) = f(w) + \frac{\mu}{2}\|w\|^2$$

Justification théorique :

- f convexe mais peut être plate si X n'a pas plein rang
- $\mu > 0 \implies F$ strictement convexe
- Hessienne définie positive $\rightarrow$ unicité du minimum global

Gradient étape par étape

Gradient MSE :

$$\nabla f_i(w) = \frac{\partial}{\partial w} \frac{1}{2}(x_i^\top w - y_i)^2 = x_i(x_i^\top w - y_i)$$

Somme sur n exemples :

$$\nabla f(w) = \frac{1}{n} \sum_{i=1}^{n} x_i(x_i^\top w - y_i) = \frac{1}{n} X^\top (Xw - y)$$

Avec régularisation L2 :

$$\nabla F(w) = \frac{1}{n} X^\top (Xw - y) + \mu w$$

Hessienne et propriétés

$$\nabla^2 F(w) = \frac{1}{n} X^\top X + \mu I_d$$

- Symétrique
- Définie positive si $\mu > 0$
- Convergence du gradient garantie

SVD et constante de Lipschitz

SVD : $X = U\Sigma V^\top$, $\Sigma = \mathrm{diag}(\sigma_1, \ldots, \sigma_d)$

$$\nabla^2 F(w) = \frac{1}{n} V \Sigma^2 V^\top + \mu I_d$$

Valeur propre maximale :

$$\lambda_{\max} = \frac{\sigma_{\max}^2}{n} + \mu$$

Impact : pas trop grand sinon divergence

Code Python : Batch Gradient Descent

```
import numpy as np
def mse_loss(w, X, y):
    return 0.5 * np.mean((X @ w - y)**2)

def mse_gradient(w, X, y):
    return (X.T @ (X @ w - y)) / X.shape[0]

def batch_gd(X, y, alpha, n_iter):
    w = np.zeros(X.shape[1])
    losses = []
    for _ in range(n_iter):
        w -= alpha * mse_gradient(w, X, y)
        losses.append(mse_loss(w, X, y))
    return w, losses
```

Gradient Stochastique (SGD)

$$\nabla f_i(w) = x_i(x_i^\top w - y_i)$$

$$\mathbb{E}_i[\nabla f_i(w)] = \nabla f(w)$$

Conclusion : SGD converge en moyenne vers le minimum global

Code Python : SGD

```
def sgd(X, y, alpha0, n_iter):
    n, d = X.shape
    w = np.zeros(d)
    losses = []
    for k in range(n_iter):
        i = np.random.randint(n)
        grad = X[i] * (X[i] @ w - y[i])
        w -= (alpha0/(1+k)) * grad
        losses.append(np.mean((X@w - y)**2)/2)
    return w, losses
```

Mini-batch

$$g_B(w) = \frac{1}{b} \sum_{i \in B} \nabla f_i(w)$$

$$\mathbb{E}[g_B(w)] = \nabla f(w)$$

Réduit le bruit comparé à SGD classique

Adam

$$
\begin{aligned}
m_k &= \beta_1 m_{k-1} + (1-\beta_1)\nabla f(w_{k-1}) \\
v_k &= \beta_2 v_{k-1} + (1-\beta_2)(\nabla f(w_{k-1}))^2 \\
\hat{m}_k &= m_k/(1-\beta_1^k), \quad \hat{v}_k = v_k/(1-\beta_2^k) \\
w_k &= w_{k-1} - \alpha \frac{\hat{m}_k}{\sqrt{\hat{v}_k} + \epsilon}
\end{aligned}
$$

Combine momentum + RMSProp $\rightarrow$ convergence plus rapide

Code Python : Mini-batch

```
def minibatch_sgd(X, y, alpha, batch_size,
    n_iter):
    n, d = X.shape
    w = np.zeros(d)
    for _ in range(n_iter):
        idx = np.random.choice(n, batch_size,
            replace=False)
        grad = (X[idx].T @ (X[idx] @ w - y[idx])
            ) / batch_size
        w -= alpha * grad
    return w
```

Code Python : Adam

```
def adam(X, y, alpha, beta1, beta2, eps, n_iter)
    :
    d = X.shape[1]
    w = np.zeros(d)
    m = np.zeros(d)
    v = np.zeros(d)
    for k in range(1, n_iter+1):
        grad = mse_gradient(w, X, y)
        m = beta1*m + (1-beta1)*grad
        v = beta2*v + (1-beta2)*(grad**2)
        m_hat = m/(1-beta1**k)
        v_hat = v/(1-beta2**k)
        w -= alpha * m_hat / (np.sqrt(v_hat) +
            eps)
    return w
```

Régularisation L1 vs L2

- L1 (Lasso) $\rightarrow$ favorise coefficients nuls $\rightarrow$ sparse solution
- L2 (Ridge) $\rightarrow$ coefficients petits mais non nuls

Soft-thresholding (Proximal L1)

$$\text{prox}_{\lambda\|.\|_1}(v) = \arg\min_w \frac{1}{2}\|w - v\|^2 + \lambda\|w\|_1$$

Solution composante i :

$$w_i = \text{sign}(v_i)\max(|v_i| - \lambda, 0)$$

Code Python : ISTA

```
def soft_thresholding(v, lam):
    return np.sign(v) * np.maximum(np.abs(v)-lam
        , 0)

def ista(X, y, lam, alpha, n_iter):
    w = np.zeros(X.shape[1])
    for _ in range(n_iter):
        w = soft_thresholding(w - alpha*
            logistic_gradient(w,X,y), alpha*lam)
    return w
```

Code Python : FISTA

```
def fista(X, y, lam, alpha, n_iter):
    w = np.zeros(X.shape[1])
    z = w.copy()
    t = 1
    for _ in range(n_iter):
        w_new = soft_thresholding(z - alpha*
            logistic_gradient(z,X,y), alpha*lam)
        t_new = (1 + np.sqrt(1 + 4*t**2))/2
        z = w_new + ((t-1)/t_new)*(w_new - w)
        w, t = w_new, t_new
    return w
```

Sélection de variables

- Varier $\lambda \rightarrow$ plus λ grand $\rightarrow$ plus de coefficients nuls
- Sparse solution $\rightarrow$ mots significatifs identifiés automatiquement
- Graphiques : MSE vs itérations, coefficients nuls vs λ

Conclusion

- Modélisation théorique complète (gradient, Hessienne, Lipschitz)
- Passage à l'échelle avec SGD, Mini-batch et Adam
- Optimisation non-lisse avec ISTA/FISTA
- Sélection automatique des variables pertinentes