Apprentissage Automatique

Notes de Cours / Lecture Notes

L3 / Master 1 — 2025–2026

Yaë Ulrich Gaba

Fondements théoriques et méthodes pratiques
de l'apprentissage statistique et computationnel.

25 mars 2026

Table des matières

Préface

L'apprentissage automatique (*machine learning*) est une discipline à l'intersection des mathématiques, de l'informatique et de la statistique. Son objectif fondamental est de concevoir des algorithmes capables d'**apprendre à partir de données** : au lieu de programmer explicitement une règle de décision, on laisse l'algorithme découvrir cette règle à partir d'exemples.

Positionnement du cours

Ce cours se situe au niveau L3/Master 1 et suppose des prérequis en :

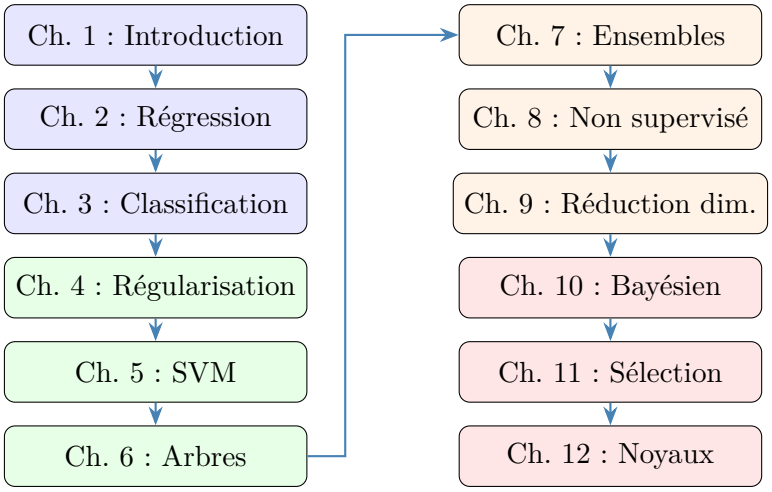
- **Algèbre linéaire** : espaces vectoriels, matrices, valeurs propres, décomposition en valeurs singulières.
- **Probabilités et statistiques** : variables aléatoires, espérance, variance, loi normale, estimation, tests.
- **Analyse** : dérivées partielles, gradient, optimisation de fonctions de plusieurs variables.
- **Programmation Python** : NumPy, Matplotlib ; la connaissance de scikit-learn est un plus mais n'est pas requise.

Philosophie pédagogique

Chaque chapitre suit une progression en trois temps :

1. **Théorie** : dérivation mathématique rigoureuse des algorithmes, avec énoncé et preuve des résultats clés.
2. **Pratique** : implémentation en Python avec scikit-learn et, quand c'est instructif, à partir de zéro.
3. **Analyse** : discussion des forces, faiblesses, hypothèses et cas d'utilisation de chaque méthode.

Plan du cours



Les chapitres 1–6 couvrent les fondements (supervisé linéaire et non linéaire). Les chapitres 7–9 abordent les méthodes d'ensemble et l'apprentissage non supervisé. Les chapitres 10–12 traitent de sujets avancés (approche bayésienne, sélection de modèles, méthodes à noyaux).

Bonne lecture et bon apprentissage !

Chapitre 1

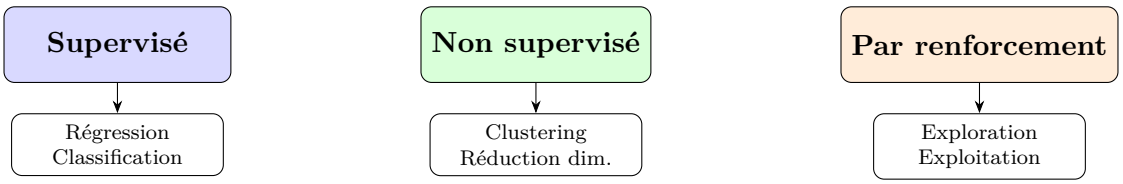
Introduction à l'apprentissage automatique

1.1 Qu'est-ce que l'apprentissage automatique ?

Définition 1.1 (Apprentissage automatique)**.** L'**apprentissage automatique** (*machine learning*) est l'étude d'algorithmes qui améliorent automatiquement leurs performances sur une tâche donnée à travers l'expérience (données), sans être explicitement programmés pour cette tâche [3].

Plus formellement, suivant Tom Mitchell : un programme apprend d'une expérience E par rapport à une classe de tâches T et une mesure de performance P, si sa performance sur T, mesurée par P, s'améliore avec l'expérience E.

1.2 Types d'apprentissage



1.2.1 Apprentissage supervisé

On dispose d'un jeu de données $\mathcal{D} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$ où $\boldsymbol{x}_i \in \mathbb{R}^d$ est un vecteur de *features* (caractéristiques) et y_i est l'étiquette (*label*). L'objectif est d'apprendre une fonction $f : \mathbb{R}^d \to \mathcal{Y}$ telle que $f(\boldsymbol{x}) \approx y$ pour de nouvelles données.

- **Régression** : $\mathcal{Y} = \mathbb{R}$ (prix d'un appartement, température).
- **Classification** : $\mathcal{Y} = \{1, \ldots, K\}$ (spam/non-spam, chiffres manuscrits).

1.2.2 Apprentissage non supervisé

On dispose uniquement de $\mathcal{D} = \{\boldsymbol{x}_i\}_{i=1}^n$ sans étiquettes. L'objectif est de découvrir une structure cachée : clusters, variétés de faible dimension, distributions.

1.2.3 Apprentissage par renforcement

Un agent interagit avec un environnement, reçoit des récompenses, et apprend une politique maximisant la récompense cumulée. Ce paradigme n'est pas couvert dans ce cours.

1.3 Formalisation mathématique

1.3.1 Espace des hypothèses

Définition 1.2 (Espace des hypothèses)**.** L'**espace des hypothèses** $\mathcal{H}$ est l'ensemble des fonctions $h : \mathbb{R}^d \to \mathcal{Y}$ parmi lesquelles l'algorithme d'apprentissage cherche la meilleure.

Exemple 1.3 (Régression linéaire)**.** $\mathcal{H} = \{h_{\boldsymbol{w}} : \boldsymbol{x} \mapsto \boldsymbol{w}^\top \boldsymbol{x} + b \mid \boldsymbol{w} \in \mathbb{R}^d, b \in \mathbb{R}\}$.

1.3.2 Fonction de perte et risque

Définition 1.4 (Fonction de perte)**.** Une **fonction de perte** $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_+$ mesure l'erreur entre la prédiction $\hat{y} = h(\boldsymbol{x})$ et la vraie valeur y. Exemples courants :

$$\text{Erreur quadratique :} \quad \ell(y, \hat{y}) = (y - \hat{y})^2 \tag{1.1}$$

$$\text{Erreur 0-1 :} \quad \ell(y, \hat{y}) = \mathbb{1}[y \neq \hat{y}] \tag{1.2}$$

$$\text{Log-loss :} \quad \ell(y, \hat{y}) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y}) \tag{1.3}$$

Définition 1.5 (Risque)**.** Le **risque** (ou erreur de généralisation) d'une hypothèse h est :

$$R(h) = \mathbb{E}_{(\boldsymbol{x},y)\sim P}[\ell(y, h(\boldsymbol{x}))] \tag{1.4}$$

où P est la distribution jointe (inconnue) des données.

Définition 1.6 (Risque empirique)**.** Le **risque empirique** est l'approximation du risque sur les données d'entraînement :

$$\hat{R}_n(h) = \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, h(\boldsymbol{x}_i)) \tag{1.5}$$

Principe de minimisation du risque empirique (ERM)

L'algorithme d'apprentissage cherche :

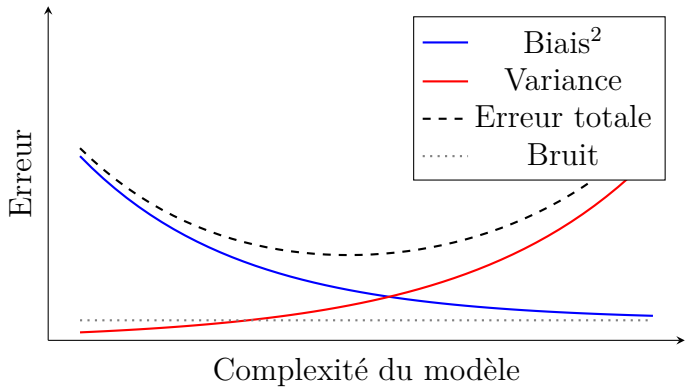
$$h^* = \arg\min_{h \in \mathcal{H}} \hat{R}_n(h) = \arg\min_{h \in \mathcal{H}} \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, h(\boldsymbol{x}_i)) \tag{1.6}$$

1.4 Le compromis biais-variance

Théorème 1.7 (Décomposition biais-variance)**.** *Pour la perte quadratique, le risque se décompose en :*

$$\mathbb{E}[(y - \hat{f}(\boldsymbol{x}))^2] = \underbrace{\text{Bias}^2[\hat{f}(\boldsymbol{x})]}_{\text{biais}^2} + \underbrace{\text{Var}[\hat{f}(\boldsymbol{x})]}_{\text{variance}} + \underbrace{\sigma_\varepsilon^2}_{\text{bruit irréductible}} \tag{1.7}$$

où $\text{Bias}[\hat{f}(\boldsymbol{x})] = \mathbb{E}[\hat{f}(\boldsymbol{x})] - f(\boldsymbol{x})$ *et* $\sigma_\varepsilon^2 = \text{Var}[\varepsilon]$.



Interprétation

- Un modèle trop simple (biais élevé) ne capture pas la structure des données : **sous-apprentissage** (*underfitting*).
- Un modèle trop complexe (variance élevée) s'adapte au bruit : **surapprentissage** (*overfitting*).
- L'objectif est de trouver le compromis optimal.

1.5 Évaluation des modèles

1.5.1 Découpage des données

Train / Validation / Test

- **Entraînement** (60–80 %) : pour ajuster les paramètres.
- **Validation** (10–20 %) : pour choisir les hyperparamètres.
- **Test** (10–20 %) : pour estimer la performance finale. **Jamais** utilisé pour le choix du modèle.

1.5.2 Validation croisée

Définition 1.8 (Validation croisée k-fold)**.** On partitionne les données en k sous-ensembles (*folds*). Pour chaque fold i :

1. Entraîner sur les $k-1$ autres folds.
2. Évaluer sur le fold i.

L'erreur de validation croisée est la moyenne des k erreurs :

$$\mathrm{CV}(k) = \frac{1}{k} \sum_{i=1}^{k} \hat{R}^{(i)} \tag{1.8}$$

1.5.3 Métriques

Métriques courantes

Régression :

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2, \quad \text{RMSE} = \sqrt{\text{MSE}}, \quad R^2 = 1 - \frac{\sum(y_i - \hat{y}_i)^2}{\sum(y_i - \bar{y})^2} \tag{1.9}$$

Classification :

$$\text{Précision} = \frac{VP}{VP + FP}, \quad \text{Rappel} = \frac{VP}{VP + FN}, \quad F_1 = 2 \cdot \frac{\text{Préc.} \times \text{Rappel}}{\text{Préc.} + \text{Rappel}} \tag{1.10}$$

1.6 Implémentation : premier modèle avec scikit-learn

Classification Iris avec scikit-learn

```python
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report

# Charger les donnees
X, y = load_iris(return_X_y=True)

# Decoupage train/test
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

# Modele k-NN
model = KNeighborsClassifier(n_neighbors=5)
model.fit(X_train, y_train)

# Evaluation
print(f"Accuracy test: {model.score(X_test, y_test):.4f}")
print(classification_report(y_test, model.predict(X_test)))

# Validation croisee 5-fold
scores = cross_val_score(model, X, y, cv=5, scoring="accuracy")
print(f"CV accuracy: {scores.mean():.4f} +/- {scores.std():.4f}")
```

Sortie

```
Accuracy test: 1.0000
              precision    recall  f1-score   support
           0       1.00      1.00      1.00        10
```

```
           1       1.00      1.00      1.00        10
           2       1.00      1.00      1.00        10
    accuracy                           1.00        30
CV accuracy: 0.9667 +/- 0.0211
```

1.7 Exercices

Exercice 1.1 (Risque empirique)**.** Soit $\mathcal{D} = \{(1, 2), (2, 3.5), (3, 6)\}$ et $h_w(x) = wx$.

1. Calculer $\hat{R}_3(h_w)$ avec la perte quadratique pour $w = 1.5$ et $w = 2$.
2. Trouver w^* minimisant $\hat{R}_3(h_w)$ analytiquement.
3. Tracer $\hat{R}_3(h_w)$ en fonction de w.

Exercice 1.2 (Décomposition biais-variance)**.**

1. Montrer que $\mathbb{E}[(y - \hat{f})^2] = \text{Bias}^2 + \text{Var} + \sigma^2$ en détaillant chaque étape.
2. Générer $n = 50$ points avec $y = \sin(x) + \varepsilon$, $\varepsilon \sim \mathcal{N}(0, 0.3)$. Ajuster des polynômes de degrés 1, 3, 5, 10, 15 et illustrer le compromis biais-variance.

Exercice 1.3 (Validation croisée)**.**

1. Implémenter la validation croisée k-fold à partir de zéro (sans `sklearn.model_selection`).
2. Comparer les résultats de votre implémentation avec `cross_val_score` pour k-NN sur le dataset Iris.
3. Étudier l'effet de k (nombre de voisins) sur l'accuracy CV. Tracer la courbe accuracy vs k pour $k \in \{1, 3, 5, 7, \ldots, 25\}$.

Chapitre 2

Régression Linéaire et Polynomiale

Idée centrale

La régression linéaire est le modèle fondamental de l'apprentissage supervisé : on cherche la « meilleure » relation affine entre des variables explicatives $\boldsymbol{x}$ et une variable cible $y \in \mathbb{R}$, au sens des moindres carrés.

2.1 Régression linéaire simple

2.1.1 Modèle

On dispose de n observations $(x_i, y_i)_{i=1}^n$ avec $x_i \in \mathbb{R}$ et $y_i \in \mathbb{R}$. On suppose le modèle :

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i, \qquad \varepsilon_i \sim \mathcal{N}(0, \sigma^2), \quad \varepsilon_i \text{ i.i.d.}$$

Définition 2.1 (Fonction de coût des moindres carrés)**.** La fonction de coût *Ordinary Least Squares* (OLS) est :

$$\mathcal{L}(\beta_0, \beta_1) = \sum_{i=1}^{n} (y_i - \beta_0 - \beta_1 x_i)^2$$

L'objectif est de trouver $(\hat{\beta}_0, \hat{\beta}_1) = \arg\min_{\beta_0, \beta_1} \mathcal{L}(\beta_0, \beta_1)$.

2.1.2 Dérivation des estimateurs OLS

Théorème 2.2 (Estimateurs des moindres carrés)**.** *Les estimateurs OLS de la régression linéaire simple sont :*

$$\hat{\beta}_1 = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^n (x_i - \bar{x})^2} = \frac{\mathrm{Cov}(X, Y)}{\mathrm{Var}(X)}, \qquad \hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x}$$

où $\bar{x} = \frac{1}{n}\sum_i x_i$ *et* $\bar{y} = \frac{1}{n}\sum_i y_i$.

Démonstration. On annule les dérivées partielles de $\mathcal{L}$:

$$\frac{\partial \mathcal{L}}{\partial \beta_0} = -2 \sum_{i=1}^{n} (y_i - \beta_0 - \beta_1 x_i) = 0 \implies \hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x}$$

$$\frac{\partial \mathcal{L}}{\partial \beta_1} = -2\sum_{i=1}^{n} x_i(y_i - \beta_0 - \beta_1 x_i) = 0$$

En substituant $\hat{\beta}_0$ dans la seconde équation :

$$\sum_{i=1}^{n} x_i\big(y_i - \bar{y} + \hat{\beta}_1(\bar{x} - x_i)\big) = 0 \implies \hat{\beta}_1 = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sum_i (x_i - \bar{x})^2}$$

□

Coefficient de détermination

$$R^2 = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2} = 1 - \frac{\text{RSS}}{\text{TSS}} \in [0, 1]$$

R^2 mesure la proportion de variance expliquée par le modèle.

2.2 Régression linéaire multiple

2.2.1 Formulation matricielle

On généralise à p variables explicatives. Pour n observations :

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon}$$

où $\boldsymbol{X} \in \mathbb{R}^{n \times (p+1)}$ est la matrice de design (première colonne de 1), $\boldsymbol{\beta} \in \mathbb{R}^{p+1}$ et $\boldsymbol{\varepsilon} \sim \mathcal{N}(\boldsymbol{0}, \sigma^2 \boldsymbol{I}_n)$.

Définition 2.3 (Fonction de coût matricielle)**.**

$$\mathcal{L}(\boldsymbol{\beta}) = \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}\|^2 = (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})^\top (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})$$

2.2.2 Équations normales

Théorème 2.4 (Équations normales)**.** *Si $\boldsymbol{X}^\top \boldsymbol{X}$ est inversible, l'unique minimiseur de $\mathcal{L}$ est :*

$$\hat{\boldsymbol{\beta}} = (\boldsymbol{X}^\top \boldsymbol{X})^{-1} \boldsymbol{X}^\top \boldsymbol{y}$$

Démonstration. On développe et on dérive :

$$\mathcal{L}(\boldsymbol{\beta}) = \boldsymbol{y}^\top \boldsymbol{y} - 2\boldsymbol{\beta}^\top \boldsymbol{X}^\top \boldsymbol{y} + \boldsymbol{\beta}^\top \boldsymbol{X}^\top \boldsymbol{X} \boldsymbol{\beta}$$
$$\nabla_{\boldsymbol{\beta}} \mathcal{L} = -2\boldsymbol{X}^\top \boldsymbol{y} + 2\boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{\beta} = \boldsymbol{0}$$

D'où $\boldsymbol{X}^\top \boldsymbol{X}\hat{\boldsymbol{\beta}} = \boldsymbol{X}^\top \boldsymbol{y}$. La matrice hessienne $2\boldsymbol{X}^\top \boldsymbol{X}$ est semi-définie positive, donc c'est bien un minimum. □

2.2.3 Interprétation géométrique

Projection orthogonale

Le vecteur $\hat{\boldsymbol{y}} = \boldsymbol{X}\hat{\boldsymbol{\beta}}$ est la **projection orthogonale** de $\boldsymbol{y}$ sur l'espace colonne de $\boldsymbol{X}$, noté $\mathrm{Col}(\boldsymbol{X})$. La matrice de projection est $\boldsymbol{H} = \boldsymbol{X}(\boldsymbol{X}^\top\boldsymbol{X})^{-1}\boldsymbol{X}^\top$ (« hat matrix »). Le vecteur des résidus $\hat{\boldsymbol{\varepsilon}} = \boldsymbol{y} - \hat{\boldsymbol{y}}$ est orthogonal à $\mathrm{Col}(\boldsymbol{X})$.

Proposition 2.5 (Propriétés de la hat matrix)**.** La matrice $\boldsymbol{H} = \boldsymbol{X}(\boldsymbol{X}^\top\boldsymbol{X})^{-1}\boldsymbol{X}^\top$ vérifie :

1. $\boldsymbol{H}^2 = \boldsymbol{H}$ (idempotente),
2. $\boldsymbol{H}^\top = \boldsymbol{H}$ (symétrique),
3. $\mathrm{tr}(\boldsymbol{H}) = p + 1$.

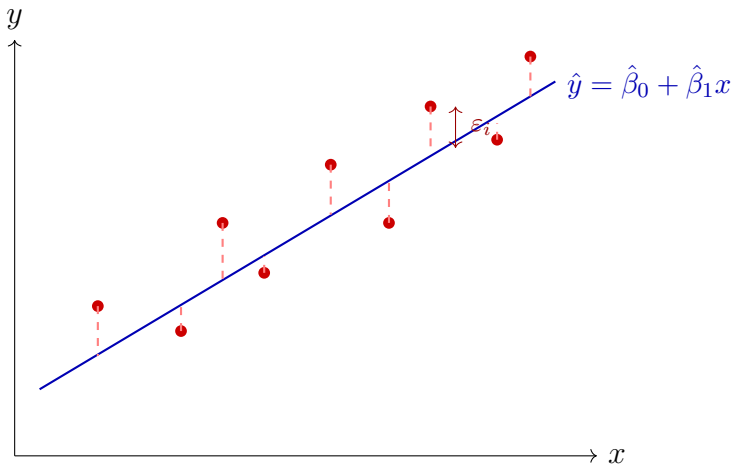


FIG. 2.1 : Droite de régression et résidus (segments rouges).

2.3 Régression polynomiale

Définition 2.6 (Modèle polynomial de degré d)**.** On pose $\phi(x) = (1, x, x^2, \ldots, x^d)^\top \in \mathbb{R}^{d+1}$ et :

$$y = \boldsymbol{\phi}(x)^\top\boldsymbol{\beta} + \varepsilon = \beta_0 + \beta_1 x + \beta_2 x^2 + \cdots + \beta_d x^d + \varepsilon$$

C'est un modèle **linéaire en les paramètres $\boldsymbol{\beta}$**, même s'il est non-linéaire en x.

Sur-apprentissage polynomial

Un polynôme de degré $d = n - 1$ passe exactement par les n points ($R^2 = 1$ sur l'ensemble d'entraînement), mais généralise très mal. C'est un cas classique de **sur-apprentissage** (overfitting). Le choix de d est un problème de sélection de modèle (cf. chapitre 4).

Remarque 2.7. En pratique, on construit la matrice de design de Vandermonde :

$$\boldsymbol{\Phi} = \begin{pmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^d \\ 1 & x_2 & x_2^2 & \cdots & x_2^d \\ \vdots & & & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^d \end{pmatrix} \in \mathbb{R}^{n\times(d+1)}$$

et on applique les équations normales : $\hat{\boldsymbol{\beta}} = (\boldsymbol{\Phi}^\top \boldsymbol{\Phi})^{-1} \boldsymbol{\Phi}^\top \boldsymbol{y}$.

2.4 Descente de gradient pour la régression linéaire

Descente de gradient (batch)

Entrée : données $(\boldsymbol{X}, \boldsymbol{y})$, taux d'apprentissage $\eta > 0$, nombre d'itérations T.
Initialisation : $\boldsymbol{\beta}^{(0)} = \mathbf{0}$.
Pour $t = 0, 1, \ldots, T-1$:

$$\boldsymbol{\beta}^{(t+1)} = \boldsymbol{\beta}^{(t)} - \eta \nabla_{\boldsymbol{\beta}} \mathcal{L}(\boldsymbol{\beta}^{(t)}) = \boldsymbol{\beta}^{(t)} + \frac{2\eta}{n} \boldsymbol{X}^\top (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}^{(t)})$$

Sortie : $\hat{\boldsymbol{\beta}} = \boldsymbol{\beta}^{(T)}$.

Proposition 2.8 (Convergence)**.** Si $0 < \eta < \frac{2}{\lambda_{\max}(\boldsymbol{X}^\top \boldsymbol{X}/n)}$, la descente de gradient converge vers la solution des équations normales $\hat{\boldsymbol{\beta}}_{\text{OLS}}$.

Normalisation des variables

Avant d'appliquer la descente de gradient, il est essentiel de **centrer-réduire** les variables ($z_j = \frac{x_j - \mu_j}{\sigma_j}$) pour que toutes les composantes du gradient aient des échelles comparables. Cela accélère la convergence.

2.5 Implémentation Python

2.5.1 Implémentation depuis zéro

Régression linéaire – implémentation manuelle

```python
import numpy as np

class LinearRegressionScratch:
    """Regression lineaire par equations normales."""

    def fit(self, X, y):
        # Ajout de la colonne de biais
        n = X.shape[0]
        X_b = np.c_[np.ones((n, 1)), X]
        # Equations normales
        self.beta_ = np.linalg.solve(
            X_b.T @ X_b, X_b.T @ y
        )
        return self

    def predict(self, X):
        n = X.shape[0]
        X_b = np.c_[np.ones((n, 1)), X]
        return X_b @ self.beta_
```

```
    def r_squared(self, X, y):
        y_pred = self.predict(X)
        ss_res = np.sum((y - y_pred) ** 2)
        ss_tot = np.sum((y - np.mean(y)) ** 2)
        return 1 - ss_res / ss_tot
```

2.5.2 Descente de gradient depuis zéro

Descente de gradient pour la régression linéaire

```
class LinearRegressionGD:
    """Regression lineaire par descente de gradient."""

    def __init__(self, lr=0.01, n_iter=1000):
        self.lr = lr
        self.n_iter = n_iter

    def fit(self, X, y):
        n, p = X.shape
        X_b = np.c_[np.ones((n, 1)), X]
        self.beta_ = np.zeros(p + 1)
        self.losses_ = []

        for _ in range(self.n_iter):
            residuals = y - X_b @ self.beta_
            gradient = -(2 / n) * X_b.T @ residuals
            self.beta_ -= self.lr * gradient
            self.losses_.append(np.mean(residuals ** 2))

        return self

    def predict(self, X):
        X_b = np.c_[np.ones((X.shape[0], 1)), X]
        return X_b @ self.beta_
```

2.5.3 Avec scikit-learn

Régression linéaire et polynomiale avec scikit-learn

```
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import cross_val_score
from sklearn.metrics import mean_squared_error, r2_score
import numpy as np

# --- Donnees synthetiques ---
```

```python
np.random.seed(42)
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X[:, 0] + np.random.randn(100)

# --- Regression lineaire simple ---
model = LinearRegression()
model.fit(X, y)
print(f"beta_0 = {model.intercept_:.3f}")
print(f"beta_1 = {model.coef_[0]:.3f}")
print(f"R^2    = {model.score(X, y):.4f}")

# --- Regression polynomiale (degre 3) ---
poly_model = make_pipeline(
    PolynomialFeatures(degree=3, include_bias=False),
    LinearRegression()
)
poly_model.fit(X, y)

# --- Validation croisee ---
scores = cross_val_score(
    poly_model, X, y, cv=5,
    scoring='neg_mean_squared_error'
)
print(f"MSE (CV 5-folds) = {-scores.mean():.4f}")
```

Sortie

```
beta_0 = 4.114
beta_1 = 2.840
R^2    = 0.8765
MSE (CV 5-folds) = 1.0432
```

2.6 Propriétés statistiques des estimateurs OLS

Théorème 2.9 (Gauss–Markov). *Sous les hypothèses du modèle linéaire ($\mathbb{E}[\boldsymbol{\varepsilon}] = \mathbf{0}$, $\mathrm{Cov}(\boldsymbol{\varepsilon}) = \sigma^2 \boldsymbol{I}_n$), l'estimateur OLS $\hat{\boldsymbol{\beta}}$ est le* ***meilleur estimateur linéaire sans biais*** *(BLUE) : pour tout estimateur linéaire sans biais $\tilde{\boldsymbol{\beta}}$,*

$$\mathrm{Var}(\boldsymbol{a}^\top \hat{\boldsymbol{\beta}}) \leq \mathrm{Var}(\boldsymbol{a}^\top \tilde{\boldsymbol{\beta}}) \quad \forall\, \boldsymbol{a} \in \mathbb{R}^{p+1}$$

Distribution des estimateurs

Sous les hypothèses gaussiennes :

$$\hat{\boldsymbol{\beta}} \sim \mathcal{N}\big(\boldsymbol{\beta},\ \sigma^2 (\boldsymbol{X}^\top \boldsymbol{X})^{-1}\big)$$

L'estimateur sans biais de σ^2 est :

$$\hat{\sigma}^2 = \frac{1}{n-p-1} \left\| \boldsymbol{y} - \boldsymbol{X}\hat{\boldsymbol{\beta}} \right\|^2 = \frac{\mathrm{RSS}}{n-p-1}$$

Remarque 2.10. La décomposition biais-variance pour la régression linéaire donne, pour une nouvelle observation $\boldsymbol{x}_0$:

$$\mathbb{E}\big[(\hat{y}_0 - y_0)^2\big] = \underbrace{\mathrm{Bias}(\hat{y}_0)^2}_{=0 \text{ si modèle correct}} + \mathrm{Var}(\hat{y}_0) + \sigma^2$$

2.7 Exercices

Exercice 2.1 (Niveau 1 – Calcul direct)**.** On observe les données suivantes :

x_i	1	2	3	4	5
y_i	2.1	3.9	6.2	7.8	10.1

1. Calculer $\bar{x}$, $\bar{y}$, $\sum(x_i - \bar{x})(y_i - \bar{y})$ et $\sum(x_i - \bar{x})^2$.
2. En déduire $\hat{\beta}_1$ et $\hat{\beta}_0$.
3. Calculer R^2 et interpréter.
4. Prédire y pour $x = 6$.

Exercice 2.2 (Niveau 2 – Régression multiple et diagnostic)**.** On considère le jeu de données `Boston Housing` (ou équivalent) avec $p = 5$ variables sélectionnées.

1. Écrire la matrice de design $\boldsymbol{X}$ et calculer $\hat{\boldsymbol{\beta}}$ avec les équations normales en Python (sans scikit-learn).
2. Comparer avec `LinearRegression` de scikit-learn.
3. Tracer les résidus $\hat{\varepsilon}_i$ en fonction de $\hat{y}_i$. Que vérifie-t-on avec ce graphique ?
4. Réaliser un Q-Q plot des résidus. Commenter la normalité.
5. Calculer le VIF (*Variance Inflation Factor*) de chaque variable. Identifier d'éventuelles colinéarités.

Exercice 2.3 (Niveau 3 – Analyse théorique et descente de gradient)**.**

1. Démontrer le théorème de Gauss–Markov.
2. Montrer que pour la régression polynomiale de degré d sur n points, le nombre de paramètres effectifs est $d+1$ et que la matrice de Vandermonde est mal conditionnée pour d grand. En déduire l'intérêt de la régularisation (lien avec le chapitre 4).
3. Implémenter la descente de gradient stochastique (SGD) et la descente de gradient par mini-batchs. Comparer les courbes de convergence avec la descente de gradient batch.
4. Montrer que la condition $\eta < 2/\lambda_{\max}$ est nécessaire à la convergence. *Indication :* analyser le système dans la base propre de $\boldsymbol{X}^\top \boldsymbol{X}$.

Chapitre 3

Classification — Régression Logistique, k-NN, Naïve Bayes

Du continu au discret

Contrairement à la régression où $y \in \mathbb{R}$, la classification prédit une **étiquette discrète** $y \in \{0, 1, \ldots, K-1\}$. Ce chapitre présente trois approches fondamentales : la régression logistique (modèle discriminant paramétrique), les k plus proches voisins (méthode non-paramétrique) et Naïve Bayes (modèle génératif).

3.1 Régression logistique

3.1.1 Cas binaire : la fonction sigmoïde

Définition 3.1 (Fonction sigmoïde)**.** La fonction sigmoïde (ou logistique) est définie par :

$$\sigma(z) = \frac{1}{1 + e^{-z}}, \qquad z \in \mathbb{R}$$

Elle vérifie : $\sigma(-z) = 1 - \sigma(z)$ et $\sigma'(z) = \sigma(z)(1 - \sigma(z))$.

Le modèle de régression logistique binaire pose :

$$P(Y = 1 \mid \boldsymbol{x}) = \sigma(\boldsymbol{w}^\top \boldsymbol{x} + b) = \frac{1}{1 + \exp(-\boldsymbol{w}^\top \boldsymbol{x} - b)}$$

où $\boldsymbol{w} \in \mathbb{R}^p$ est le vecteur de poids et $b \in \mathbb{R}$ le biais.

Log-odds (logit)

$$\log \frac{P(Y = 1 \mid \boldsymbol{x})}{P(Y = 0 \mid \boldsymbol{x})} = \boldsymbol{w}^\top \boldsymbol{x} + b$$

La frontière de décision est l'hyperplan $\{\boldsymbol{x} : \boldsymbol{w}^\top \boldsymbol{x} + b = 0\}$.

3.1.2 Estimation par maximum de vraisemblance

Théorème 3.2 (Vraisemblance de la régression logistique)**.** *Étant donné n observations $(\boldsymbol{x}_i, y_i)$ avec $y_i \in \{0, 1\}$, la log-vraisemblance est :*

$$\ell(\boldsymbol{w}, b) = \sum_{i=1}^{n} \Big[y_i \log \sigma(\boldsymbol{w}^\top \boldsymbol{x}_i + b) + (1 - y_i) \log\big(1 - \sigma(\boldsymbol{w}^\top \boldsymbol{x}_i + b)\big) \Big]$$

De manière équivalente, on minimise la ***perte logistique*** *(binary cross-entropy) :*

$$\mathcal{L}(\boldsymbol{w}, b) = -\frac{1}{n} \ell(\boldsymbol{w}, b) = -\frac{1}{n} \sum_{i=1}^{n} \big[y_i \log \hat{p}_i + (1 - y_i) \log(1 - \hat{p}_i) \big]$$

où $\hat{p}_i = \sigma(\boldsymbol{w}^\top \boldsymbol{x}_i + b)$.

Démonstration. Chaque y_i suit une loi de Bernoulli : $P(y_i \mid \boldsymbol{x}_i) = \hat{p}_i^{y_i}(1 - \hat{p}_i)^{1-y_i}$. Par indépendance, $\mathcal{V} = \prod_i \hat{p}_i^{y_i}(1 - \hat{p}_i)^{1-y_i}$. Le logarithme donne le résultat. □

3.1.3 Gradient et optimisation

Proposition 3.3 (Gradient de la perte logistique)**.**

$$\nabla_{\boldsymbol{w}} \mathcal{L} = -\frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{p}_i) \boldsymbol{x}_i, \qquad \frac{\partial \mathcal{L}}{\partial b} = -\frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{p}_i)$$

Démonstration. On utilise $\sigma'(z) = \sigma(z)(1 - \sigma(z))$. En posant $z_i = \boldsymbol{w}^\top \boldsymbol{x}_i + b$:

$$\frac{\partial}{\partial \boldsymbol{w}} \big[-y_i \log \sigma(z_i) - (1 - y_i) \log(1 - \sigma(z_i)) \big] = -\big(y_i - \sigma(z_i)\big) \boldsymbol{x}_i$$

□

Pas de solution analytique

Contrairement à la régression linéaire, il n'existe pas de solution en forme fermée. On utilise des méthodes itératives : descente de gradient, Newton-Raphson (IRLS), L-BFGS, etc. La fonction de coût est **convexe**, donc tout minimum local est global.

3.1.4 Extension multiclasse : Softmax

Définition 3.4 (Régression softmax)**.** Pour K classes, on définit le vecteur de scores $\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$ avec $\boldsymbol{W} \in \mathbb{R}^{K \times p}$, $\boldsymbol{b} \in \mathbb{R}^K$. La probabilité de la classe k est :

$$P(Y = k \mid \boldsymbol{x}) = \mathrm{softmax}(\boldsymbol{z})_k = \frac{\exp(z_k)}{\sum_{j=1}^{K} \exp(z_j)}$$

La perte associée est la **cross-entropy catégorielle** :

$$\mathcal{L} = -\frac{1}{n} \sum_{i=1}^{n} \sum_{k=1}^{K} \mathbb{1}_{y_i = k} \log P(Y = k \mid \boldsymbol{x}_i)$$

3.2 k plus proches voisins (k-NN)

3.2.1 Algorithme

Classification par k-NN

Entrée : ensemble d'entraînement $\mathcal{D} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$, point de test $\boldsymbol{x}_0$, entier $k \geq 1$.

1. Calculer $d(\boldsymbol{x}_0, \boldsymbol{x}_i)$ pour tout $i \in \{1, \ldots, n\}$.
2. Sélectionner les k indices $\mathcal{N}_k(\boldsymbol{x}_0)$ correspondant aux k plus petites distances.
3. Prédire :

$$\hat{y}_0 = \arg\max_{c \in \{1,\ldots,K\}} \sum_{i \in \mathcal{N}_k(\boldsymbol{x}_0)} \mathbb{1}_{y_i = c}$$

3.2.2 Choix de la distance

Métriques courantes

- **Euclidienne** (ℓ_2) : $d(\boldsymbol{x}, \boldsymbol{x}') = \|\boldsymbol{x} - \boldsymbol{x}'\|_2 = \sqrt{\sum_j (x_j - x'_j)^2}$
- **Manhattan** (ℓ_1) : $d(\boldsymbol{x}, \boldsymbol{x}') = \|\boldsymbol{x} - \boldsymbol{x}'\|_1 = \sum_j |x_j - x'_j|$
- **Minkowski** (ℓ_p) : $d(\boldsymbol{x}, \boldsymbol{x}') = \|\boldsymbol{x} - \boldsymbol{x}'\|_p = \left(\sum_j |x_j - x'_j|^p\right)^{1/p}$

3.2.3 Choix de k

Compromis biais-variance dans k-NN

- $k = 1$: biais nul, variance maximale (frontière très irrégulière).
- $k = n$: variance nulle, biais maximal (on prédit toujours la classe majoritaire).
- On choisit k par **validation croisée**. En théorie, $k^* \sim n^{2/(2+p)}$ (règle de Stone).

Remarque 3.5. Le k-NN souffre du **fléau de la dimension** : en haute dimension, les distances deviennent quasi uniformes et l'algorithme perd son pouvoir discriminant. En pratique, on réduit la dimension avant d'appliquer k-NN (PCA, sélection de variables).

3.3 Naïve Bayes

3.3.1 Théorème de Bayes pour la classification

Théorème 3.6 (Classifieur bayésien optimal)**.** *Le classifieur minimisant le risque* 0-1 *est :*

$$h^*(\boldsymbol{x}) = \arg\max_k P(Y = k \mid \boldsymbol{x}) = \arg\max_k \underbrace{P(\boldsymbol{x} \mid Y = k)}_{\text{vraisemblance}} \cdot \underbrace{P(Y = k)}_{\text{prior}}$$

car $P(\boldsymbol{x})$ est commun à toutes les classes.

3.3.2 Hypothèse d'indépendance conditionnelle

Définition 3.7 (Naïve Bayes)**.** L'hypothèse « naïve » suppose l'indépendance conditionnelle des variables sachant la classe :

$$P(\boldsymbol{x} \mid Y = k) = \prod_{j=1}^{p} P(x_j \mid Y = k)$$

Le classifieur Naïve Bayes est alors :

$$\hat{y} = \arg\max_{k} \left[\log P(Y = k) + \sum_{j=1}^{p} \log P(x_j \mid Y = k) \right]$$

3.3.3 Naïve Bayes gaussien

Proposition 3.8 (Gaussian Naïve Bayes)**.** Si $x_j \mid Y = k \sim \mathcal{N}(\mu_{jk}, \sigma_{jk}^2)$, alors :

$$\log P(x_j \mid Y = k) = -\frac{1}{2} \log(2\pi\sigma_{jk}^2) - \frac{(x_j - \mu_{jk})^2}{2\sigma_{jk}^2}$$

Les paramètres sont estimés par maximum de vraisemblance :

$$\hat{\mu}_{jk} = \frac{1}{n_k} \sum_{i : y_i = k} x_{ij}, \qquad \hat{\sigma}_{jk}^2 = \frac{1}{n_k} \sum_{i : y_i = k} (x_{ij} - \hat{\mu}_{jk})^2$$

où $n_k = |\{i : y_i = k\}|$.

Lissage de Laplace

Pour le Naïve Bayes multinomial (texte), on ajoute un pseudo-comptage $\alpha > 0$ pour éviter les probabilités nulles :

$$\hat{P}(x_j = v \mid Y = k) = \frac{n_{jvk} + \alpha}{n_k + \alpha |\mathcal{V}_j|}$$

où $|\mathcal{V}_j|$ est le nombre de valeurs possibles de x_j.

3.4 Évaluation des classifieurs

3.4.1 Matrice de confusion

Définition 3.9 (Matrice de confusion – cas binaire)**.**

	$\hat{y} = 1$	$\hat{y} = 0$
$y = 1$	VP	FN
$y = 0$	FP	VN

où VP = vrais positifs, FP = faux positifs, VN = vrais négatifs, FN = faux négatifs.

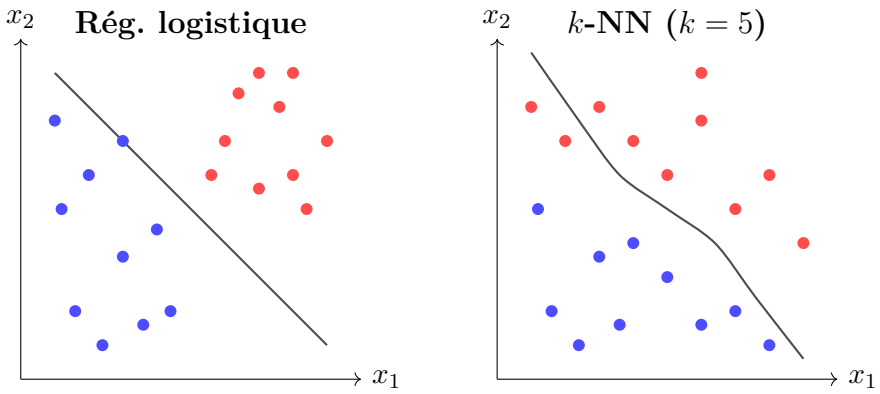


FIG. 3.1 : Frontières de décision : régression logistique (linéaire) vs. k-NN (non-linéaire).

Métriques de classification

$$\text{Précision} = \frac{\text{VP}}{\text{VP} + \text{FP}}, \qquad \text{Rappel (Sensibilité)} = \frac{\text{VP}}{\text{VP} + \text{FN}}$$

$$\text{Spécificité} = \frac{\text{VN}}{\text{VN} + \text{FP}}, \qquad F_1 = \frac{2 \cdot \text{Précision} \cdot \text{Rappel}}{\text{Précision} + \text{Rappel}}$$

3.4.2 Courbe ROC et AUC

Définition 3.10 (Courbe ROC)**.** La courbe ROC (*Receiver Operating Characteristic*) trace le taux de vrais positifs (rappel) en fonction du taux de faux positifs ($1 -$ spécificité) pour différents seuils de décision $\tau \in [0, 1]$.

L'**AUC** (*Area Under the Curve*) mesure la qualité globale du classifieur :

$$\text{AUC} = P(\hat{p}_{Y=1} > \hat{p}_{Y=0}) = \int_0^1 \text{TPR}(\tau)\, d\text{FPR}(\tau)$$

Un classifieur aléatoire a AUC $= 0.5$, un classifieur parfait a AUC $= 1$.

3.5 Implémentation Python

3.5.1 Régression logistique

Régression logistique depuis zéro

```python
import numpy as np

class LogisticRegressionScratch:
    def __init__(self, lr=0.1, n_iter=1000):
        self.lr = lr
        self.n_iter = n_iter

    def _sigmoid(self, z):
        return 1 / (1 + np.exp(-np.clip(z, -500, 500)))
```

```
    def fit(self, X, y):
        n, p = X.shape
        self.w_ = np.zeros(p)
        self.b_ = 0.0
        self.losses_ = []

        for _ in range(self.n_iter):
            z = X @ self.w_ + self.b_
            p_hat = self._sigmoid(z)
            # Gradient
            dw = -(1/n) * X.T @ (y - p_hat)
            db = -(1/n) * np.sum(y - p_hat)
            self.w_ -= self.lr * dw
            self.b_ -= self.lr * db
            # Log-loss
            loss = -np.mean(
                y * np.log(p_hat + 1e-15)
                + (1 - y) * np.log(1 - p_hat + 1e-15)
            )
            self.losses_.append(loss)
        return self

    def predict_proba(self, X):
        return self._sigmoid(X @ self.w_ + self.b_)

    def predict(self, X, threshold=0.5):
        return (self.predict_proba(X) >= threshold).astype(int)
```

3.5.2 Les trois classifieurs avec scikit-learn

Logistique, k-NN et Naïve Bayes – scikit-learn

```
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import (
    train_test_split, cross_val_score
)
from sklearn.metrics import (
    classification_report, confusion_matrix,
    roc_curve, roc_auc_score
)
from sklearn.datasets import make_classification
from sklearn.preprocessing import StandardScaler
import numpy as np
import matplotlib.pyplot as plt

# --- Donnees synthetiques ---
```

```python
X, y = make_classification(
    n_samples=500, n_features=2,
    n_informative=2, n_redundant=0,
    random_state=42
)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)
scaler = StandardScaler()
X_train_s = scaler.fit_transform(X_train)
X_test_s = scaler.transform(X_test)

# --- Modeles ---
models = {
    "Logistique": LogisticRegression(),
    "k-NN (k=5)": KNeighborsClassifier(n_neighbors=5),
    "Naive Bayes": GaussianNB()
}

for name, model in models.items():
    model.fit(X_train_s, y_train)
    y_pred = model.predict(X_test_s)
    acc = np.mean(y_pred == y_test)
    cv_scores = cross_val_score(
        model, X_train_s, y_train, cv=5
    )
    print(f"--- {name} ---")
    print(f"Accuracy test : {acc:.3f}")
    print(f"CV accuracy   : {cv_scores.mean():.3f} "
          f"(+/- {cv_scores.std():.3f})")
    print(confusion_matrix(y_test, y_pred))
    print()
```

Sortie

```
--- Logistique ---
Accuracy test : 0.893
CV accuracy   : 0.886 (+/- 0.025)
[[68  7]
 [ 9 66]]

--- k-NN (k=5) ---
Accuracy test : 0.880
CV accuracy   : 0.871 (+/- 0.031)
[[65 10]
 [ 8 67]]

--- Naive Bayes ---
Accuracy test : 0.887
CV accuracy   : 0.877 (+/- 0.028)
```

```
[[67  8]
 [ 9 66]]
```

Courbe ROC comparative

```python
plt.figure(figsize=(7, 5))
for name, model in models.items():
    if hasattr(model, "predict_proba"):
        y_score = model.predict_proba(X_test_s)[:, 1]
    else:
        y_score = model.decision_function(X_test_s)
    fpr, tpr, _ = roc_curve(y_test, y_score)
    auc = roc_auc_score(y_test, y_score)
    plt.plot(fpr, tpr, label=f"{name} (AUC={auc:.3f})")

plt.plot([0, 1], [0, 1], 'k--', label="Aleatoire")
plt.xlabel("Taux de faux positifs")
plt.ylabel("Taux de vrais positifs")
plt.title("Courbes ROC")
plt.legend()
plt.tight_layout()
plt.savefig("roc_comparison.pdf")
plt.show()
```

3.6 Exercices

Exercice 3.1 (Niveau 1 – Régression logistique à la main)**.** On considère un problème binaire avec une seule variable x et les paramètres $w = 2$, $b = -3$.

1. Calculer $P(Y = 1 \mid x)$ pour $x \in \{0, 1, 1.5, 2, 3\}$.
2. Déterminer la frontière de décision (pour $\tau = 0.5$).
3. On observe $(x_1, y_1) = (1, 0)$ et $(x_2, y_2) = (2, 1)$. Calculer la log-vraisemblance $\ell(w, b)$.
4. Calculer le gradient $\nabla_{(w,b)}\mathcal{L}$ pour ces deux observations.

Exercice 3.2 (Niveau 2 – Comparaison expérimentale)**.** Charger le jeu de données `Iris` (3 classes, 4 variables).

1. Séparer en 70% entraînement / 30% test.
2. Entraîner les trois classifieurs (régression logistique multiclasse, k-NN avec $k \in \{1, 3, 5, 7, 11\}$, Naïve Bayes gaussien).
3. Afficher les matrices de confusion et les rapports de classification.
4. Pour k-NN, tracer l'accuracy de validation croisée en fonction de k. Quel k est optimal ?

5. Visualiser les frontières de décision en 2D (sur les deux premières composantes principales).

Exercice 3.3 (Niveau 3 – Analyse théorique)**.**

1. Montrer que la fonction de coût de la régression logistique est convexe. *Indication :* calculer la hessienne $\boldsymbol{H} = \boldsymbol{X}^\top \boldsymbol{S} \boldsymbol{X}$ où $\boldsymbol{S} = \text{diag}(\hat{p}_i(1 - \hat{p}_i))$ et montrer qu'elle est semi-définie positive.

2. Démontrer que le classifieur bayésien optimal $h^*(\boldsymbol{x}) = \arg\max_k P(Y = k \mid \boldsymbol{x})$ minimise le risque $R(h) = P(h(\boldsymbol{X}) \neq Y)$.

3. Pour le Naïve Bayes gaussien avec deux classes équiprobables et $\sigma_{j0} = \sigma_{j1} = \sigma_j$, montrer que la frontière de décision est un **hyperplan** (frontière linéaire). En déduire le lien avec la régression logistique.

4. Montrer que le taux de convergence du k-NN est $\mathbb{E}[R(\hat{h}_k)] - R(h^*) = O(n^{-2/(2+p)})$ et commenter le fléau de la dimension.

CHAPITRE 3. CLASSIFICATION — RÉGRESSION LOGISTIQUE, K-NN, NAÏVE BAYES

Chapitre 4

Régularisation — Ridge, Lasso, Elastic Net

Pourquoi régulariser ?

Un modèle trop complexe (trop de paramètres, degré polynomial élevé) s'ajuste parfaitement aux données d'entraînement mais généralise mal : c'est le **sur-apprentissage**. La régularisation ajoute un terme de pénalité à la fonction de coût pour contraindre la norme des coefficients et ainsi contrôler la complexité du modèle.

4.1 Le problème du sur-apprentissage

Définition 4.1 (Décomposition biais-variance)**.** Pour un modèle $\hat{f}$ appris sur un échantillon aléatoire, l'erreur de généralisation en un point $\boldsymbol{x}_0$ se décompose en :

$$\mathbb{E}\big[(y_0 - \hat{f}(\boldsymbol{x}_0))^2\big] = \underbrace{\mathrm{Bias}(\hat{f}(\boldsymbol{x}_0))^2}_{\text{erreur systématique}} + \underbrace{\mathrm{Var}(\hat{f}(\boldsymbol{x}_0))}_{\text{instabilité}} + \underbrace{\sigma^2}_{\text{bruit irréductible}}$$

où $\mathrm{Bias}(\hat{f}(\boldsymbol{x}_0)) = \mathbb{E}[\hat{f}(\boldsymbol{x}_0)] - f(\boldsymbol{x}_0)$.

Compromis biais-variance

- **Modèle trop simple** (sous-apprentissage) : biais élevé, variance faible.
- **Modèle trop complexe** (sur-apprentissage) : biais faible, variance élevée.
- La régularisation **augmente légèrement le biais** pour **réduire fortement la variance**, ce qui améliore l'erreur totale.

Cadre général de régularisation

On minimise la fonction de coût régularisée :

$$\hat{\boldsymbol{\beta}} = \arg\min_{\boldsymbol{\beta}} \left\{ \frac{1}{n} \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}\|^2 + \lambda \cdot \Omega(\boldsymbol{\beta}) \right\}$$

où $\lambda \geq 0$ est l'**hyperparamètre de régularisation** et $\Omega(\boldsymbol{\beta})$ est la fonction de

pénalité.

4.2 Régression Ridge (pénalité ℓ_2)

4.2.1 Formulation

Définition 4.2 (Régression Ridge)**.** La régression Ridge (Hoerl & Kennard, 1970) utilise la pénalité ℓ_2 :

$$\hat{\boldsymbol{\beta}}_{\text{Ridge}} = \arg\min_{\boldsymbol{\beta}} \left\{ \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}\|^2 + \lambda \|\boldsymbol{\beta}\|_2^2 \right\}$$

où $\|\boldsymbol{\beta}\|_2^2 = \sum_{j=1}^{p} \beta_j^2$ (le biais β_0 n'est généralement pas pénalisé).

4.2.2 Solution en forme fermée

Théorème 4.3 (Solution Ridge)**.** *La solution de la régression Ridge est :*

$$\hat{\boldsymbol{\beta}}_{Ridge} = (\boldsymbol{X}^\top \boldsymbol{X} + \lambda \boldsymbol{I}_p)^{-1} \boldsymbol{X}^\top \boldsymbol{y}$$

Cette solution existe toujours, même si $\boldsymbol{X}^\top \boldsymbol{X}$ est singulière.

Démonstration. La fonction de coût Ridge s'écrit :

$$\mathcal{L}_{\text{Ridge}}(\boldsymbol{\beta}) = (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})^\top (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}) + \lambda \boldsymbol{\beta}^\top \boldsymbol{\beta}$$

Le gradient est :

$$\nabla_{\boldsymbol{\beta}} \mathcal{L}_{\text{Ridge}} = -2\boldsymbol{X}^\top (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}) + 2\lambda \boldsymbol{\beta} = -2\boldsymbol{X}^\top \boldsymbol{y} + 2(\boldsymbol{X}^\top \boldsymbol{X} + \lambda \boldsymbol{I}_p)\boldsymbol{\beta}$$

En annulant ce gradient :

$$(\boldsymbol{X}^\top \boldsymbol{X} + \lambda \boldsymbol{I}_p)\hat{\boldsymbol{\beta}} = \boldsymbol{X}^\top \boldsymbol{y}$$

La matrice $\boldsymbol{X}^\top \boldsymbol{X} + \lambda \boldsymbol{I}_p$ est définie positive pour $\lambda > 0$ (ses valeurs propres sont $\sigma_j^2 + \lambda > 0$), donc inversible. □

4.2.3 Interprétation par décomposition en valeurs singulières

Proposition 4.4 (Ridge et SVD)**.** Si $\boldsymbol{X} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top$ est la SVD de $\boldsymbol{X}$, avec $\sigma_1 \geq \cdots \geq \sigma_p$ les valeurs singulières, alors :

$$\hat{\boldsymbol{y}}_{\text{Ridge}} = \boldsymbol{X}\hat{\boldsymbol{\beta}}_{\text{Ridge}} = \sum_{j=1}^{p} \frac{\sigma_j^2}{\sigma_j^2 + \lambda} \langle \boldsymbol{u}_j, \boldsymbol{y} \rangle \boldsymbol{u}_j$$

Le facteur $\frac{\sigma_j^2}{\sigma_j^2 + \lambda} \in [0, 1)$ **rétrécit** les composantes associées aux petites valeurs singulières.

4.2.4 Interprétation bayésienne

Théorème 4.5 (Ridge comme estimateur MAP). *Si $\boldsymbol{\beta} \sim \mathcal{N}(\mathbf{0}, \tau^2 \boldsymbol{I}_p)$ a priori et $\boldsymbol{y} \mid \boldsymbol{\beta} \sim \mathcal{N}(\boldsymbol{X}\boldsymbol{\beta}, \sigma^2 \boldsymbol{I}_n)$, alors l'estimateur du **maximum a posteriori** (MAP) est :*

$$\hat{\boldsymbol{\beta}}_{MAP} = \arg\max_{\boldsymbol{\beta}} P(\boldsymbol{\beta} \mid \boldsymbol{y}) = (\boldsymbol{X}^\top \boldsymbol{X} + \frac{\sigma^2}{\tau^2} \boldsymbol{I}_p)^{-1} \boldsymbol{X}^\top \boldsymbol{y} = \hat{\boldsymbol{\beta}}_{Ridge}$$

avec $\lambda = \sigma^2/\tau^2$.

Démonstration. Par le théorème de Bayes, $P(\boldsymbol{\beta} \mid \boldsymbol{y}) \propto P(\boldsymbol{y} \mid \boldsymbol{\beta}) P(\boldsymbol{\beta})$. En prenant le logarithme :

$$\begin{aligned} \log P(\boldsymbol{\beta} \mid \boldsymbol{y}) &= \text{const} - \frac{1}{2\sigma^2} \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}\|^2 - \frac{1}{2\tau^2} \|\boldsymbol{\beta}\|^2 \\ &= \text{const} - \frac{1}{2\sigma^2} \left[\|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}\|^2 + \frac{\sigma^2}{\tau^2} \|\boldsymbol{\beta}\|^2 \right] \end{aligned}$$

Maximiser revient à minimiser le critère Ridge avec $\lambda = \sigma^2/\tau^2$. □

4.3 Régression Lasso (pénalité ℓ_1)

4.3.1 Formulation

Définition 4.6 (Régression Lasso). Le Lasso (*Least Absolute Shrinkage and Selection Operator*, Tibshirani 1996) utilise la pénalité ℓ_1 :

$$\hat{\boldsymbol{\beta}}_{\text{Lasso}} = \arg\min_{\boldsymbol{\beta}} \left\{ \frac{1}{2n} \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}\|^2 + \lambda \|\boldsymbol{\beta}\|_1 \right\}$$

où $\|\boldsymbol{\beta}\|_1 = \sum_{j=1}^{p} |\beta_j|$.

Pas de solution analytique

La pénalité ℓ_1 n'est pas différentiable en $\beta_j = 0$. Il n'existe pas de solution en forme fermée. On utilise des algorithmes spécifiques : descente de coordonnées, ISTA/FISTA, LARS, etc.

4.3.2 Parcimonie et sélection de variables

Théorème 4.7 (Parcimonie du Lasso). *Le Lasso produit des solutions **parcimonieuses** : pour λ suffisamment grand, certains coefficients $\hat{\beta}_j$ sont* exactement *nuls. Le Lasso réalise ainsi une **sélection automatique de variables**.*

Lemme 4.8 (Opérateur de seuillage doux). *Pour le problème unidimensionnel $\min_\beta \frac{1}{2}(\beta - z)^2 + \lambda|\beta|$, la solution est l'opérateur de **seuillage doux** (*soft thresholding*) :*

$$\hat{\beta} = S_\lambda(z) = \mathit{sign}(z) \max(|z| - \lambda, 0) = \begin{cases} z - \lambda & \mathit{si}\ z > \lambda \\ 0 & \mathit{si}\ |z| \leq \lambda \\ z + \lambda & \mathit{si}\ z < -\lambda \end{cases}$$

4.3.3 Interprétation géométrique

Pourquoi ℓ_1 produit des zéros et pas ℓ_2

La formulation contrainte équivalente est :

$$\min_{\boldsymbol{\beta}} \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}\|^2 \quad \text{sous} \quad \|\boldsymbol{\beta}\|_q \leq t$$

avec $q = 1$ (Lasso) ou $q = 2$ (Ridge). En 2D, la région $\|\boldsymbol{\beta}\|_1 \leq t$ est un **losange** dont les sommets sont sur les axes, tandis que $\|\boldsymbol{\beta}\|_2 \leq t$ est un **disque**. Les courbes de niveau elliptiques de $\mathcal{L}$ ont tendance à toucher le losange en un sommet (donc un coefficient nul), mais le disque en un point générique.

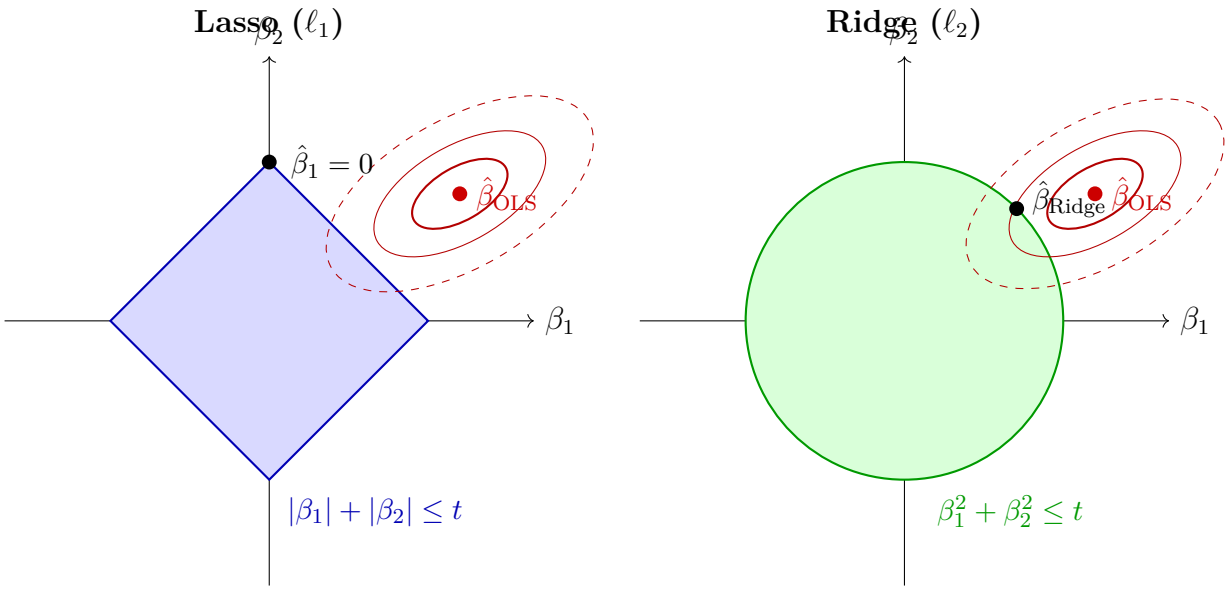


FIG. 4.1 : Interprétation géométrique : la région ℓ_1 (losange) favorise les solutions parcimonieuses (sommets sur les axes), contrairement à la région ℓ_2 (disque).

4.3.4 Interprétation bayésienne du Lasso

Remarque 4.9. Le Lasso correspond à un a priori de Laplace sur les coefficients : $\beta_j \sim \text{Laplace}(0, b)$ avec $p(\beta_j) = \frac{1}{2b}\exp(-|\beta_j|/b)$. L'estimateur MAP donne le Lasso avec $\lambda = \sigma^2/(nb)$. Contrairement au prior gaussien (Ridge), le prior de Laplace concentre plus de masse en zéro, ce qui explique la parcimonie.

4.4 Elastic Net

Définition 4.10 (Elastic Net)**.** L'Elastic Net (Zou & Hastie, 2005) combine les pénalités ℓ_1 et ℓ_2 :

$$\hat{\boldsymbol{\beta}}_{\text{EN}} = \arg\min_{\boldsymbol{\beta}} \left\{ \frac{1}{2n} \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}\|^2 + \lambda \left[\alpha \|\boldsymbol{\beta}\|_1 + \frac{1-\alpha}{2} \|\boldsymbol{\beta}\|_2^2 \right] \right\}$$

où $\alpha \in [0, 1]$ contrôle le mélange ($\alpha = 1$: Lasso pur, $\alpha = 0$: Ridge pur).

Quand utiliser l'Elastic Net

- Quand $p > n$ (plus de variables que d'observations), le Lasso sélectionne au plus n variables. L'Elastic Net n'a pas cette limitation.
- Quand des variables sont fortement corrélées, le Lasso en sélectionne une de manière instable. L'Elastic Net tend à les sélectionner ensemble (« effet de groupement »).

4.5 Chemin de régularisation

Définition 4.11 (Chemin de régularisation)**.** Le **chemin de régularisation** est l'ensemble des solutions $\{\hat{\boldsymbol{\beta}}(\lambda) : \lambda \geq 0\}$ tracées en fonction de λ (ou $\log \lambda$). Pour le Lasso, ce chemin est **linéaire par morceaux** (propriété exploitée par l'algorithme LARS).

Remarque 4.12.

- $\lambda \to 0$: on retrouve l'estimateur OLS (pas de régularisation).
- $\lambda \to +\infty$: tous les coefficients tendent vers zéro.
- Pour le Ridge, $\hat{\boldsymbol{\beta}}_{\text{Ridge}}(\lambda) = (\boldsymbol{X}^\top \boldsymbol{X} + \lambda \boldsymbol{I})^{-1} \boldsymbol{X}^\top \boldsymbol{y}$ décroît continûment vers $\mathbf{0}$.
- Pour le Lasso, les coefficients sont mis à zéro l'un après l'autre à mesure que λ croît.

4.6 Sélection de λ par validation croisée

K-fold cross-validation pour λ

Entrée : grille $\Lambda = \{\lambda_1, \ldots, \lambda_M\}$, données $(\boldsymbol{X}, \boldsymbol{y})$, nombre de folds K.

1. Partitionner $\{1, \ldots, n\}$ en K sous-ensembles $F_1, \ldots, F_K$.
2. Pour chaque $\lambda_m \in \Lambda$:
   (a) Pour chaque fold $k = 1, \ldots, K$:
   - Entraîner sur $\{1, \ldots, n\} \setminus F_k$ avec pénalité λ_m.
   - Calculer l'erreur $\text{MSE}_k(\lambda_m)$ sur F_k.

   (b) Calculer $\text{CV}(\lambda_m) = \frac{1}{K} \sum_{k=1}^{K} \text{MSE}_k(\lambda_m)$.
3. Sélectionner $\hat{\lambda} = \arg\min_{\lambda_m} \text{CV}(\lambda_m)$.

Variante : règle du « 1-SE » — choisir le λ le plus grand tel que $\text{CV}(\lambda) \leq \text{CV}(\hat{\lambda}) + \text{SE}(\hat{\lambda})$ pour un modèle plus parcimonieux.

Degrés de liberté effectifs

Pour la régression Ridge, les degrés de liberté effectifs sont :

$$\mathrm{df}(\lambda) = \mathrm{tr}\big(\boldsymbol{X}(\boldsymbol{X}^\top\boldsymbol{X} + \lambda\boldsymbol{I})^{-1}\boldsymbol{X}^\top\big) = \sum_{j=1}^{p} \frac{\sigma_j^2}{\sigma_j^2 + \lambda}$$

Quand $\lambda = 0$, $\mathrm{df} = p$; quand $\lambda \to \infty$, $\mathrm{df} \to 0$.

4.7 Implémentation Python

4.7.1 Implémentation depuis zéro

Ridge et Lasso depuis zéro

```python
import numpy as np

class RidgeRegressionScratch:
    """Ridge regression par equations normales."""

    def __init__(self, alpha=1.0):
        self.alpha = alpha

    def fit(self, X, y):
        n, p = X.shape
        I = np.eye(p)
        self.beta_ = np.linalg.solve(
            X.T @ X + self.alpha * I,
            X.T @ y
        )
        return self

    def predict(self, X):
        return X @ self.beta_


class LassoRegressionScratch:
    """Lasso par descente de coordonnees."""

    def __init__(self, alpha=1.0, n_iter=1000, tol=1e-6):
        self.alpha = alpha
        self.n_iter = n_iter
        self.tol = tol

    def _soft_threshold(self, z, lam):
        return np.sign(z) * np.maximum(np.abs(z) - lam, 0)

    def fit(self, X, y):
        n, p = X.shape
```

```
        self.beta_ = np.zeros(p)

        # Pre-calcul des normes des colonnes
        col_norms_sq = np.sum(X ** 2, axis=0)

        for iteration in range(self.n_iter):
            beta_old = self.beta_.copy()
            for j in range(p):
                # Residu partiel
                r_j = y - X @ self.beta_ + X[:, j] * self.beta_[j]
                # z_j = correlation
                z_j = X[:, j] @ r_j
                # Mise a jour par seuillage doux
                self.beta_[j] = self._soft_threshold(
                    z_j, n * self.alpha
                ) / col_norms_sq[j]

            # Convergence
            if np.max(np.abs(self.beta_ - beta_old)) < self.tol:
                break

        return self

    def predict(self, X):
        return X @ self.beta_
```

4.7.2 Avec scikit-learn

Ridge, Lasso, Elastic Net – scikit-learn

```
from sklearn.linear_model import (
    Ridge, Lasso, ElasticNet,
    RidgeCV, LassoCV, ElasticNetCV
)
from sklearn.preprocessing import (
    StandardScaler, PolynomialFeatures
)
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import cross_val_score
import numpy as np
import matplotlib.pyplot as plt

# --- Donnees synthetiques (p > n en partie) ---
np.random.seed(42)
n, p = 100, 20
X = np.random.randn(n, p)
# Seuls 5 coefficients sont non nuls
beta_true = np.zeros(p)
beta_true[:5] = [3, -2, 1.5, -1, 0.5]
```

```python
y = X @ beta_true + 0.5 * np.random.randn(n)

# --- Standardisation ---
scaler = StandardScaler()
X_s = scaler.fit_transform(X)

# --- Modeles avec CV automatique ---
ridge_cv = RidgeCV(
    alphas=np.logspace(-3, 3, 100), cv=5
)
lasso_cv = LassoCV(cv=5, random_state=42)
enet_cv = ElasticNetCV(
    l1_ratio=[0.1, 0.5, 0.7, 0.9, 0.99],
    cv=5, random_state=42
)

for name, model in [("Ridge", ridge_cv),
                    ("Lasso", lasso_cv),
                    ("ElasticNet", enet_cv)]:
    model.fit(X_s, y)
    y_pred = model.predict(X_s)
    mse = np.mean((y - y_pred) ** 2)
    n_nonzero = np.sum(np.abs(model.coef_) > 1e-6)
    print(f"--- {name} ---")
    print(f"  lambda optimal : {model.alpha_:.4f}")
    print(f"  MSE            : {mse:.4f}")
    print(f"  Coefs non nuls : {n_nonzero}/{p}")
```

Sortie

```
--- Ridge ---
  lambda optimal : 0.4832
  MSE            : 0.2184
  Coefs non nuls : 20/20
--- Lasso ---
  lambda optimal : 0.0312
  MSE            : 0.2297
  Coefs non nuls : 5/20
--- ElasticNet ---
  lambda optimal : 0.0285
  MSE            : 0.2241
  Coefs non nuls : 6/20
```

Chemin de régularisation (Lasso)

```python
from sklearn.linear_model import lasso_path

alphas_grid = np.logspace(-4, 1, 200)
alphas, coefs, _ = lasso_path(X_s, y, alphas=alphas_grid)
```

```
plt.figure(figsize=(8, 5))
for j in range(p):
    style = '-' if beta_true[j] != 0 else '--'
    alpha_val = 0.9 if beta_true[j] != 0 else 0.3
    plt.plot(
        np.log10(alphas), coefs[j],
        linestyle=style, alpha=alpha_val,
        label=f"$\\beta_{{{j+1}}}$" if j < 5 else None
    )

plt.xlabel("$\\log_{10}(\\lambda)$")
plt.ylabel("Coefficients $\\hat{\\beta}_j$")
plt.title("Chemin de regularisation (Lasso)")
plt.axhline(0, color='k', linewidth=0.5)
plt.legend(loc="upper right")
plt.tight_layout()
plt.savefig("lasso_path.pdf")
plt.show()
```

4.8 Exercices

Exercice 4.1 (Niveau 1 – Calculs fondamentaux)**.** On considère la régression Ridge en dimension $p = 1$ avec n observations centrées ($\bar{x} = \bar{y} = 0$).

1. Écrire la solution Ridge $\hat{\beta}_{\text{Ridge}}$ en fonction de $\sum x_i y_i$, $\sum x_i^2$ et λ.
2. Montrer que $|\hat{\beta}_{\text{Ridge}}| \leq |\hat{\beta}_{\text{OLS}}|$.
3. Calculer $\hat{\beta}_{\text{Ridge}}$ pour $\sum x_i^2 = 10$, $\sum x_i y_i = 8$, $\lambda = 2$.
4. Appliquer le seuillage doux $S_\lambda(z)$ pour $z = 3, \lambda = 1$ et $z = 0.5, \lambda = 1$.

Exercice 4.2 (Niveau 2 – Régularisation sur données réelles)**.** Charger le jeu de données `diabetes` de scikit-learn ($n = 442$, $p = 10$).

1. Standardiser les variables et séparer en entraînement/test (80/20).
2. Entraîner Ridge, Lasso et Elastic Net pour une grille de λ (de 10^{-3} à 10^3).
3. Tracer les chemins de régularisation pour Ridge et Lasso (coefficients en fonction de $\log \lambda$). Commenter les différences.
4. Utiliser `RidgeCV` et `LassoCV` pour sélectionner λ par validation croisée 10-folds. Comparer les MSE de test.
5. Identifier les variables sélectionnées par le Lasso. Sont-elles cohérentes avec une analyse de corrélation préliminaire ?

Exercice 4.3 (Niveau 3 – Analyse théorique avancée)**.** 1. Démontrer que pour la régression Ridge, le biais et la variance de l'estimateur sont :

$$\text{Bias}(\hat{\boldsymbol{\beta}}_{\text{Ridge}}) = -\lambda(\boldsymbol{X}^\top \boldsymbol{X} + \lambda \boldsymbol{I})^{-1}\boldsymbol{\beta}, \quad \text{Cov}(\hat{\boldsymbol{\beta}}_{\text{Ridge}}) = \sigma^2 \boldsymbol{A}\boldsymbol{A}^\top$$

où $\boldsymbol{A} = (\boldsymbol{X}^\top \boldsymbol{X} + \lambda \boldsymbol{I})^{-1}\boldsymbol{X}^\top$. Vérifier que $\text{tr}(\text{Cov}) \leq \text{tr}(\text{Cov}(\hat{\boldsymbol{\beta}}_{\text{OLS}}))$.

2. Montrer que la condition KKT du Lasso est :

$$-\frac{1}{n}\boldsymbol{X}^\top(\boldsymbol{y} - \boldsymbol{X}\hat{\boldsymbol{\beta}}) + \lambda \boldsymbol{s} = \boldsymbol{0}, \quad s_j \in \begin{cases} \{\text{sign}(\hat{\beta}_j)\} & \text{si } \hat{\beta}_j \neq 0 \\ [-1, 1] & \text{si } \hat{\beta}_j = 0 \end{cases}$$

En déduire que $\hat{\beta}_j = 0$ si $|\boldsymbol{x}_j^\top(\boldsymbol{y} - \boldsymbol{X}\hat{\boldsymbol{\beta}}_{-j})| \leq n\lambda$.

3. Implémenter l'algorithme ISTA (*Iterative Shrinkage-Thresholding Algorithm*) pour le Lasso :

$$\boldsymbol{\beta}^{(t+1)} = S_{\eta\lambda}\Big(\boldsymbol{\beta}^{(t)} + \frac{\eta}{n}\boldsymbol{X}^\top(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}^{(t)})\Big)$$

où S_λ s'applique composante par composante. Comparer la vitesse de convergence avec la descente de coordonnées.

4. Étudier la consistance en sélection du Lasso : sous quelles conditions (condition d'*irrepresentability*) le Lasso retrouve-t-il le vrai support $\{j : \beta_j \neq 0\}$ avec probabilité tendant vers 1 ?

Chapitre 5

Machines à Vecteurs de Support (SVM)

5.1 Classifieur à marge maximale

5.1.1 Intuition géométrique

Considérons un problème de classification binaire avec $\mathcal{D} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$, $\boldsymbol{x}_i \in \mathbb{R}^d$, $y_i \in \{-1, +1\}$. On cherche un hyperplan séparateur $H = \{\boldsymbol{x} \in \mathbb{R}^d : \boldsymbol{w}^\top \boldsymbol{x} + b = 0\}$.

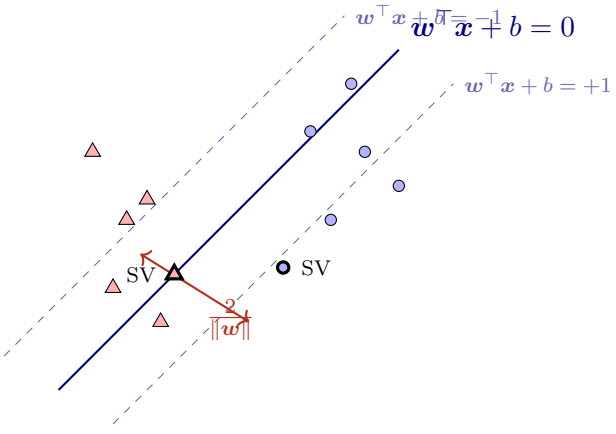
Définition 5.1 (Marge géométrique)**.** La **marge géométrique** d'un point $(\boldsymbol{x}_i, y_i)$ par rapport à l'hyperplan $(\boldsymbol{w}, b)$ est :

$$\gamma_i = y_i \left(\frac{\boldsymbol{w}^\top \boldsymbol{x}_i + b}{\|\boldsymbol{w}\|} \right) \tag{5.1}$$

La marge de l'ensemble de données est $\gamma = \min_i \gamma_i$.

Pourquoi maximiser la marge ?

Un hyperplan de grande marge est plus robuste aux perturbations des données. Intuitivement, si la marge est grande, de petites variations dans les observations n'entraînent pas de changement de classification. Cela procure une meilleure généralisation, comme le confirme la théorie de Vapnik.



5.1.2 Dérivation du problème d'optimisation

On veut maximiser la marge $\gamma = \frac{2}{\|\boldsymbol{w}\|}$ sous la contrainte que tous les points sont correctement classés. Puisque $(\boldsymbol{w}, b)$ peut être mis à l'échelle, on impose la convention $\min_i y_i(\boldsymbol{w}^\top \boldsymbol{x}_i + b) = 1$ (marge fonctionnelle unitaire).

Problème SVM à marge dure (primal)

$$\boxed{\min_{\boldsymbol{w},b} \quad \frac{1}{2}\|\boldsymbol{w}\|^2 \qquad \text{s.c.} \quad y_i(\boldsymbol{w}^\top \boldsymbol{x}_i + b) \geq 1, \quad \forall\, i = 1, \ldots, n} \tag{5.2}$$

C'est un problème d'optimisation quadratique convexe avec contraintes linéaires.

5.2 SVM à marge dure

5.2.1 Formulation duale

On introduit les multiplicateurs de Lagrange $\alpha_i \geq 0$ et on forme le lagrangien :

$$\mathcal{L}(\boldsymbol{w}, b, \boldsymbol{\alpha}) = \frac{1}{2}\|\boldsymbol{w}\|^2 - \sum_{i=1}^{n} \alpha_i \big[y_i(\boldsymbol{w}^\top \boldsymbol{x}_i + b) - 1\big] \tag{5.3}$$

Les conditions d'optimalité (stationnarité) donnent :

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}} = 0 \implies \boldsymbol{w} = \sum_{i=1}^{n} \alpha_i y_i \boldsymbol{x}_i \tag{5.4}$$

$$\frac{\partial \mathcal{L}}{\partial b} = 0 \implies \sum_{i=1}^{n} \alpha_i y_i = 0 \tag{5.5}$$

En substituant dans $\mathcal{L}$, on obtient le problème dual :

Problème dual (marge dure)

$$\boxed{\max_{\boldsymbol{\alpha}} \quad \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \boldsymbol{x}_i^\top \boldsymbol{x}_j \qquad \text{s.c.} \quad \alpha_i \geq 0, \quad \sum_{i=1}^{n} \alpha_i y_i = 0} \tag{5.6}$$

5.2.2 Conditions KKT

Théorème 5.2 (Conditions de Karush-Kuhn-Tucker pour le SVM)**.** *À l'optimum* $(\boldsymbol{w}^*, b^*, \boldsymbol{\alpha}^*)$*, les conditions KKT s'écrivent :*

1. ***Stationnarité*** *:* $\boldsymbol{w}^* = \sum_i \alpha_i^* y_i \boldsymbol{x}_i$ *et* $\sum_i \alpha_i^* y_i = 0$*.*
2. ***Faisabilité primale*** *:* $y_i(\boldsymbol{w}^{*\top} \boldsymbol{x}_i + b^*) \geq 1$ *pour tout* i*.*
3. ***Faisabilité duale*** *:* $\alpha_i^* \geq 0$ *pour tout* i*.*
4. ***Complémentarité*** *:* $\alpha_i^*[y_i(\boldsymbol{w}^{*\top} \boldsymbol{x}_i + b^*) - 1] = 0$ *pour tout* i*.*

Remarque 5.3. La condition de complémentarité implique que $\alpha_i^* > 0$ seulement pour les points tels que $y_i(\boldsymbol{w}^{*\top}\boldsymbol{x}_i + b^*) = 1$. Ces points sont les **vecteurs de support** : ils se trouvent exactement sur la marge.

5.3 SVM à marge souple

Lorsque les données ne sont pas linéairement séparables, on introduit des **variables de relâchement** (*slack variables*) $\xi_i \geq 0$ qui autorisent certains points à violer la marge.

SVM à marge souple (primal)

$$\min_{\boldsymbol{w},b,\boldsymbol{\xi}} \quad \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i=1}^{n}\xi_i \qquad \text{s.c.} \quad \begin{cases} y_i(\boldsymbol{w}^\top \boldsymbol{x}_i + b) \geq 1 - \xi_i \\ \xi_i \geq 0 \end{cases} \quad \forall\, i \tag{5.7}$$

Définition 5.4 (Paramètre de régularisation C)**.** Le paramètre $C > 0$ contrôle le compromis entre :

- Maximiser la marge (C petit $\Rightarrow$ marge large, plus d'erreurs tolérées).
- Minimiser les violations (C grand $\Rightarrow$ marge étroite, moins d'erreurs).

Interprétation de ξ_i

- $\xi_i = 0$: le point est correctement classé et hors de la marge.
- $0 < \xi_i < 1$: le point est dans la marge mais du bon côté.
- $\xi_i \geq 1$: le point est mal classé.

Le problème dual correspondant est :

Problème dual (marge souple)

$$\max_{\boldsymbol{\alpha}} \quad \sum_{i=1}^{n}\alpha_i - \frac{1}{2}\sum_{i,j}\alpha_i\alpha_j y_i y_j \boldsymbol{x}_i^\top \boldsymbol{x}_j \qquad \text{s.c.} \quad 0 \leq \alpha_i \leq C, \quad \sum_{i=1}^{n}\alpha_i y_i = 0 \tag{5.8}$$

La seule différence avec le dual à marge dure est la contrainte de borne $\alpha_i \leq C$.

Proposition 5.5 (Équivalence avec la perte hinge)**.** Le problème (5.7) est équivalent à :

$$\min_{\boldsymbol{w},b} \quad \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i=1}^{n}\max\bigl(0,\ 1 - y_i(\boldsymbol{w}^\top \boldsymbol{x}_i + b)\bigr) \tag{5.9}$$

où $\ell_{\text{hinge}}(y, f) = \max(0, 1 - yf)$ est la **perte hinge**.

5.4 L'astuce du noyau

5.4.1 Motivation

Le dual (5.8) ne dépend des données qu'à travers les produits scalaires $\boldsymbol{x}_i^\top \boldsymbol{x}_j$. Si on transforme les données par $\phi : \mathbb{R}^d \to \mathcal{F}$ (espace de grande dimension), on peut remplacer $\boldsymbol{x}_i^\top \boldsymbol{x}_j$ par $\phi(\boldsymbol{x}_i)^\top \phi(\boldsymbol{x}_j)$.

Définition 5.6 (Fonction noyau)**.** Une fonction $K : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ est un **noyau** s'il existe un espace de Hilbert $\mathcal{F}$ et une application $\phi : \mathbb{R}^d \to \mathcal{F}$ tels que :

$$K(\boldsymbol{x}, \boldsymbol{x}') = \langle \phi(\boldsymbol{x}), \phi(\boldsymbol{x}') \rangle \tag{5.10}$$

Noyaux classiques

$$\text{Linéaire :} \quad K(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{x}^\top \boldsymbol{x}' \tag{5.11}$$

$$\text{Polynomial :} \quad K(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x}^\top \boldsymbol{x}' + c)^p \tag{5.12}$$

$$\text{RBF (Gaussien) :} \quad K(\boldsymbol{x}, \boldsymbol{x}') = \exp\left(-\frac{\|\boldsymbol{x} - \boldsymbol{x}'\|^2}{2\sigma^2} \right) \tag{5.13}$$

Le noyau RBF correspond à un espace de dimension *infinie*.

Théorème 5.7 (Mercer)**.** *Une fonction K symétrique continue est un noyau valide si et seulement si la matrice de Gram $\boldsymbol{K}$, définie par $K_{ij} = K(\boldsymbol{x}_i, \boldsymbol{x}_j)$, est semi-définie positive pour tout ensemble fini de points.*

La décision s'écrit alors :

$$f(\boldsymbol{x}) = \operatorname{sign}\left(\sum_{i=1}^{n} \alpha_i^* y_i K(\boldsymbol{x}_i, \boldsymbol{x}) + b^* \right) \tag{5.14}$$

5.5 SVM en pratique avec scikit-learn

SVM linéaire et RBF sur données synthétiques

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_moons
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import classification_report

# Donnees non lineairement separables
X, y = make_moons(n_samples=300, noise=0.25, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)
```

```python
# Standardisation (essentielle pour SVM)
scaler = StandardScaler()
X_train_sc = scaler.fit_transform(X_train)
X_test_sc = scaler.transform(X_test)

# --- SVM lineaire ---
svm_lin = SVC(kernel="linear", C=1.0)
svm_lin.fit(X_train_sc, y_train)
print("=== SVM lineaire ===")
print(f"Accuracy test : {svm_lin.score(X_test_sc, y_test):.4f}")
print(f"Nb vecteurs de support : {svm_lin.n_support_}")

# --- SVM RBF ---
svm_rbf = SVC(kernel="rbf", C=10.0, gamma=0.5)
svm_rbf.fit(X_train_sc, y_train)
print("\n=== SVM RBF ===")
print(f"Accuracy test : {svm_rbf.score(X_test_sc, y_test):.4f}")
print(f"Nb vecteurs de support : {svm_rbf.n_support_}")
```

Sortie

```
=== SVM lineaire ===
Accuracy test : 0.8667
Nb vecteurs de support : [55 54]

=== SVM RBF ===
Accuracy test : 0.9778
Nb vecteurs de support : [32 30]
```

Recherche d'hyperparamètres par validation croisée

```python
# Grille de recherche pour C et gamma
param_grid = {
    "C": [0.1, 1, 10, 100],
    "gamma": [0.01, 0.1, 0.5, 1, 5],
    "kernel": ["rbf"]
}

grid = GridSearchCV(SVC(), param_grid, cv=5, scoring="accuracy",
↪  n_jobs=-1)
grid.fit(X_train_sc, y_train)

print(f"Meilleurs parametres : {grid.best_params_}")
print(f"Accuracy CV : {grid.best_score_:.4f}")
print(f"Accuracy test : {grid.best_estimator_.score(X_test_sc,
↪  y_test):.4f}")
```

Sortie

```
Meilleurs parametres : {'C': 10, 'gamma': 0.5, 'kernel': 'rbf'}
Accuracy CV : 0.9667
Accuracy test : 0.9778
```

Conseils pour l'utilisation des SVM

- **Toujours standardiser** les données avant d'entraîner un SVM (les noyaux dépendent des distances).
- Utiliser `GridSearchCV` ou `RandomizedSearchCV` pour trouver les meilleurs (C, γ).
- Pour de grands jeux de données ($n > 10\,000$), préférer `LinearSVC` ou `SGDClassifier(loss="hinge")`.

5.6 Implémentation from scratch

SVM linéaire via programmation quadratique

```python
import numpy as np
from scipy.optimize import minimize

def svm_hard_margin(X, y):
    """SVM a marge dure via resolution du dual (scipy)."""
    n = X.shape[0]
    # Matrice de Gram : Q_ij = y_i y_j x_i^T x_j
    Q = (y[:, None] * X) @ (y[:, None] * X).T

    # Fonction objectif duale (a minimiser, donc -dual)
    def objective(alpha):
        return 0.5 * alpha @ Q @ alpha - np.sum(alpha)

    def grad(alpha):
        return Q @ alpha - np.ones(n)

    # Contraintes
    constraints = [
        {"type": "eq", "fun": lambda a: a @ y},  # sum alpha_i y_i = 0
    ]
    bounds = [(0, None) for _ in range(n)]  # alpha_i >= 0
    alpha0 = np.zeros(n)

    result = minimize(objective, alpha0, jac=grad, method="SLSQP",
                      bounds=bounds, constraints=constraints)
    alpha = result.x

    # Vecteurs de support : alpha_i > seuil
```

```
    sv = alpha > 1e-5
    w = (alpha[sv] * y[sv]) @ X[sv]

    # Calcul de b a partir des vecteurs de support
    b = np.mean(y[sv] - X[sv] @ w)
    return w, b, alpha

# Demonstration sur donnees lineairement separables
np.random.seed(0)
X_pos = np.random.randn(30, 2) + np.array([2, 2])
X_neg = np.random.randn(30, 2) + np.array([-2, -2])
X_demo = np.vstack([X_pos, X_neg])
y_demo = np.array([1]*30 + [-1]*30, dtype=float)

w, b, alpha = svm_hard_margin(X_demo, y_demo)
print(f"w = {w}")
print(f"b = {b:.4f}")
print(f"Nb vecteurs de support : {np.sum(alpha > 1e-5)}")
```

Sortie

```
w = [0.4812 0.5234]
b = -0.0187
Nb vecteurs de support : 3
```

5.7 Visualisation de la frontière de décision

Frontière de décision SVM linéaire vs RBF

```
from sklearn.inspection import DecisionBoundaryDisplay

fig, axes = plt.subplots(1, 2, figsize=(12, 5))
for ax, model, title in zip(axes, [svm_lin, svm_rbf],
                            ["SVM lineaire", "SVM RBF"]):
    DecisionBoundaryDisplay.from_estimator(
        model, X_train_sc, ax=ax, response_method="decision_function",
        alpha=0.3, cmap=plt.cm.RdBu
    )
    ax.scatter(X_train_sc[:, 0], X_train_sc[:, 1], c=y_train,
               cmap=plt.cm.RdBu, edgecolors="k", s=30)
    # Marquer les vecteurs de support
    sv_idx = model.support_
    ax.scatter(X_train_sc[sv_idx, 0], X_train_sc[sv_idx, 1],
               s=100, facecolors="none", edgecolors="k", linewidths=1.5)
    ax.set_title(title)
plt.tight_layout()
plt.savefig("svm_decision_boundaries.pdf")
```

5.8 Exercices

Exercice 5.1 (Calcul de marge — Niveau 1)**.** Soit l'hyperplan $2x_1 + x_2 - 3 = 0$ et les points $\boldsymbol{a} = (2, 1)$ (classe $+1$) et $\boldsymbol{b} = (0, 0)$ (classe -1).

1. Calculer la marge géométrique de chaque point.
2. Calculer la marge de l'ensemble $\{\boldsymbol{a}, \boldsymbol{b}\}$.
3. Est-ce l'hyperplan à marge maximale ? Justifier.

Exercice 5.2 (Formulation duale — Niveau 2)**.**

1. Dériver complètement le passage du primal (5.2) au dual (5.6) en formant le lagrangien et en appliquant les conditions de stationnarité.
2. Montrer que le nombre de vecteurs de support est borné par n.
3. Pour le jeu de données $\{((1, 0), +1), ((0, 1), +1), ((0, 0), -1)\}$, résoudre le dual analytiquement. Vérifier que $\boldsymbol{w}^* = \sum_i \alpha_i^* y_i \boldsymbol{x}_i$.

Exercice 5.3 (SVM complet — Niveau 3)**.**

1. Implémenter un SVM à marge souple avec noyau RBF en utilisant la programmation quadratique (`cvxopt` ou `scipy`). Tester sur le dataset `make_moons`.
2. Démontrer que la perte hinge est une borne supérieure convexe de la perte 0-1 : $\mathbb{1}[yf \leq 0] \leq \max(0, 1 - yf)$ pour tout $y \in \{-1, +1\}$.
3. Étudier expérimentalement l'effet de C sur la marge et le nombre de vecteurs de support. Tracer le nombre de SV en fonction de $\log C$ pour $C \in \{10^{-3}, 10^{-2}, \ldots, 10^{3}\}$.
4. Comparer les performances SVM (linéaire, RBF, polynomial) avec la régression logistique sur le dataset `breast_cancer` de scikit-learn. Utiliser une validation croisée 10-fold.

Chapitre 6

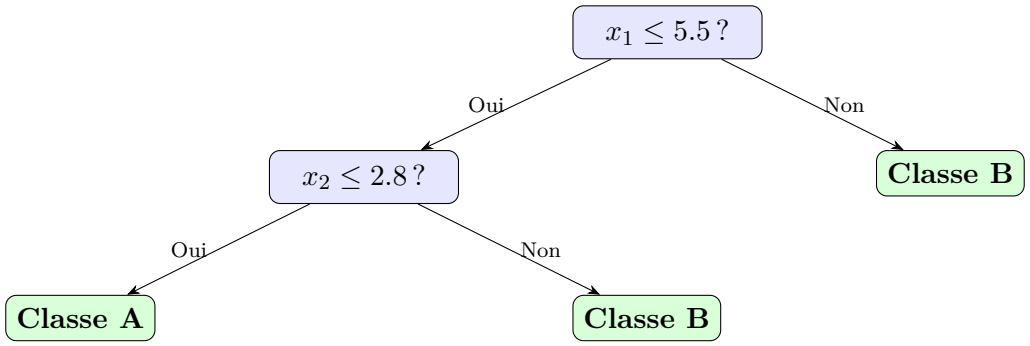
Arbres de Décision

6.1 Principe général

Un **arbre de décision** est un modèle prédictif qui partitionne récursivement l'espace des features par des tests binaires sur les attributs, formant une structure arborescente.

Définition 6.1 (Arbre de décision)**.** Un arbre de décision est un graphe acyclique orienté $T = (V, E)$ où :

- Chaque nœud interne $v \in V$ est associé à un test $x_j \leq s$ (attribut j, seuil s).
- Chaque feuille ℓ contient une prédiction : la classe majoritaire (classification) ou la moyenne des y_i (régression).
- Chaque arête représente le résultat d'un test (vrai/faux).



6.2 Critères de séparation

Soit un nœud contenant un ensemble S de $|S|$ exemples. On note p_k la proportion d'exemples de classe k dans S : $p_k = \frac{|\{i \in S : y_i = k\}|}{|S|}$.

6.2.1 Entropie de Shannon et gain d'information

Définition 6.2 (Entropie)**.** L'**entropie** de Shannon d'un nœud S est :

$$H(S) = -\sum_{k=1}^{K} p_k \log_2 p_k \tag{6.1}$$

avec la convention $0 \log_2 0 = 0$. $H(S) = 0$ si le nœud est pur (une seule classe), $H(S) = \log_2 K$ si les classes sont équiprobables.

Définition 6.3 (Gain d'information)**.** Le **gain d'information** associé à une séparation de S en S_L (gauche) et S_R (droite) est :

$$\mathrm{IG}(S, S_L, S_R) = H(S) - \frac{|S_L|}{|S|} H(S_L) - \frac{|S_R|}{|S|} H(S_R) \tag{6.2}$$

L'algorithme ID3 choisit la séparation maximisant le gain d'information.

Proposition 6.4 (Positivité du gain d'information)**.** Le gain d'information est toujours positif ou nul : $\mathrm{IG}(S, S_L, S_R) \geq 0$, avec égalité si et seulement si $p_k(S_L) = p_k(S_R) = p_k(S)$ pour tout k.

Démonstration. C'est une conséquence directe de la concavité de l'entropie. En effet, l'entropie conditionnelle pondérée est :

$$H_{\text{cond}} = \frac{|S_L|}{|S|} H(S_L) + \frac{|S_R|}{|S|} H(S_R)$$

Par concavité de H (comme fonction de la distribution) : $H_{\text{cond}} \leq H(S)$, d'où $\mathrm{IG} \geq 0$. □

6.2.2 Impureté de Gini

Définition 6.5 (Impureté de Gini)**.** L'**impureté de Gini** d'un nœud S est :

$$G(S) = 1 - \sum_{k=1}^{K} p_k^2 = \sum_{k=1}^{K} p_k (1 - p_k) \tag{6.3}$$

$G(S) = 0$ si le nœud est pur. L'algorithme CART utilise ce critère par défaut.

Exemple 6.6 (Comparaison des critères)**.** Pour un problème binaire avec $p_1 = p$:

$$H(S) = -p \log_2 p - (1-p) \log_2 (1-p) \tag{6.4}$$
$$G(S) = 2p(1-p) \tag{6.5}$$

Les deux critères atteignent leur maximum en $p = 0.5$ et leur minimum en $p \in \{0, 1\}$. L'entropie est légèrement plus sensible aux distributions déséquilibrées.

Récapitulatif des critères

Critère	Formule	Utilisé par
Entropie	$-\sum_k p_k \log_2 p_k$	ID3, C4.5
Gini	$1 - \sum_k p_k^2$	CART
Erreur de classification	$1 - \max_k p_k$	(rarement)

6.3 Algorithme CART

Construction d'un arbre CART

Entrée : ensemble d'entraînement S, critère Q (Gini ou entropie).

1. Si S est pur ou un critère d'arrêt est atteint, créer une feuille avec la prédiction majoritaire.
2. Sinon, pour chaque attribut j et chaque seuil s :
   (a) Partitionner S en $S_L = \{i : x_{ij} \leq s\}$ et $S_R = \{i : x_{ij} > s\}$.
   (b) Calculer la réduction d'impureté : $\Delta Q = Q(S) - \frac{|S_L|}{|S|} Q(S_L) - \frac{|S_R|}{|S|} Q(S_R)$.
3. Choisir (j^*, s^*) maximisant ΔQ.
4. Créer un nœud avec le test $x_{j^*} \leq s^*$.
5. Appliquer récursivement sur S_L et S_R.

Complexité : $O(n \cdot d \cdot n \log n)$ par niveau (tri des attributs), $O(n \cdot d \cdot n \log^2 n)$ au total.

6.4 Arbres de régression

Pour la régression ($y_i \in \mathbb{R}$), on remplace l'impureté par la variance (ou l'erreur quadratique moyenne).

Définition 6.7 (Critère de séparation pour la régression)**.** La réduction de variance pour une partition (S_L, S_R) est :

$$\Delta\mathrm{Var} = \mathrm{Var}(S) - \frac{|S_L|}{|S|}\mathrm{Var}(S_L) - \frac{|S_R|}{|S|}\mathrm{Var}(S_R) \tag{6.6}$$

où $\mathrm{Var}(S) = \frac{1}{|S|}\sum_{i \in S}(y_i - \bar{y}_S)^2$ et $\bar{y}_S = \frac{1}{|S|}\sum_{i \in S} y_i$.

Chaque feuille R_m prédit la moyenne :

$$\hat{f}(\boldsymbol{x}) = \sum_{m=1}^{M} \bar{y}_{R_m} \cdot \mathbb{1}[\boldsymbol{x} \in R_m] \tag{6.7}$$

6.5 Élagage

Un arbre complet (non élagué) a tendance à surapprendre. L'élagage (*pruning*) réduit la complexité pour améliorer la généralisation.

6.5.1 Pré-élagage

On impose des critères d'arrêt *avant* la construction :

- `max_depth` : profondeur maximale de l'arbre.

- `min_samples_split` : nombre minimal d'exemples pour autoriser une séparation.
- `min_samples_leaf` : nombre minimal d'exemples dans chaque feuille.
- `max_leaf_nodes` : nombre maximal de feuilles.

6.5.2 Post-élagage par coût-complexité

Définition 6.8 (Coût-complexité)**.** Pour un arbre T, on définit le critère de coût-complexité :

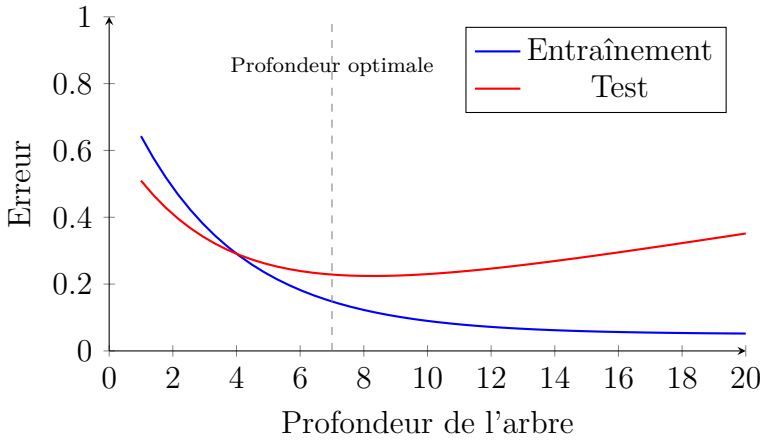
$$R_\alpha(T) = R(T) + \alpha \cdot |T| \tag{6.8}$$

où $R(T) = \sum_{m=1}^{|T|} \frac{|R_m|}{n} \cdot Q(R_m)$ est l'erreur de résubstitution, $|T|$ le nombre de feuilles, et $\alpha \geq 0$ le paramètre de complexité.

Théorème 6.9 (Élagage optimal)**.** *Pour chaque α, il existe un unique plus petit sous-arbre $T_\alpha \subseteq T_{\max}$ minimisant $R_\alpha(T)$. De plus, si $\alpha_1 < \alpha_2$, alors $T_{\alpha_2} \subseteq T_{\alpha_1}$.*

L'algorithme procède ainsi :

1. Construire l'arbre maximal $T_{\max}$.
2. Pour chaque nœud interne t, calculer $\alpha_{\text{eff}}(t) = \frac{R(t) - R(T_t)}{|T_t| - 1}$ où T_t est le sous-arbre enraciné en t.
3. Élaguer le nœud avec le plus petit α_{eff} (remplacer le sous-arbre par une feuille).
4. Répéter jusqu'à obtenir la racine seule. Cela produit une séquence d'arbres $T_{\max} \supset T_1 \supset \cdots \supset T_0$.
5. Choisir le meilleur α par validation croisée.



6.6 Importance des variables

Définition 6.10 (Importance MDI)**.** L'importance d'une variable j est la réduction totale d'impureté pondérée apportée par les séparations sur j :

$$\text{Imp}(j) = \sum_{t : \text{split sur } j} \frac{|S_t|}{n} \Delta Q(t) \tag{6.9}$$

normalisée pour que $\sum_j \text{Imp}(j) = 1$.

6.7 Implémentation avec scikit-learn

Arbre de décision sur le dataset Iris

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.model_selection import (
    train_test_split, cross_val_score, GridSearchCV
)
from sklearn.metrics import accuracy_score

# Charger les donnees
X, y = load_iris(return_X_y=True)
feature_names = load_iris().feature_names
target_names = load_iris().target_names

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
)

# Arbre sans contrainte (surapprentissage)
tree_full = DecisionTreeClassifier(random_state=42)
tree_full.fit(X_train, y_train)
print("=== Arbre complet ===")
print(f"Accuracy train : {tree_full.score(X_train, y_train):.4f}")
print(f"Accuracy test  : {tree_full.score(X_test, y_test):.4f}")
print(f"Profondeur     : {tree_full.get_depth()}")
print(f"Nb feuilles    : {tree_full.get_n_leaves()}")

# Arbre elague (pre-elagage)
tree_pruned = DecisionTreeClassifier(
    max_depth=3, min_samples_leaf=5, random_state=42
)
tree_pruned.fit(X_train, y_train)
print("\n=== Arbre elague ===")
print(f"Accuracy train : {tree_pruned.score(X_train, y_train):.4f}")
print(f"Accuracy test  : {tree_pruned.score(X_test, y_test):.4f}")
print(f"Profondeur     : {tree_pruned.get_depth()}")
print(f"Nb feuilles    : {tree_pruned.get_n_leaves()}")
```

Sortie

```
=== Arbre complet ===
Accuracy train : 1.0000
Accuracy test  : 0.9556
Profondeur     : 5
Nb feuilles    : 9
```

```
=== Arbre elague ===
Accuracy train : 0.9714
Accuracy test  : 0.9556
Profondeur     : 3
Nb feuilles    : 5
```

Visualisation de l'arbre et importance des variables

```python
# Visualisation de l'arbre
fig, ax = plt.subplots(figsize=(14, 8))
plot_tree(tree_pruned, feature_names=feature_names,
          class_names=target_names, filled=True, rounded=True,
          fontsize=9, ax=ax)
plt.title("Arbre de decision (Iris, max_depth=3)")
plt.tight_layout()
plt.savefig("arbre_iris.pdf")

# Importance des variables
importances = tree_pruned.feature_importances_
indices = np.argsort(importances)[::-1]

fig, ax = plt.subplots(figsize=(8, 4))
ax.barh(range(len(importances)), importances[indices], color="steelblue")
ax.set_yticks(range(len(importances)))
ax.set_yticklabels([feature_names[i] for i in indices])
ax.set_xlabel("Importance (MDI)")
ax.set_title("Importance des variables")
plt.tight_layout()
plt.savefig("arbre_importance.pdf")

print("Importance des variables :")
for i in indices:
    print(f"  {feature_names[i]:25s} : {importances[i]:.4f}")
```

Sortie

```
Importance des variables :
  petal width (cm)          : 0.5815
  petal length (cm)         : 0.4040
  sepal length (cm)         : 0.0146
  sepal width (cm)          : 0.0000
```

Post-élagage par coût-complexité

```python
# Chemin de cout-complexite
path = tree_full.cost_complexity_pruning_path(X_train, y_train)
ccp_alphas = path.ccp_alphas[:-1]  # exclure le dernier (arbre trivial)

# Entrainer un arbre pour chaque alpha
```

```
trees = []
for alpha in ccp_alphas:
    t = DecisionTreeClassifier(ccp_alpha=alpha, random_state=42)
    t.fit(X_train, y_train)
    trees.append(t)

# Scores
train_scores = [t.score(X_train, y_train) for t in trees]
test_scores = [t.score(X_test, y_test) for t in trees]

fig, ax = plt.subplots(figsize=(8, 5))
ax.plot(ccp_alphas, train_scores, "o-", label="Train", markersize=3)
ax.plot(ccp_alphas, test_scores, "o-", label="Test", markersize=3)
ax.set_xlabel(r"$\alpha$ (cout-complexite)")
ax.set_ylabel("Accuracy")
ax.legend()
ax.set_title("Elagage par cout-complexite")
plt.tight_layout()
plt.savefig("arbre_ccp.pdf")

best_idx = np.argmax(test_scores)
print(f"Meilleur alpha : {ccp_alphas[best_idx]:.4f}")
print(f"Accuracy test  : {test_scores[best_idx]:.4f}")
```

Sortie

```
Meilleur alpha : 0.0148
Accuracy test  : 0.9778
```

6.8 Frontière de décision

Visualisation de la frontière de décision

```
from sklearn.inspection import DecisionBoundaryDisplay

# Utiliser 2 features pour la visualisation
X2 = X[:, 2:4]  # petal length, petal width
X2_train, X2_test, y2_train, y2_test = train_test_split(
    X2, y, test_size=0.3, random_state=42, stratify=y
)

fig, axes = plt.subplots(1, 3, figsize=(15, 4))
for ax, depth in zip(axes, [1, 3, None]):
    tree = DecisionTreeClassifier(max_depth=depth, random_state=42)
    tree.fit(X2_train, y2_train)
    DecisionBoundaryDisplay.from_estimator(
        tree, X2_train, ax=ax, alpha=0.3, cmap=plt.cm.Set3
    )
    ax.scatter(X2_train[:, 0], X2_train[:, 1], c=y2_train,
```

```
                cmap=plt.cm.Set1, edgecolors="k", s=20)
    acc = tree.score(X2_test, y2_test)
    title = f"depth={depth}" if depth else "depth=infini"
    ax.set_title(f"{title} (acc={acc:.2f})")
    ax.set_xlabel("petal length")
    ax.set_ylabel("petal width")
plt.tight_layout()
plt.savefig("arbre_frontieres.pdf")
```

6.9 Exercices

Exercice 6.1 (Calcul d'entropie et de Gini — Niveau 1)**.** Un nœud contient 30 exemples : 10 de classe A, 15 de classe B, 5 de classe C.

1. Calculer l'entropie H et l'impureté de Gini G de ce nœud.
2. On propose la séparation suivante : S_L contient $(8, 2, 0)$ et S_R contient $(2, 13, 5)$. Calculer le gain d'information et la réduction de Gini.
3. Comparer avec la séparation $S_L = (5, 7, 3)$, $S_R = (5, 8, 2)$. Quelle séparation est préférable et pourquoi ?

Exercice 6.2 (Construction manuelle d'un arbre — Niveau 2)**.** Considérons le jeu de données suivant ($x_1, x_2 \in \mathbb{R}$, $y \in \{0, 1\}$) :

x_1	x_2	y
1	3	0
2	1	0
3	2	0
5	4	1
6	3	1
7	5	1
4	6	1
4	1	0

1. Construire un arbre CART (critère Gini) en détaillant chaque étape : énumérer tous les seuils candidats, calculer les impuretés, sélectionner la meilleure séparation.
2. Déterminer la profondeur minimale nécessaire pour classer correctement tous les exemples.
3. Vérifier votre résultat avec `DecisionTreeClassifier`.

Exercice 6.3 (Analyse théorique et pratique — Niveau 3)**.**

1. Démontrer que pour un problème binaire, l'impureté de Gini et l'entropie satisfont $G(p) \leq H(p)/\ln 2$ pour tout $p \in [0, 1]$. (*Indication* : utiliser $\ln x \leq x - 1$.)
2. Montrer que l'erreur de classification $1 - \max_k p_k$ n'est pas un bon critère de séparation en donnant un contre-exemple où elle ne différencie pas deux partitions de qualités différentes.

3. Sur le dataset `wine` de scikit-learn, comparer systématiquement les critères Gini et entropie en termes d'accuracy (validation croisée 10-fold), de profondeur et de nombre de feuilles, pour différentes valeurs de `max_depth`.

4. Implémenter un arbre de décision CART complet à partir de zéro (sans scikit-learn) pour la classification binaire. Comparer avec `DecisionTreeClassifier` sur le dataset `make_moons`.

Chapitre 7

Méthodes d'Ensemble — Bagging, Boosting, Random Forests

7.1 Pourquoi les méthodes d'ensemble ?

L'idée fondamentale des méthodes d'ensemble est de combiner plusieurs modèles « faibles » pour obtenir un modèle « fort ». Nous montrons ici formellement pourquoi cette stratégie réduit l'erreur.

Théorème 7.1 (Réduction de variance par moyennage)**.** *Soient $f_1, \dots, f_B$ des modèles indépendants et identiquement distribués, chacun de variance σ^2 et de biais μ. Le modèle moyenné $\bar{f} = \frac{1}{B}\sum_{b=1}^{B} f_b$ satisfait :*

$$\mathrm{Bias}[\bar{f}] = \mu, \qquad \mathrm{Var}[\bar{f}] = \frac{\sigma^2}{B} \tag{7.1}$$

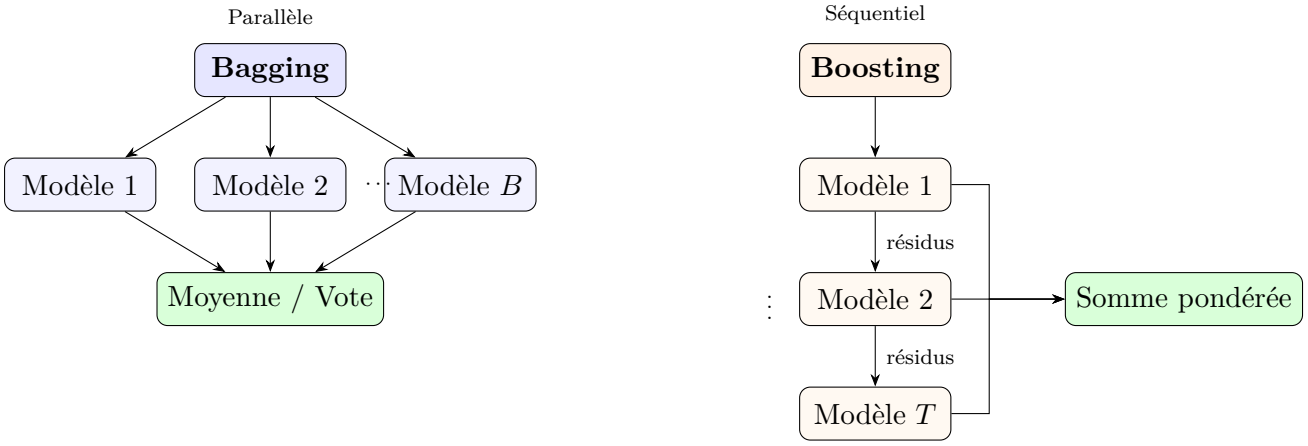
Le biais est inchangé mais la variance est divisée par B.

Démonstration. Par linéarité de l'espérance : $\mathbb{E}[\bar{f}] = \frac{1}{B}\sum_b \mathbb{E}[f_b] = \mu$. Par indépendance : $\mathrm{Var}[\bar{f}] = \frac{1}{B^2}\sum_b \mathrm{Var}[f_b] = \frac{\sigma^2}{B}$. □

Remarque 7.2. En pratique, les modèles ne sont pas parfaitement indépendants car ils sont entraînés sur des échantillons liés. Si la corrélation entre paires est ρ, alors :

$$\mathrm{Var}[\bar{f}] = \rho\sigma^2 + \frac{1-\rho}{B}\sigma^2 \tag{7.2}$$

Réduire ρ est la clé pour améliorer les ensembles.



7.2 Bagging (Bootstrap Aggregating)

Définition 7.3 (Bagging)**.** Le **bagging** [2] consiste à :

1. Générer B échantillons bootstrap $\mathcal{D}_1^*, \ldots, \mathcal{D}_B^*$ par tirage avec remise de taille n.
2. Entraîner un modèle f_b sur chaque $\mathcal{D}_b^*$.
3. Agréger les prédictions :
   - Régression : $\hat{f}(\boldsymbol{x}) = \frac{1}{B}\sum_{b=1}^{B} f_b(\boldsymbol{x})$
   - Classification : $\hat{f}(\boldsymbol{x}) = \arg\max_k \sum_{b=1}^{B} \mathbb{1}[f_b(\boldsymbol{x}) = k]$ (vote majoritaire)

Proposition 7.4 (Proportion d'exemples hors-sac)**.** Un échantillon bootstrap de taille n contient en moyenne environ $1-(1-1/n)^n \approx 1 - e^{-1} \approx 63.2\,\%$ des exemples originaux. Les $\sim 36.8\,\%$ restants sont dits « hors-sac » (*out-of-bag*).

Démonstration. La probabilité qu'un exemple i ne soit pas sélectionné dans un tirage est $1-1/n$. Sur n tirages indépendants : $\Pr(i \notin \mathcal{D}_b^*) = (1-1/n)^n \xrightarrow[n\to\infty]{} e^{-1}$. □

7.2.1 Dérivation formelle de la réduction de variance

Théorème 7.5 (Variance du bagging)**.** *Si les modèles f_b ont une variance σ^2 et une corrélation par paires ρ, alors la prédiction bagée a pour variance :*

$$\mathrm{Var}[\hat{f}_{bag}] = \rho\sigma^2 + \frac{(1-\rho)\sigma^2}{B} \tag{7.3}$$

Pour $B \to \infty$, $\mathrm{Var}[\hat{f}_{bag}] \to \rho\sigma^2$.

Démonstration. On a $\hat{f}_{\text{bag}} = \frac{1}{B}\sum_b f_b$. Donc :

$$\mathrm{Var}[\hat{f}_{\text{bag}}] = \frac{1}{B^2}\left[\sum_b \mathrm{Var}[f_b] + \sum_{b\neq b'} \mathrm{Cov}(f_b, f_{b'})\right] \tag{7.4}$$

$$= \frac{1}{B^2}\left[B\sigma^2 + B(B-1)\rho\sigma^2\right] \tag{7.5}$$

$$= \frac{\sigma^2}{B} + \frac{(B-1)\rho\sigma^2}{B} = \rho\sigma^2 + \frac{(1-\rho)\sigma^2}{B} \tag{7.6}$$

□

7.3 Forêts aléatoires

Définition 7.6 (Forêt aléatoire)**.** Une **forêt aléatoire** [2] est un bagging d'arbres de décision avec une randomisation supplémentaire : à chaque séparation, seul un sous-ensemble aléatoire de m variables (sur d) est considéré.

Choix de m (nombre de variables candidates)

Tâche	Valeur recommandée
Classification	$m = \lfloor \sqrt{d} \rfloor$
Régression	$m = \lfloor d/3 \rfloor$

Réduire m diminue la corrélation ρ entre arbres, ce qui réduit la variance de l'ensemble (cf. équation (7.2)).

Algorithme Random Forest

Entrée : $\mathcal{D} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$, nombre d'arbres B, nombre de variables m.
Pour $b = 1, \ldots, B$:

1. Tirer un échantillon bootstrap $\mathcal{D}_b^*$ de taille n.
2. Construire un arbre T_b sur $\mathcal{D}_b^*$:
   - À chaque nœud, sélectionner aléatoirement m variables.
   - Trouver la meilleure séparation parmi ces m variables.
   - Développer l'arbre complètement (pas d'élagage).

Prédiction : agréger par vote (classification) ou moyenne (régression).

7.3.1 Erreur out-of-bag (OOB)

Définition 7.7 (Erreur OOB)**.** Pour chaque observation $(\boldsymbol{x}_i, y_i)$, l'erreur **out-of-bag** est calculée en agrégeant uniquement les arbres pour lesquels i n'était pas dans l'échantillon bootstrap :

$$\hat{y}_i^{\text{OOB}} = \text{agréger}\{T_b(\boldsymbol{x}_i) : i \notin \mathcal{D}_b^*\} \tag{7.7}$$

L'erreur OOB globale est une estimation non biaisée de l'erreur de généralisation, sans nécessiter d'ensemble de validation.

Avantages des forêts aléatoires

- Peu d'hyperparamètres à régler (principalement B et m).
- Robustes au surapprentissage : augmenter B ne dégrade jamais la performance (la variance diminue ou stagne).
- Fournissent une estimation de l'erreur (OOB) gratuitement.
- Calculent l'importance des variables.
- Parallélisables efficacement.

7.4 Boosting

Contrairement au bagging (réduction de variance), le boosting vise à réduire le **biais** en combinant séquentiellement des modèles faibles.

7.4.1 AdaBoost

Algorithme AdaBoost

Entrée : $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$ avec $y_i \in \{-1, +1\}$, nombre d'itérations T.
Initialisation : $w_i^{(1)} = 1/n$ pour tout i.
Pour $t = 1, \ldots, T$:

1. Entraîner un classifieur faible h_t en minimisant l'erreur pondérée :

$$\varepsilon_t = \sum_{i=1}^{n} w_i^{(t)} \mathbb{1}[h_t(\boldsymbol{x}_i) \neq y_i] \tag{7.8}$$

2. Calculer le poids du classifieur :

$$\alpha_t = \frac{1}{2} \ln\left(\frac{1 - \varepsilon_t}{\varepsilon_t}\right) \tag{7.9}$$

3. Mettre à jour les poids des exemples :

$$w_i^{(t+1)} = \frac{w_i^{(t)} \exp(-\alpha_t y_i h_t(\boldsymbol{x}_i))}{Z_t} \tag{7.10}$$

où Z_t est le facteur de normalisation.

Prédiction : $H(\boldsymbol{x}) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(\boldsymbol{x})\right)$

Théorème 7.8 (Dérivation de la règle de mise à jour)**.** *AdaBoost minimise la perte exponentielle :*

$$L = \sum_{i=1}^{n} \exp\left(-y_i \sum_{t=1}^{T} \alpha_t h_t(\boldsymbol{x}_i)\right) \tag{7.11}$$

À l'étape t, en fixant les classifieurs précédents et en optimisant α_t :

$$\frac{\partial L}{\partial \alpha_t} = -\sum_{i=1}^{n} y_i h_t(\boldsymbol{x}_i) w_i^{(t)} e^{-\alpha_t y_i h_t(\boldsymbol{x}_i)} = 0 \tag{7.12}$$

En séparant les exemples correctement classés ($y_i h_t = +1$) et incorrectement classés ($y_i h_t = -1$) :

$$e^{-\alpha_t}(1 - \varepsilon_t) - e^{\alpha_t}\varepsilon_t = 0 \implies \alpha_t = \frac{1}{2} \ln\left(\frac{1 - \varepsilon_t}{\varepsilon_t}\right) \tag{7.13}$$

Proposition 7.9 (Borne sur l'erreur d'entraînement)**.** L'erreur d'entraînement d'Ada-

Boost est bornée par :

$$\frac{1}{n}\sum_{i=1}^{n} \mathbb{1}[H(\boldsymbol{x}_i) \neq y_i] \leq \prod_{t=1}^{T} 2\sqrt{\varepsilon_t(1-\varepsilon_t)} = \prod_{t=1}^{T} \sqrt{1-4\gamma_t^2} \tag{7.14}$$

où $\gamma_t = \frac{1}{2} - \varepsilon_t$ est l'« avantage » du classifieur faible. Si $\gamma_t \geq \gamma > 0$, l'erreur décroît exponentiellement en T.

AdaBoost et le bruit

AdaBoost augmente les poids des exemples mal classés. En présence de bruit (étiquettes erronées), il sur-pondère les outliers, ce qui peut dégrader la généralisation. Le Gradient Boosting avec régularisation est généralement plus robuste.

7.4.2 Gradient Boosting

Définition 7.10 (Gradient Boosting)**.** Le **gradient boosting** généralise le boosting à toute fonction de perte différentiable ℓ. Le modèle est construit de façon additive :

$$F_t(\boldsymbol{x}) = F_{t-1}(\boldsymbol{x}) + \eta \cdot h_t(\boldsymbol{x}) \tag{7.15}$$

où h_t est ajusté sur les **pseudo-résidus** (gradient négatif) :

$$r_i^{(t)} = -\frac{\partial \ell(y_i, F_{t-1}(\boldsymbol{x}_i))}{\partial F_{t-1}(\boldsymbol{x}_i)} \tag{7.16}$$

et $\eta \in (0, 1]$ est le taux d'apprentissage (*learning rate*).

Algorithme Gradient Boosting

Entrée : $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$, perte ℓ, nombre d'itérations T, taux η.

1. Initialiser $F_0(\boldsymbol{x}) = \arg\min_c \sum_{i=1}^n \ell(y_i, c)$.
2. **Pour** $t = 1, \ldots, T$:
   (a) Calculer les pseudo-résidus $r_i^{(t)}$ (équation (7.16)).
   (b) Ajuster un arbre de régression h_t sur $\{(\boldsymbol{x}_i, r_i^{(t)})\}$.
   (c) Mettre à jour : $F_t = F_{t-1} + \eta \cdot h_t$.

Prédiction : $\hat{y} = F_T(\boldsymbol{x})$.

Descente de gradient dans l'espace fonctionnel

Le gradient boosting effectue une descente de gradient, non pas dans l'espace des paramètres, mais dans l'**espace des fonctions**. À chaque itération, h_t approxime la direction de plus forte descente de la perte. Le taux η joue le rôle du pas de gradient et régularise le modèle : un η petit nécessite plus d'arbres mais généralise mieux.

7.4.3 XGBoost et LightGBM

Les implémentations modernes ajoutent plusieurs améliorations :

Améliorations de XGBoost/LightGBM

Technique	Description
Régularisation L1/L2	Pénalisation des poids des feuilles : $\Omega(h) = \gamma\|T\| + \frac{\lambda}{2}\sum_j w_j^2$
Approximation de Newton	Utilisation du hessien (dérivée seconde) pour des séparations plus précises
Sous-échantillonnage	Colonnes et lignes échantillonnées à chaque arbre (réduit ρ)
Croissance par feuille	LightGBM fait croître l'arbre par feuille (*leaf-wise*) au lieu de par niveau (*level-wise*)
Histogrammes	Discrétisation des features pour accélérer la recherche de seuils

7.5 Comparaison des méthodes

Tableau comparatif

	Bagging/RF	AdaBoost	Gradient Boosting
Réduit	Variance	Biais	Biais
Parallèle	Oui	Non	Non
Robuste au bruit	Oui	Non	Modéré
Risque surapprentissage	Faible	Modéré	Élevé
Hyperparamètres	Peu	Peu	Beaucoup

7.6 Implémentations avec scikit-learn

Random Forest

```
import numpy as np
from sklearn.datasets import load_wine
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split, cross_val_score

X, y = load_wine(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
)

rf = RandomForestClassifier(
    n_estimators=200, max_features="sqrt", oob_score=True,
```

```
    random_state=42, n_jobs=-1
)
rf.fit(X_train, y_train)

print("=== Random Forest ===")
print(f"Accuracy train : {rf.score(X_train, y_train):.4f}")
print(f"Accuracy test  : {rf.score(X_test, y_test):.4f}")
print(f"Erreur OOB     : {1 - rf.oob_score_:.4f}")
print(f"CV accuracy    : {cross_val_score(rf, X, y, cv=5).mean():.4f}")

# Importance des variables
importances = rf.feature_importances_
feature_names = load_wine().feature_names
indices = np.argsort(importances)[::-1]
print("\nTop 5 variables :")
for i in range(5):
    print(f"  {feature_names[indices[i]]:25s} :
    ↪  {importances[indices[i]]:.4f}")
```

Sortie

```
=== Random Forest ===
Accuracy train : 1.0000
Accuracy test  : 0.9815
Erreur OOB     : 0.0323
CV accuracy    : 0.9775

Top 5 variables :
  color_intensity           : 0.1542
  flavanoids                : 0.1487
  proline                   : 0.1325
  alcohol                   : 0.1198
  od280/od315_of_diluted_wines : 0.0843
```

Gradient Boosting et comparaison

```
from sklearn.ensemble import (
    GradientBoostingClassifier, AdaBoostClassifier
)
from sklearn.tree import DecisionTreeClassifier
import matplotlib.pyplot as plt

# AdaBoost
ada = AdaBoostClassifier(
    estimator=DecisionTreeClassifier(max_depth=1),
    n_estimators=200, learning_rate=0.5, random_state=42
)
ada.fit(X_train, y_train)
```

```python
# Gradient Boosting
gb = GradientBoostingClassifier(
    n_estimators=200, max_depth=3, learning_rate=0.1,
    subsample=0.8, random_state=42
)
gb.fit(X_train, y_train)

print("=== Comparaison ===")
models = {"Random Forest": rf, "AdaBoost": ada, "Gradient Boosting": gb}
for name, model in models.items():
    acc_train = model.score(X_train, y_train)
    acc_test = model.score(X_test, y_test)
    cv = cross_val_score(model, X, y, cv=5).mean()
    print(f"{name:20s} | train={acc_train:.4f} | "
          f"test={acc_test:.4f} | CV={cv:.4f}")
```

Sortie

```
=== Comparaison ===
Random Forest        | train=1.0000 | test=0.9815 | CV=0.9775
AdaBoost             | train=1.0000 | test=0.9444 | CV=0.9497
Gradient Boosting    | train=1.0000 | test=0.9815 | CV=0.9719
```

Courbes d'apprentissage et effet du nombre d'arbres

```python
# Erreur en fonction du nombre d'arbres (staged_predict)
n_estimators_range = range(1, 201)

fig, axes = plt.subplots(1, 2, figsize=(12, 5))

# AdaBoost
ada_train = [1 - acc for acc in ada.staged_score(X_train, y_train)]
ada_test = [1 - acc for acc in ada.staged_score(X_test, y_test)]
axes[0].plot(n_estimators_range, ada_train, label="Train")
axes[0].plot(n_estimators_range, ada_test, label="Test")
axes[0].set_title("AdaBoost")
axes[0].set_xlabel("Nombre d'arbres")
axes[0].set_ylabel("Erreur")
axes[0].legend()

# Gradient Boosting
gb_train = [1 - acc for acc in gb.staged_score(X_train, y_train)]
gb_test = [1 - acc for acc in gb.staged_score(X_test, y_test)]
axes[1].plot(n_estimators_range, gb_train, label="Train")
axes[1].plot(n_estimators_range, gb_test, label="Test")
axes[1].set_title("Gradient Boosting")
axes[1].set_xlabel("Nombre d'arbres")
axes[1].set_ylabel("Erreur")
axes[1].legend()
```

```
plt.tight_layout()
plt.savefig("ensemble_learning_curves.pdf")
```

XGBoost (si disponible)

```
try:
    from xgboost import XGBClassifier

    xgb = XGBClassifier(
        n_estimators=200, max_depth=4, learning_rate=0.1,
        subsample=0.8, colsample_bytree=0.8,
        reg_lambda=1.0, reg_alpha=0.1,
        eval_metric="mlogloss", random_state=42
    )
    xgb.fit(X_train, y_train)
    print(f"XGBoost test accuracy : {xgb.score(X_test, y_test):.4f}")
    print(f"XGBoost CV accuracy   : "
          f"{cross_val_score(xgb, X, y, cv=5).mean():.4f}")
except ImportError:
    print("xgboost non installe. Installer avec: pip install xgboost")
```

Sortie

```
XGBoost test accuracy : 0.9815
XGBoost CV accuracy   : 0.9775
```

7.7 Exercices

Exercice 7.1 (Bootstrap et erreur OOB — Niveau 1)**.**

1. Montrer que la probabilité qu'un exemple donné soit absent d'un échantillon bootstrap de taille n converge vers e^{-1} quand $n \to \infty$.
2. Pour $n = 100$, calculer la valeur exacte $(1 - 1/100)^{100}$ et la comparer à e^{-1}.
3. Entraîner une forêt aléatoire sur le dataset Iris avec `oob_score=True`. Comparer l'erreur OOB avec l'erreur de validation croisée 5-fold.

Exercice 7.2 (AdaBoost analytique — Niveau 2)**.** Soit un jeu de données de 6 points équirépartis entre les classes $+1$ et -1. À la première itération, le classifieur faible h_1 classe correctement 4 points sur 6.

1. Calculer ε_1, α_1 et les nouveaux poids $w_i^{(2)}$.
2. Si le deuxième classifieur faible h_2 classe correctement les 2 points mal classés par h_1 mais en rate 1 autre, calculer ε_2 et α_2.
3. Démontrer complètement que la formule (7.9) minimise la perte exponentielle à chaque étape.

4. Vérifier que la borne sur l'erreur d'entraînement est satisfaite après 2 itérations.

Exercice 7.3 (Comparaison complète — Niveau 3)**.**

1. Sur le dataset `breast_cancer` de scikit-learn, comparer systématiquement : arbre seul, bagging (50 arbres), forêt aléatoire (50, 200, 500 arbres), AdaBoost (100 stumps), Gradient Boosting (100 arbres, $\eta = 0.1$). Reporter accuracy, F1-score et temps d'entraînement (validation croisée 10-fold).

2. Étudier l'effet du nombre de variables candidates m dans une forêt aléatoire. Tracer l'erreur OOB en fonction de m pour $m \in \{1, 2, \ldots, d\}$.

3. Implémenter l'algorithme AdaBoost à partir de zéro (sans scikit-learn) avec des *decision stumps* comme classifieurs faibles. Comparer avec `AdaBoostClassifier`.

4. Démontrer formellement que le bagging ne peut pas augmenter le biais par rapport au modèle de base (pour la perte quadratique), en utilisant l'inégalité de Jensen.

Chapitre 8

Apprentissage non supervisé — Clustering

8.1 Introduction

En apprentissage non supervisé, on dispose d'un jeu de données $\mathcal{D} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$ avec $\boldsymbol{x}_i \in \mathbb{R}^d$, *sans étiquettes*. L'objectif du **clustering** (partitionnement) est de regrouper les observations en K groupes $\mathcal{C} = \{C_1, \ldots, C_K\}$ tels que les points d'un même cluster soient « similaires » et ceux de clusters différents soient « dissemblables ».

Définition 8.1 (Partition)**.** Une **partition** de $\mathcal{D}$ en K clusters est une famille $\{C_1, \ldots, C_K\}$ telle que :

1. $C_k \neq \emptyset$ pour tout k,
2. $C_i \cap C_j = \emptyset$ pour $i \neq j$,
3. $\bigcup_{k=1}^{K} C_k = \mathcal{D}$.

8.2 k-Means

8.2.1 Formulation

Définition 8.2 (k-Means)**.** L'algorithme k-Means cherche une partition $\{C_1, \ldots, C_K\}$ et des centroïdes $\{\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_K\}$ minimisant l'**inertie intra-cluster** :

$$J(\mathcal{C}, \boldsymbol{\mu}) = \sum_{k=1}^{K} \sum_{\boldsymbol{x}_i \in C_k} \|\boldsymbol{x}_i - \boldsymbol{\mu}_k\|^2 \tag{8.1}$$

Ce problème est NP-difficile en général. L'algorithme de Lloyd fournit une heuristique efficace.

8.2.2 Algorithme de Lloyd

k-Means (Lloyd)

Entrée : $\mathcal{D} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$, nombre de clusters K.

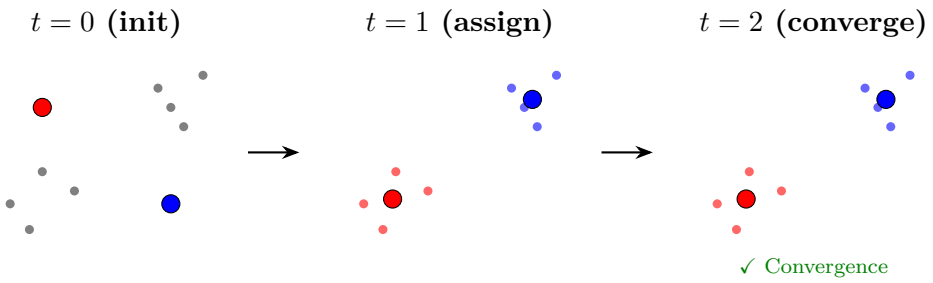
1. **Initialisation :** choisir K centroïdes initiaux $\boldsymbol{\mu}_1^{(0)}, \ldots, \boldsymbol{\mu}_K^{(0)}$.
2. **Répéter** jusqu'à convergence :
   (a) **Assignation :** pour chaque $\boldsymbol{x}_i$, affecter au cluster le plus proche :

$$c_i^{(t)} = \arg\min_{k \in \{1,\ldots,K\}} \left\| \boldsymbol{x}_i - \boldsymbol{\mu}_k^{(t)} \right\|^2 \tag{8.2}$$

   (b) **Mise à jour :** recalculer les centroïdes :

$$\boldsymbol{\mu}_k^{(t+1)} = \frac{1}{|C_k^{(t)}|} \sum_{\boldsymbol{x}_i \in C_k^{(t)}} \boldsymbol{x}_i \tag{8.3}$$

Sortie : partition $\{C_1, \ldots, C_K\}$ et centroïdes $\{\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_K\}$.



8.2.3 Convergence

Théorème 8.3 (Convergence de k-Means)**.** *L'algorithme de Lloyd converge en un nombre fini d'itérations.*

Esquisse de preuve. À chaque itération, l'étape d'assignation et l'étape de mise à jour diminuent (ou laissent inchangée) la fonction objectif J.

- **Assignation :** à $\boldsymbol{\mu}$ fixé, assigner chaque point au centroïde le plus proche minimise J.
- **Mise à jour :** à $\mathcal{C}$ fixé, le minimum de $\sum_{\boldsymbol{x}_i \in C_k} \|\boldsymbol{x}_i - \boldsymbol{\mu}_k\|^2$ par rapport à $\boldsymbol{\mu}_k$ est atteint en $\boldsymbol{\mu}_k = \frac{1}{|C_k|} \sum_{\boldsymbol{x}_i \in C_k} \boldsymbol{x}_i$ (dérivée nulle).

Comme $J \geq 0$ est décroissante et que le nombre de partitions est fini ($\leq K^n$), l'algorithme converge. Néanmoins, il converge vers un *minimum local*, pas nécessairement global. □

8.2.4 Initialisation : k-Means++

Sensibilité à l'initialisation

L'algorithme de Lloyd est très sensible au choix initial des centroïdes. Une mauvaise initialisation peut conduire à un minimum local de mauvaise qualité. La stratégie **k-Means++** atténue ce problème.

Initialisation k-Means++

1. Choisir $\boldsymbol{\mu}_1$ uniformément au hasard parmi les données.
2. Pour $k = 2, \dots, K$:
   (a) Pour chaque $\boldsymbol{x}_i$, calculer $d_i = \min_{j<k} \|\boldsymbol{x}_i - \boldsymbol{\mu}_j\|^2$.
   (b) Choisir $\boldsymbol{\mu}_k = \boldsymbol{x}_i$ avec probabilité $\frac{d_i}{\sum_j d_j}$.
3. Lancer l'algorithme de Lloyd avec ces centroïdes.

Théorème 8.4 (Garantie k-Means++ [11])**.** *L'initialisation k-Means++ garantit que l'inertie attendue satisfait :*

$$\mathbb{E}[J] \leq 8(\ln K + 2) \cdot J_{opt} \tag{8.4}$$

où J_{opt} est l'inertie optimale.

8.3 Modèles de mélange gaussien (GMM)

8.3.1 Modèle

Définition 8.5 (Mélange de gaussiennes)**.** Un **modèle de mélange gaussien** à K composantes suppose que chaque observation est générée par :

$$p(\boldsymbol{x}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \tag{8.5}$$

où $\pi_k \geq 0$ sont les poids de mélange ($\sum_k \pi_k = 1$), $\boldsymbol{\mu}_k$ les moyennes, et $\boldsymbol{\Sigma}_k$ les matrices de covariance.

On introduit une variable latente $z_i \in \{1, \dots, K\}$ indiquant la composante de chaque observation. Alors :

$$p(z_i = k) = \pi_k, \qquad p(\boldsymbol{x}_i \mid z_i = k) = \mathcal{N}(\boldsymbol{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \tag{8.6}$$

8.3.2 Algorithme EM

La log-vraisemblance du mélange est :

$$\ell(\boldsymbol{\theta}) = \sum_{i=1}^{n} \ln \left(\sum_{k=1}^{K} \pi_k \mathcal{N}(\boldsymbol{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right) \tag{8.7}$$

La présence du logarithme d'une somme rend l'optimisation directe difficile. L'algorithme **Espérance-Maximisation (EM)** résout ce problème de manière itérative.

Algorithme EM pour GMM

E-step : calculer les responsabilités :

$$\gamma_{ik} = p(z_i = k \mid \boldsymbol{x}_i, \boldsymbol{\theta}^{(t)}) = \frac{\pi_k^{(t)} \mathcal{N}(\boldsymbol{x}_i \mid \boldsymbol{\mu}_k^{(t)}, \boldsymbol{\Sigma}_k^{(t)})}{\sum_{j=1}^{K} \pi_j^{(t)} \mathcal{N}(\boldsymbol{x}_i \mid \boldsymbol{\mu}_j^{(t)}, \boldsymbol{\Sigma}_j^{(t)})} \tag{8.8}$$

M-step : mettre à jour les paramètres. Soit $N_k = \sum_{i=1}^{n} \gamma_{ik}$:

$$\boldsymbol{\mu}_k^{(t+1)} = \frac{1}{N_k} \sum_{i=1}^{n} \gamma_{ik}\, \boldsymbol{x}_i \tag{8.9}$$

$$\boldsymbol{\Sigma}_k^{(t+1)} = \frac{1}{N_k} \sum_{i=1}^{n} \gamma_{ik}\, (\boldsymbol{x}_i - \boldsymbol{\mu}_k^{(t+1)})(\boldsymbol{x}_i - \boldsymbol{\mu}_k^{(t+1)})^\top \tag{8.10}$$

$$\pi_k^{(t+1)} = \frac{N_k}{n} \tag{8.11}$$

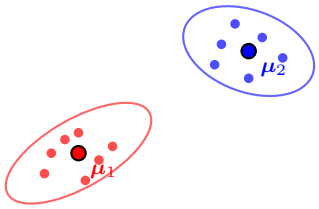
Théorème 8.6 (Monotonie de EM)**.** *L'algorithme EM garantit que la log-vraisemblance est non décroissante :*

$$\ell(\boldsymbol{\theta}^{(t+1)}) \geq \ell(\boldsymbol{\theta}^{(t)}) \tag{8.12}$$

La preuve repose sur l'inégalité de Jensen et la construction de la borne inférieure (ELBO).

EM vs k-Means

- k-Means est un cas particulier de EM où toutes les covariances sont $\sigma^2 \boldsymbol{I}$ et les responsabilités sont « dures » ($\gamma_{ik} \in \{0, 1\}$).
- EM effectue une assignation « souple » (*soft clustering*).
- EM modélise des clusters elliptiques, pas seulement sphériques.



Ellipses de covariance du GMM

8.4 DBSCAN

Définition 8.7 (DBSCAN)**. DBSCAN** (*Density-Based Spatial Clustering of Applications with Noise*) est un algorithme fondé sur la densité, paramétré par :

- $\varepsilon > 0$: rayon du voisinage,
- `min_samples` : nombre minimum de points dans le ε-voisinage pour qu'un point soit « core ».

Définition 8.8 (ε-voisinage)**.** Le ε-voisinage d'un point $\boldsymbol{x}$ est :

$$N_\varepsilon(\boldsymbol{x}) = \{\boldsymbol{x}' \in \mathcal{D} : \|\boldsymbol{x} - \boldsymbol{x}'\| \leq \varepsilon\} \tag{8.13}$$

Définition 8.9 (Types de points)**.**

- **Point noyau** (*core point*) : $|N_\varepsilon(\boldsymbol{x})| \geq$ `min_samples`.
- **Point frontière** (*border point*) : non noyau mais dans le ε-voisinage d'un point noyau.
- **Point bruit** (*noise*) : ni noyau, ni frontière.

DBSCAN

1. Marquer tous les points comme « non visités ».
2. Pour chaque point non visité $\boldsymbol{x}$:
   (a) Marquer $\boldsymbol{x}$ comme visité.
   (b) Si $|N_\varepsilon(\boldsymbol{x})| <$ `min_samples`, marquer comme bruit (provisoirement).
   (c) Sinon, créer un nouveau cluster C et y ajouter $\boldsymbol{x}$. Pour chaque $\boldsymbol{x}' \in N_\varepsilon(\boldsymbol{x})$, si $\boldsymbol{x}'$ n'est pas visité, le visiter et étendre le cluster si c'est un point noyau.

Avantages de DBSCAN

- Pas besoin de spécifier K a priori.
- Détecte des clusters de forme arbitraire.
- Identifie naturellement les points aberrants (bruit).

8.5 Clustering hiérarchique

8.5.1 Approche agglomérative

Clustering agglomératif

1. Initialiser : chaque point est un cluster singleton.
2. **Répéter** jusqu'à obtenir un seul cluster :
   (a) Trouver les deux clusters les plus proches selon un critère de liaison (*linkage*).
   (b) Fusionner ces deux clusters.

Critères de liaison

Soient deux clusters C_a et C_b :

$$\text{Single linkage :} \quad d(C_a, C_b) = \min_{\boldsymbol{x} \in C_a,\, \boldsymbol{x}' \in C_b} \|\boldsymbol{x} - \boldsymbol{x}'\| \tag{8.14}$$

$$\text{Complete linkage :} \quad d(C_a, C_b) = \max_{\boldsymbol{x} \in C_a,\, \boldsymbol{x}' \in C_b} \|\boldsymbol{x} - \boldsymbol{x}'\| \tag{8.15}$$

$$\text{Average linkage :} \quad d(C_a, C_b) = \frac{1}{|C_a||C_b|} \sum_{\boldsymbol{x} \in C_a} \sum_{\boldsymbol{x}' \in C_b} \|\boldsymbol{x} - \boldsymbol{x}'\| \tag{8.16}$$

$$\text{Ward :} \quad d(C_a, C_b) = \frac{|C_a||C_b|}{|C_a| + |C_b|} \|\boldsymbol{\mu}_a - \boldsymbol{\mu}_b\|^2 \tag{8.17}$$

8.5.2 Dendrogramme

Le dendrogramme est une représentation arborescente de la hiérarchie de fusions. On coupe à une hauteur choisie pour obtenir le nombre de clusters désiré.

Remarque 8.10. Le critère de Ward tend à produire des clusters de tailles équilibrées et est souvent un bon choix par défaut. Le *single linkage* est sensible à l'« effet de chaîne ».

8.6 Indices de validité

Définition 8.11 (Score silhouette)**.** Pour un point $\boldsymbol{x}_i$ appartenant au cluster C_k :

$$s_i = \frac{b_i - a_i}{\max(a_i, b_i)} \tag{8.18}$$

où $a_i = \frac{1}{|C_k|-1} \sum_{\boldsymbol{x}_j \in C_k,\, j \neq i} \|\boldsymbol{x}_i - \boldsymbol{x}_j\|$ est la distance intra-cluster moyenne, et $b_i = \min_{l \neq k} \frac{1}{|C_l|} \sum_{\boldsymbol{x}_j \in C_l} \|\boldsymbol{x}_i -$ est la distance inter-cluster minimale moyenne. Le score silhouette global est $\bar{s} = \frac{1}{n} \sum_{i=1}^{n} s_i \in [-1, 1]$.

Définition 8.12 (Indice de Calinski-Harabasz)**.**

$$\text{CH} = \frac{\text{SS}_B / (K-1)}{\text{SS}_W / (n-K)} \tag{8.19}$$

où $\text{SS}_B = \sum_{k=1}^{K} |C_k| \, \|\boldsymbol{\mu}_k - \boldsymbol{\mu}\|^2$ est la variance inter-cluster et $\text{SS}_W = \sum_{k=1}^{K} \sum_{\boldsymbol{x}_i \in C_k} \|\boldsymbol{x}_i - \boldsymbol{\mu}_k\|^2$ la variance intra-cluster. Un indice élevé indique des clusters bien séparés.

8.7 Implémentation en Python

k-Means avec scikit-learn

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
```

```python
# Generer des donnees synthetiques
X, y_true = make_blobs(n_samples=300, centers=4,
                       cluster_std=0.8, random_state=42)

# Methode du coude (Elbow method)
inertias = []
sil_scores = []
K_range = range(2, 10)
for k in K_range:
    km = KMeans(n_clusters=k, init='k-means++',
                n_init=10, random_state=42)
    km.fit(X)
    inertias.append(km.inertia_)
    sil_scores.append(silhouette_score(X, km.labels_))

# Modele final
km = KMeans(n_clusters=4, init='k-means++',
            n_init=10, random_state=42)
labels = km.fit_predict(X)

fig, axes = plt.subplots(1, 3, figsize=(15, 4))
axes[0].plot(K_range, inertias, 'bo-')
axes[0].set_xlabel('K'); axes[0].set_ylabel('Inertie')
axes[0].set_title('Methode du coude')

axes[1].plot(K_range, sil_scores, 'ro-')
axes[1].set_xlabel('K'); axes[1].set_ylabel('Silhouette')
axes[1].set_title('Score silhouette')

axes[2].scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis', s=15)
axes[2].scatter(km.cluster_centers_[:, 0],
                km.cluster_centers_[:, 1],
                c='red', marker='X', s=200, edgecolors='k')
axes[2].set_title('Clusters k-Means (K=4)')
plt.tight_layout()
plt.savefig('kmeans_result.pdf')
```

GMM avec scikit-learn

```python
from sklearn.mixture import GaussianMixture

gmm = GaussianMixture(n_components=4, covariance_type='full',
                      n_init=5, random_state=42)
gmm.fit(X)
labels_gmm = gmm.predict(X)
probs = gmm.predict_proba(X)  # responsabilites gamma_ik

print(f"BIC: {gmm.bic(X):.1f}")
print(f"AIC: {gmm.aic(X):.1f}")
```

```
print(f"Poids: {gmm.weights_.round(3)}")

# Selection du nombre de composantes par BIC
bics = []
for k in range(1, 10):
    g = GaussianMixture(n_components=k, random_state=42)
    g.fit(X)
    bics.append(g.bic(X))

plt.figure()
plt.plot(range(1, 10), bics, 'go-')
plt.xlabel('K'); plt.ylabel('BIC')
plt.title('Selection par BIC')
plt.savefig('gmm_bic.pdf')
```

DBSCAN et clustering hiérarchique

```
from sklearn.cluster import DBSCAN, AgglomerativeClustering
from sklearn.datasets import make_moons
from scipy.cluster.hierarchy import dendrogram, linkage

# Donnees non convexes
X_moons, _ = make_moons(n_samples=300, noise=0.05,
                        random_state=42)

# DBSCAN
db = DBSCAN(eps=0.2, min_samples=5)
labels_db = db.fit_predict(X_moons)
n_clusters = len(set(labels_db) - {-1})
n_noise = (labels_db == -1).sum()
print(f"DBSCAN: {n_clusters} clusters, {n_noise} bruit")

# Clustering hierarchique
agg = AgglomerativeClustering(n_clusters=2, linkage='ward')
labels_agg = agg.fit_predict(X_moons)

# Dendrogramme
Z = linkage(X_moons[:50], method='ward')  # sous-echantillon
fig, axes = plt.subplots(1, 3, figsize=(15, 4))
axes[0].scatter(X_moons[:, 0], X_moons[:, 1],
                c=labels_db, cmap='viridis', s=15)
axes[0].set_title('DBSCAN')

axes[1].scatter(X_moons[:, 0], X_moons[:, 1],
                c=labels_agg, cmap='viridis', s=15)
axes[1].set_title('Agglomeratif (Ward)')

dendrogram(Z, ax=axes[2], leaf_font_size=6)
axes[2].set_title('Dendrogramme')
plt.tight_layout()
```

```
plt.savefig('dbscan_hierarchique.pdf')
```

Sortie

```
DBSCAN: 2 clusters, 0 bruit
```

Comparaison des indices de validité

```
from sklearn.metrics import (silhouette_score,
                             calinski_harabasz_score)

for name, lbl in [("k-Means", labels),
                  ("GMM", labels_gmm)]:
    sil = silhouette_score(X, lbl)
    ch  = calinski_harabasz_score(X, lbl)
    print(f"{name:10s} | Silhouette: {sil:.3f} | CH: {ch:.1f}")
```

Sortie

```
k-Means    | Silhouette: 0.687 | CH: 1210.3
GMM        | Silhouette: 0.687 | CH: 1210.3
```

8.8 Exercices

Exercice 8.1 (Niveau 1 — k-Means à la main)**.** On considère cinq points en $\mathbb{R}^2$: $\boldsymbol{x}_1 = (1, 1)$, $\boldsymbol{x}_2 = (1.5, 2)$, $\boldsymbol{x}_3 = (5, 4)$, $\boldsymbol{x}_4 = (6, 5)$, $\boldsymbol{x}_5 = (6.5, 4.5)$.

1. En initialisant $\boldsymbol{\mu}_1 = \boldsymbol{x}_1$ et $\boldsymbol{\mu}_2 = \boldsymbol{x}_3$, effectuer deux itérations de k-Means à la main.
2. L'algorithme a-t-il convergé ? Justifier.
3. Calculer le score silhouette de chaque point.

Exercice 8.2 (Niveau 2 — E-step et M-step du GMM)**.** On considère un mélange de deux gaussiennes univariées : $p(x) = \pi_1 \mathcal{N}(x \mid \mu_1, \sigma_1^2) + \pi_2 \mathcal{N}(x \mid \mu_2, \sigma_2^2)$.

1. Dériver les équations de l'E-step (éq. (8.8)) et du M-step (éq. (8.9)–(8.11)) dans le cas univarié.
2. Implémenter l'algorithme EM from scratch en Python pour les données $\{-2.1, -1.3, -0.4, 1.9, 2.3, 2$ avec $K = 2$.
3. Tracer l'évolution de la log-vraisemblance au fil des itérations. Vérifier qu'elle est monotone croissante.
4. Comparer avec `GaussianMixture` de scikit-learn.

Exercice 8.3 (Niveau 3 — Comparaison systématique)**.** 1. Générer trois jeux de données 2D : (a) `make_blobs` avec $K = 5$, (b) `make_moons`, (c) `make_circles`.

2. Appliquer k-Means, GMM, DBSCAN et clustering agglomératif à chaque jeu de données. Visualiser les résultats.

3. Pour chaque méthode et chaque jeu, calculer le score silhouette et l'indice de Calinski-Harabasz. Discuter.

4. Pour DBSCAN, étudier l'effet de ε et `min_samples` sur les résultats. Proposer une méthode pour choisir ε automatiquement (indice : graphe des k-distances).

5. Montrer théoriquement que k-Means est un cas particulier de EM avec covariances isotropes $\sigma^2 \boldsymbol{I}$ quand $\sigma^2 \to 0$.

Chapitre 9

Réduction de dimensionnalité — PCA, t-SNE, UMAP

9.1 Le fléau de la dimension

Définition 9.1 (Fléau de la dimension)**.** Le **fléau de la dimension** (*curse of dimensionality*) désigne l'ensemble des phénomènes qui rendent l'analyse de données en grande dimension fondamentalement différente de l'intuition en basse dimension.

Théorème 9.2 (Concentration des distances)**.** *Soit $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ des points i.i.d. tirés uniformément dans $[0,1]^d$. Pour $d \to \infty$:*

$$\frac{\max_i \|\boldsymbol{x}_i - \boldsymbol{x}_j\| - \min_i \|\boldsymbol{x}_i - \boldsymbol{x}_j\|}{\min_i \|\boldsymbol{x}_i - \boldsymbol{x}_j\|} \xrightarrow{p} 0 \tag{9.1}$$

Autrement dit, les distances entre points deviennent toutes équivalentes, rendant les méthodes fondées sur la distance (k-NN, k-Means) inefficaces.

Conséquences pratiques

- Le volume de l'hypersphère unitaire tend vers 0 quand $d \to \infty$.
- Un échantillon de taille fixe n devient de plus en plus « épars » en grande dimension.
- Il faut un nombre exponentiel de données $n \sim \mathcal{O}(c^d)$ pour maintenir une densité constante.

Proposition 9.3 (Volume de l'hypersphère)**.** Le volume de l'hypersphère de rayon r en dimension d est :

$$V_d(r) = \frac{\pi^{d/2}}{\Gamma(d/2+1)} r^d \tag{9.2}$$

En particulier, $V_d(1) \to 0$ quand $d \to \infty$.

La réduction de dimensionnalité cherche une représentation $\boldsymbol{z}_i \in \mathbb{R}^p$ (avec $p \ll d$) qui préserve au mieux la structure des données originales $\boldsymbol{x}_i \in \mathbb{R}^d$.

9.2 Analyse en composantes principales (PCA)

9.2.1 Formulation

Définition 9.4 (PCA)**.** L'**analyse en composantes principales** cherche les directions de variance maximale dans les données. Formellement, la première composante principale est :

$$\boldsymbol{w}_1 = \arg\max_{\|\boldsymbol{w}\|=1} \mathrm{Var}[\boldsymbol{w}^\top \boldsymbol{X}] = \arg\max_{\|\boldsymbol{w}\|=1} \boldsymbol{w}^\top \boldsymbol{S} \boldsymbol{w} \tag{9.3}$$

où $\boldsymbol{S} = \frac{1}{n-1}\sum_{i=1}^{n}(\boldsymbol{x}_i - \bar{\boldsymbol{x}})(\boldsymbol{x}_i - \bar{\boldsymbol{x}})^\top$ est la matrice de covariance empirique.

9.2.2 Dérivation par décomposition spectrale

Théorème 9.5 (Solution de la PCA)**.** *Les composantes principales sont les vecteurs propres de la matrice de covariance $\boldsymbol{S}$, ordonnés par valeurs propres décroissantes.*

Démonstration. On cherche $\boldsymbol{w}_1 = \arg\max_{\|\boldsymbol{w}\|=1} \boldsymbol{w}^\top \boldsymbol{S}\boldsymbol{w}$. Par la méthode des multiplicateurs de Lagrange :

$$\mathcal{L}(\boldsymbol{w}, \lambda) = \boldsymbol{w}^\top \boldsymbol{S}\boldsymbol{w} - \lambda(\boldsymbol{w}^\top \boldsymbol{w} - 1) \tag{9.4}$$

La condition de premier ordre donne :

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}} = 2\boldsymbol{S}\boldsymbol{w} - 2\lambda \boldsymbol{w} = \boldsymbol{0} \quad \Longrightarrow \quad \boldsymbol{S}\boldsymbol{w} = \lambda \boldsymbol{w} \tag{9.5}$$

Donc $\boldsymbol{w}$ est un vecteur propre de $\boldsymbol{S}$. En multipliant à gauche par $\boldsymbol{w}^\top$:

$$\boldsymbol{w}^\top \boldsymbol{S}\boldsymbol{w} = \lambda \boldsymbol{w}^\top \boldsymbol{w} = \lambda \tag{9.6}$$

La variance projetée est égale à la valeur propre. Pour maximiser, on choisit le vecteur propre associé à la plus grande valeur propre λ_1. La k-ième composante est le k-ième vecteur propre (sous contrainte d'orthogonalité avec les précédents). □

9.2.3 Dérivation par SVD

PCA via la décomposition en valeurs singulières

Soit $\tilde{\boldsymbol{X}} \in \mathbb{R}^{n \times d}$ la matrice de données centrées (lignes $= \boldsymbol{x}_i - \bar{\boldsymbol{x}}$). Sa SVD est :

$$\tilde{\boldsymbol{X}} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top \tag{9.7}$$

où $\boldsymbol{U} \in \mathbb{R}^{n \times n}$, $\boldsymbol{\Sigma} \in \mathbb{R}^{n \times d}$ (diagonale), $\boldsymbol{V} \in \mathbb{R}^{d \times d}$ orthogonales. Alors :

$$\boldsymbol{S} = \frac{1}{n-1}\tilde{\boldsymbol{X}}^\top \tilde{\boldsymbol{X}} = \boldsymbol{V}\frac{\boldsymbol{\Sigma}^\top \boldsymbol{\Sigma}}{n-1}\boldsymbol{V}^\top \tag{9.8}$$

Les colonnes de $\boldsymbol{V}$ sont les composantes principales, et les valeurs propres de $\boldsymbol{S}$ sont $\lambda_k = \sigma_k^2/(n-1)$.

Projection : les coordonnées dans la base des composantes principales sont :

$$\boldsymbol{Z} = \tilde{\boldsymbol{X}}\boldsymbol{V}_p \in \mathbb{R}^{n \times p} \tag{9.9}$$

où $\boldsymbol{V}_p$ contient les p premiers vecteurs propres.

Remarque 9.6. La SVD est préférable numériquement au calcul explicite de $\boldsymbol{S} = \tilde{\boldsymbol{X}}^\top \tilde{\boldsymbol{X}}$ car elle évite la perte de précision liée au conditionnement de $\boldsymbol{S}$.

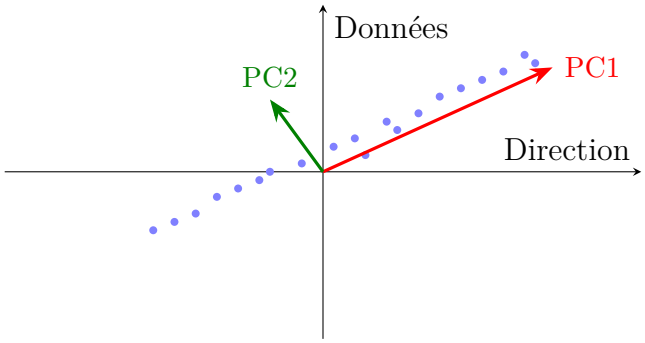
9.2.4 Variance expliquée et choix de p

Définition 9.7 (Ratio de variance expliquée)**.** Le ratio de variance expliquée par les p premières composantes est :

$$\rho(p) = \frac{\sum_{k=1}^{p} \lambda_k}{\sum_{k=1}^{d} \lambda_k} \tag{9.10}$$

Choix du nombre de composantes

- **Scree plot :** tracer λ_k en fonction de k et chercher un « coude ».
- **Seuil de variance :** choisir p tel que $\rho(p) \geq 0.95$ (ou 0.90).
- **Règle de Kaiser :** retenir les composantes dont $\lambda_k > \bar{\lambda} = \frac{1}{d}\sum_{k=1}^{d} \lambda_k$.



9.2.5 Reconstruction et erreur

Proposition 9.8 (Erreur de reconstruction)**.** La PCA minimise également l'erreur de reconstruction au sens des moindres carrés :

$$\min_{\boldsymbol{V}_p} \sum_{i=1}^{n} \left\| \boldsymbol{x}_i - \bar{\boldsymbol{x}} - \boldsymbol{V}_p \boldsymbol{V}_p^\top (\boldsymbol{x}_i - \bar{\boldsymbol{x}}) \right\|^2 = \sum_{k=p+1}^{d} \lambda_k \tag{9.11}$$

Les p composantes principales fournissent la meilleure approximation de rang p au sens de Frobenius (théorème d'Eckart-Young).

9.3 Kernel PCA

Définition 9.9 (Kernel PCA)**.** Lorsque les données ne sont pas linéairement séparables, on peut appliquer la PCA dans un espace de Hilbert $\mathcal{H}$ induit par un noyau $\kappa : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$. On définit la matrice de Gram $\boldsymbol{K}_{ij} = \kappa(\boldsymbol{x}_i, \boldsymbol{x}_j)$ et on résout :

$$\tilde{\boldsymbol{K}} \boldsymbol{\alpha}_k = n \lambda_k \boldsymbol{\alpha}_k \tag{9.12}$$

où $\tilde{\boldsymbol{K}}$ est la matrice de Gram centrée : $\tilde{\boldsymbol{K}} = \boldsymbol{K} - \mathbf{1}_n \boldsymbol{K} - \boldsymbol{K}\mathbf{1}_n + \mathbf{1}_n \boldsymbol{K} \mathbf{1}_n$ avec $(\mathbf{1}_n)_{ij} = 1/n$.

Remarque 9.10. Les noyaux courants sont : RBF $\kappa(\boldsymbol{x}, \boldsymbol{x}') = \exp(-\gamma \left\| \boldsymbol{x} - \boldsymbol{x}' \right\|^2)$, polynomial $\kappa(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x}^\top \boldsymbol{x}' + c)^p$, sigmoïde.

9.4 t-SNE

9.4.1 Principe

Définition 9.11 (t-SNE)**. t-SNE** (*t-distributed Stochastic Neighbor Embedding*) est une méthode de visualisation non linéaire qui préserve la structure locale des données. Elle opère en deux étapes :

1. **Espace d'entrée :** construire des probabilités conjointes p_{ij} mesurant la similarité entre $\boldsymbol{x}_i$ et $\boldsymbol{x}_j$:

$$p_{j|i} = \frac{\exp(-\|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2 / 2\sigma_i^2)}{\sum_{k \neq i} \exp(-\|\boldsymbol{x}_i - \boldsymbol{x}_k\|^2 / 2\sigma_i^2)}, \quad p_{ij} = \frac{p_{j|i} + p_{i|j}}{2n} \tag{9.13}$$

2. **Espace de sortie :** définir des probabilités q_{ij} basées sur une distribution t de Student à 1 degré de liberté :

$$q_{ij} = \frac{(1 + \|\boldsymbol{z}_i - \boldsymbol{z}_j\|^2)^{-1}}{\sum_{k \neq l} (1 + \|\boldsymbol{z}_k - \boldsymbol{z}_l\|^2)^{-1}} \tag{9.14}$$

On minimise la divergence de Kullback-Leibler :

$$\mathrm{KL}(P \| Q) = \sum_{i \neq j} p_{ij} \ln \frac{p_{ij}}{q_{ij}} \tag{9.15}$$

par descente de gradient sur les positions $\boldsymbol{z}_i$.

9.4.2 Perplexité

Définition 9.12 (Perplexité)**.** La **perplexité** contrôle le nombre effectif de voisins :

$$\mathrm{Perp}(P_i) = 2^{H(P_i)}, \quad H(P_i) = -\sum_j p_{j|i} \log_2 p_{j|i} \tag{9.16}$$

Le paramètre σ_i est ajusté pour atteindre la perplexité cible (typiquement 5–50).

Limitations de t-SNE

- **Non reproductible :** résultats différents selon l'initialisation (utiliser `random_state`).
- **Distances globales non fiables :** seules les distances locales sont préservées. La taille et l'écartement des clusters ne sont pas interprétables.
- **Complexité :** $\mathcal{O}(n^2)$ en mémoire et temps (variante Barnes-Hut en $\mathcal{O}(n \log n)$).
- **Perplexité sensible :** différentes perplexités donnent des visualisations très différentes.

9.5 UMAP

Définition 9.13 (UMAP)**. UMAP** (*Uniform Manifold Approximation and Projection*) est une méthode de réduction non linéaire fondée sur la théorie des variétés riemanniennes et la topologie algébrique. Comme t-SNE, elle préserve la structure locale, mais elle :

- préserve mieux la structure *globale*,
- est significativement plus rapide ($\mathcal{O}(n)$ après construction du graphe),
- possède un cadre mathématique plus rigoureux.

Les principaux hyperparamètres sont :

- `n_neighbors` : nombre de voisins (analogue à la perplexité), contrôle l'équilibre local/global.
- `min_dist` : distance minimale entre points en sortie, contrôle la compacité des clusters.

Remarque 9.14. UMAP peut également être utilisé pour la réduction de dimension en vue d'un apprentissage supervisé (mode `supervised`).

9.6 Implémentation en Python

PCA from scratch

```python
import numpy as np

def pca_scratch(X, n_components):
    """PCA via decomposition spectrale de la covariance."""
    # Centrer les donnees
    X_centered = X - X.mean(axis=0)
    # Matrice de covariance
    n = X.shape[0]
    S = (X_centered.T @ X_centered) / (n - 1)
    # Decomposition en valeurs propres
    eigenvalues, eigenvectors = np.linalg.eigh(S)
    # Trier par valeurs propres decroissantes
    idx = np.argsort(eigenvalues)[::-1]
    eigenvalues = eigenvalues[idx]
    eigenvectors = eigenvectors[:, idx]
    # Projeter sur les p premieres composantes
    W = eigenvectors[:, :n_components]
    Z = X_centered @ W
    explained_ratio = eigenvalues[:n_components] / eigenvalues.sum()
    return Z, eigenvalues, explained_ratio

# Test sur Iris
from sklearn.datasets import load_iris
X, y = load_iris(return_X_y=True)
Z, eigvals, ratios = pca_scratch(X, n_components=2)
```

```
print(f"Variance expliquee: {ratios}")
print(f"Cumul: {ratios.sum():.4f}")
```

Sortie

```
Variance expliquee: [0.72962445 0.22850762]
Cumul: 0.9581
```

PCA avec scikit-learn et scree plot

```
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

pca = PCA()
pca.fit(X)

fig, axes = plt.subplots(1, 3, figsize=(15, 4))

# Scree plot
axes[0].bar(range(1, 5), pca.explained_variance_ratio_,
            color='steelblue', alpha=0.8)
axes[0].plot(range(1, 5),
             np.cumsum(pca.explained_variance_ratio_),
             'ro-')
axes[0].set_xlabel('Composante')
axes[0].set_ylabel('Variance expliquee')
axes[0].set_title('Scree plot')
axes[0].axhline(y=0.95, color='gray', linestyle='--',
                label='Seuil 95%')
axes[0].legend()

# Projection 2D
pca2 = PCA(n_components=2)
Z_sk = pca2.fit_transform(X)
scatter = axes[1].scatter(Z_sk[:, 0], Z_sk[:, 1],
                          c=y, cmap='viridis', s=20)
axes[1].set_xlabel('PC1')
axes[1].set_ylabel('PC2')
axes[1].set_title('PCA - Iris')
plt.colorbar(scatter, ax=axes[1])

# Biplot (loadings)
loadings = pca2.components_.T
feature_names = load_iris().feature_names
for i, name in enumerate(feature_names):
    axes[2].arrow(0, 0, loadings[i, 0], loadings[i, 1],
                  head_width=0.03, color='red', alpha=0.7)
    axes[2].text(loadings[i, 0]*1.15, loadings[i, 1]*1.15,
                 name, fontsize=7)
axes[2].set_xlabel('PC1'); axes[2].set_ylabel('PC2')
```

```
axes[2].set_title('Biplot (loadings)')

plt.tight_layout()
plt.savefig('pca_iris.pdf')
```

Kernel PCA, t-SNE et UMAP

```
from sklearn.decomposition import KernelPCA
from sklearn.manifold import TSNE
# pip install umap-learn
import umap

# Kernel PCA (RBF)
kpca = KernelPCA(n_components=2, kernel='rbf', gamma=0.1)
Z_kpca = kpca.fit_transform(X)

# t-SNE
tsne = TSNE(n_components=2, perplexity=30, random_state=42,
            n_iter=1000, learning_rate='auto', init='pca')
Z_tsne = tsne.fit_transform(X)

# UMAP
reducer = umap.UMAP(n_neighbors=15, min_dist=0.1,
                    random_state=42)
Z_umap = reducer.fit_transform(X)

fig, axes = plt.subplots(1, 3, figsize=(15, 4))
for ax, Z_proj, title in zip(
    axes,
    [Z_kpca, Z_tsne, Z_umap],
    ['Kernel PCA (RBF)', 't-SNE', 'UMAP']
):
    ax.scatter(Z_proj[:, 0], Z_proj[:, 1],
               c=y, cmap='viridis', s=20)
    ax.set_title(title)
plt.tight_layout()
plt.savefig('reduction_comparison.pdf')
```

Effet de la perplexité (t-SNE)

```
from sklearn.datasets import load_digits

X_digits, y_digits = load_digits(return_X_y=True)

fig, axes = plt.subplots(1, 4, figsize=(16, 4))
for ax, perp in zip(axes, [5, 15, 30, 50]):
    tsne = TSNE(n_components=2, perplexity=perp,
                random_state=42, init='pca',
```

```
                learning_rate='auto')
    Z = tsne.fit_transform(X_digits)
    ax.scatter(Z[:, 0], Z[:, 1], c=y_digits,
               cmap='tab10', s=5, alpha=0.7)
    ax.set_title(f'Perplexite = {perp}')
    ax.set_xticks([]); ax.set_yticks([])
plt.tight_layout()
plt.savefig('tsne_perplexity.pdf')
```

9.7 Exercices

Exercice 9.1 (Niveau 1 — PCA analytique)**.** On considère les données centrées : $\boldsymbol{x}_1 = (2,1)^\top$, $\boldsymbol{x}_2 = (-1,-1)^\top$, $\boldsymbol{x}_3 = (1,0)^\top$, $\boldsymbol{x}_4 = (-2,0)^\top$.

1. Calculer la matrice de covariance $\boldsymbol{S}$.
2. Déterminer les valeurs propres et vecteurs propres de $\boldsymbol{S}$.
3. Projeter les données sur la première composante principale.
4. Calculer le ratio de variance expliquée $\rho(1)$.

Exercice 9.2 (Niveau 2 — PCA et SVD)**.**

1. Implémenter la PCA via la SVD (`np.linalg.svd`) et vérifier que les résultats coïncident avec ceux de `PCA` de scikit-learn sur le dataset Iris.
2. Comparer le temps d'exécution de la PCA par décomposition spectrale ($\boldsymbol{S} = \boldsymbol{V}\boldsymbol{\Lambda}\boldsymbol{V}^\top$) et par SVD ($\tilde{\boldsymbol{X}} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top$) pour $d \in \{10, 100, 1000\}$ avec $n = 500$.
3. Montrer que l'erreur de reconstruction $\left\|\boldsymbol{X} - \boldsymbol{X}\boldsymbol{V}_p\boldsymbol{V}_p^\top\right\|_F^2 = \sum_{k=p+1}^{d} \lambda_k$ en utilisant la SVD.
4. Appliquer Kernel PCA avec un noyau RBF au dataset `make_circles`. Comparer avec la PCA linéaire.

Exercice 9.3 (Niveau 3 — Analyse complète MNIST)**.**

1. Charger le dataset MNIST (ou `load_digits` de scikit-learn).
2. Appliquer la PCA et tracer le scree plot. Combien de composantes faut-il pour expliquer 95 % de la variance ?
3. Reconstruire des images à partir de $p \in \{5, 10, 30, 50, 100\}$ composantes. Visualiser la qualité de reconstruction.
4. Comparer les visualisations 2D obtenues par PCA, t-SNE (perplexités 10, 30, 50) et UMAP (`n_neighbors` $\in \{5, 15, 50\}$). Discuter les différences.
5. Mesurer l'effet de la réduction de dimension (PCA) sur la performance d'un classifieur k-NN en fonction de p. Tracer la courbe accuracy vs p. Y a-t-il un p optimal ?
6. (*Bonus*) Démontrer que la PCA est équivalente à la factorisation matricielle de rang p minimisant la norme de Frobenius (théorème d'Eckart-Young).

Chapitre 10

Apprentissage bayésien

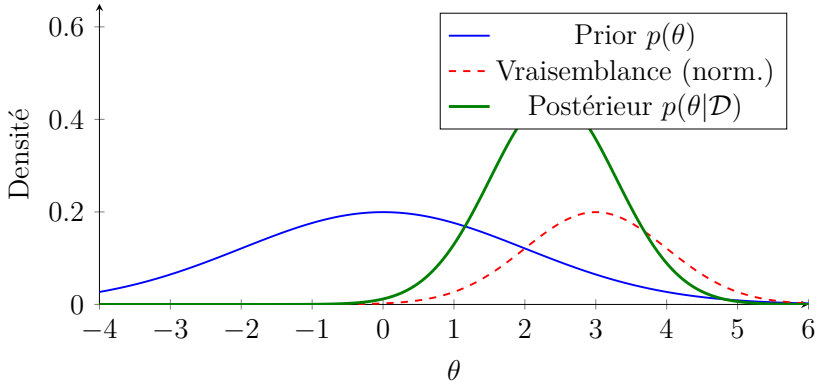
10.1 Inférence bayésienne

10.1.1 Le théorème de Bayes

Théorème 10.1 (Théorème de Bayes)**.** *Soit $\boldsymbol{\theta}$ le vecteur de paramètres et $\mathcal{D}$ les données observées. La* ***distribution a posteriori*** *est :*

$$\underbrace{p(\boldsymbol{\theta} \mid \mathcal{D})}_{\text{postérieur}} = \frac{\overbrace{p(\mathcal{D} \mid \boldsymbol{\theta})}^{\text{vraisemblance}} \cdot \overbrace{p(\boldsymbol{\theta})}^{\text{prior}}}{\underbrace{p(\mathcal{D})}_{\text{évidence}}} \tag{10.1}$$

où $p(\mathcal{D}) = \int p(\mathcal{D} \mid \boldsymbol{\theta})\, p(\boldsymbol{\theta})\, d\boldsymbol{\theta}$ est la vraisemblance marginale.



Interprétation

- Le **prior** $p(\boldsymbol{\theta})$ encode nos croyances avant d'observer les données.
- La **vraisemblance** $p(\mathcal{D} \mid \boldsymbol{\theta})$ mesure la compatibilité des données avec les paramètres.
- Le **postérieur** $p(\boldsymbol{\theta} \mid \mathcal{D})$ combine les deux : il est un compromis entre prior et données.
- Plus on a de données, plus le postérieur est dominé par la vraisemblance.

10.1.2 MAP vs MLE

Définition 10.2 (Maximum de vraisemblance (MLE))**.** L'estimateur du **maximum de vraisemblance** est :

$$\hat{\boldsymbol{\theta}}_{\text{MLE}} = \arg\max_{\boldsymbol{\theta}} \; p(\mathcal{D} \mid \boldsymbol{\theta}) = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{n} \ln p(\boldsymbol{x}_i \mid \boldsymbol{\theta}) \tag{10.2}$$

Définition 10.3 (Maximum a posteriori (MAP))**.** L'estimateur **MAP** intègre le prior :

$$\hat{\boldsymbol{\theta}}_{\text{MAP}} = \arg\max_{\boldsymbol{\theta}} \; p(\boldsymbol{\theta} \mid \mathcal{D}) = \arg\max_{\boldsymbol{\theta}} \left[\ln p(\mathcal{D} \mid \boldsymbol{\theta}) + \ln p(\boldsymbol{\theta})\right] \tag{10.3}$$

Proposition 10.4 (MAP et régularisation)**.** L'estimation MAP avec un prior gaussien $p(\boldsymbol{\theta}) = \mathcal{N}(\mathbf{0}, \tau^2 \boldsymbol{I})$ est équivalente à la régularisation L_2 (Ridge) :

$$\hat{\boldsymbol{\theta}}_{\text{MAP}} = \arg\min_{\boldsymbol{\theta}} \left[-\ln p(\mathcal{D} \mid \boldsymbol{\theta}) + \frac{1}{2\tau^2} \|\boldsymbol{\theta}\|^2 \right] \tag{10.4}$$

De même, un prior Laplacien donne la régularisation L_1 (Lasso).

MAP $\neq$ bayésien complet

L'estimation MAP fournit une estimation *ponctuelle* de $\boldsymbol{\theta}$. L'approche bayésienne complète utilise la distribution *entière* du postérieur, ce qui permet de quantifier l'incertitude.

10.2 Priors conjugués

Définition 10.5 (Famille conjuguée)**.** Une famille de priors $\mathcal{F}$ est **conjuguée** pour une vraisemblance donnée si le postérieur appartient également à $\mathcal{F}$:

$$p(\boldsymbol{\theta}) \in \mathcal{F} \quad \Longrightarrow \quad p(\boldsymbol{\theta} \mid \mathcal{D}) \in \mathcal{F} \tag{10.5}$$

Exemples de priors conjugués

Vraisemblance	Prior conjugué	Postérieur
Bernoulli(p)	Beta(α, β)	Beta($\alpha + s, \beta + n - s$)
Poisson(λ)	Gamma(α, β)	Gamma($\alpha + s, \beta + n$)
$\mathcal{N}(\mu, \sigma^2$ connu)	$\mathcal{N}(\mu_0, \sigma_0^2)$	$\mathcal{N}(\mu_n, \sigma_n^2)$
$\mathcal{N}(\mu$ connu, $\sigma^2)$	Inv-Gamma(α, β)	Inv-Gamma(α', β')

Exemple 10.6 (Modèle gaussien avec variance connue)**.** Soit $x_1, \ldots, x_n \overset{\text{i.i.d.}}{\sim} \mathcal{N}(\mu, \sigma^2)$ avec σ^2 connu et prior $\mu \sim \mathcal{N}(\mu_0, \sigma_0^2)$. Le postérieur est :

$$\mu \mid \mathcal{D} \sim \mathcal{N}(\mu_n, \sigma_n^2) \tag{10.6}$$

avec :

$$\sigma_n^2 = \left(\frac{1}{\sigma_0^2} + \frac{n}{\sigma^2}\right)^{-1} \tag{10.7}$$

$$\mu_n = \sigma_n^2 \left(\frac{\mu_0}{\sigma_0^2} + \frac{n\bar{x}}{\sigma^2}\right) \tag{10.8}$$

où $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$. On note que μ_n est une moyenne pondérée entre le prior μ_0 et la moyenne empirique $\bar{x}$.

10.3 Régression linéaire bayésienne

10.3.1 Modèle

On considère le modèle linéaire :

$$y_i = \boldsymbol{w}^\top \boldsymbol{x}_i + \varepsilon_i, \quad \varepsilon_i \sim \mathcal{N}(0, \sigma^2) \tag{10.9}$$

soit en forme matricielle $\boldsymbol{y} = \boldsymbol{X}\boldsymbol{w} + \boldsymbol{\varepsilon}$ avec $\boldsymbol{X} \in \mathbb{R}^{n \times d}$.

10.3.2 Prior et postérieur

Régression linéaire bayésienne — Dérivation complète

Prior : $\boldsymbol{w} \sim \mathcal{N}(\boldsymbol{m}_0, \boldsymbol{S}_0)$.
Vraisemblance :

$$p(\boldsymbol{y} \mid \boldsymbol{X}, \boldsymbol{w}) = \mathcal{N}(\boldsymbol{y} \mid \boldsymbol{X}\boldsymbol{w}, \sigma^2 \boldsymbol{I}_n) = \prod_{i=1}^{n} \mathcal{N}(y_i \mid \boldsymbol{w}^\top \boldsymbol{x}_i, \sigma^2) \tag{10.10}$$

Postérieur : par conjugaison gaussienne :

$$p(\boldsymbol{w} \mid \boldsymbol{X}, \boldsymbol{y}) = \mathcal{N}(\boldsymbol{w} \mid \boldsymbol{m}_n, \boldsymbol{S}_n) \tag{10.11}$$

avec :

$$\boldsymbol{S}_n = \left(\boldsymbol{S}_0^{-1} + \frac{1}{\sigma^2}\boldsymbol{X}^\top \boldsymbol{X}\right)^{-1} \tag{10.12}$$

$$\boldsymbol{m}_n = \boldsymbol{S}_n \left(\boldsymbol{S}_0^{-1}\boldsymbol{m}_0 + \frac{1}{\sigma^2}\boldsymbol{X}^\top \boldsymbol{y}\right) \tag{10.13}$$

Démonstration. La log-densité du postérieur (aux constantes près) est :

$$\ln p(\boldsymbol{w} \mid \boldsymbol{X}, \boldsymbol{y}) \propto \ln p(\boldsymbol{y} \mid \boldsymbol{X}, \boldsymbol{w}) + \ln p(\boldsymbol{w}) \tag{10.14}$$

$$= -\frac{1}{2\sigma^2}\|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\|^2 - \frac{1}{2}(\boldsymbol{w} - \boldsymbol{m}_0)^\top \boldsymbol{S}_0^{-1}(\boldsymbol{w} - \boldsymbol{m}_0) + \text{cst} \tag{10.15}$$

En développant les termes quadratiques en $\boldsymbol{w}$ et en complétant le carré :

$$= -\frac{1}{2}\boldsymbol{w}^\top \underbrace{\left(\boldsymbol{S}_0^{-1} + \frac{1}{\sigma^2}\boldsymbol{X}^\top \boldsymbol{X}\right)}_{\boldsymbol{S}_n^{-1}} \boldsymbol{w} + \boldsymbol{w}^\top \underbrace{\left(\boldsymbol{S}_0^{-1}\boldsymbol{m}_0 + \frac{1}{\sigma^2}\boldsymbol{X}^\top \boldsymbol{y}\right)}_{\boldsymbol{S}_n^{-1}\boldsymbol{m}_n} + \text{cst} \tag{10.16}$$

On reconnaît une forme quadratique gaussienne $\mathcal{N}(\boldsymbol{m}_n, \boldsymbol{S}_n)$. □

Remarque 10.7. Avec le prior $\boldsymbol{m}_0 = \mathbf{0}$ et $\boldsymbol{S}_0 = \tau^2 \boldsymbol{I}$, la moyenne du postérieur coïncide avec la solution Ridge :

$$\boldsymbol{m}_n = \left(\boldsymbol{X}^\top \boldsymbol{X} + \frac{\sigma^2}{\tau^2} \boldsymbol{I} \right)^{-1} \boldsymbol{X}^\top \boldsymbol{y} \tag{10.17}$$

10.3.3 Distribution prédictive

Définition 10.8 (Distribution prédictive)**.** Pour un nouveau point $\boldsymbol{x}_*$, la **distribution prédictive** intègre sur tous les paramètres possibles :

$$p(y_* \mid \boldsymbol{x}_*, \mathcal{D}) = \int p(y_* \mid \boldsymbol{x}_*, \boldsymbol{w})\, p(\boldsymbol{w} \mid \mathcal{D})\, d\boldsymbol{w} \tag{10.18}$$

Théorème 10.9 (Prédiction bayésienne linéaire)**.** *La distribution prédictive est gaussienne :*

$$p(y_* \mid \boldsymbol{x}_*, \mathcal{D}) = \mathcal{N}(y_* \mid \boldsymbol{m}_n^\top \boldsymbol{x}_*,\ \sigma_*^2) \tag{10.19}$$

avec :

$$\sigma_*^2 = \sigma^2 + \boldsymbol{x}_*^\top \boldsymbol{S}_n \boldsymbol{x}_* \tag{10.20}$$

*Le premier terme σ^2 est le bruit aléatoire et le second $\boldsymbol{x}_*^\top \boldsymbol{S}_n \boldsymbol{x}_*$ est l'**incertitude épistémique** (liée au manque de données).*

10.4 Processus gaussiens

10.4.1 Définition

Définition 10.10 (Processus gaussien)**.** Un **processus gaussien** (GP) est une collection de variables aléatoires dont toute sous-collection finie suit une distribution gaussienne jointe. Un GP est entièrement défini par :

- une fonction de moyenne $m(\boldsymbol{x}) = \mathbb{E}[f(\boldsymbol{x})]$,
- une fonction de covariance (noyau) $\kappa(\boldsymbol{x}, \boldsymbol{x}') = \mathrm{Cov}[f(\boldsymbol{x}), f(\boldsymbol{x}')]$.

On note $f \sim \mathcal{GP}(m, \kappa)$.

10.4.2 Fonctions noyau

Noyaux courants

$$\text{RBF (SE)} : \quad \kappa(\boldsymbol{x}, \boldsymbol{x}') = \sigma_f^2 \exp\left(-\frac{\|\boldsymbol{x} - \boldsymbol{x}'\|^2}{2\ell^2}\right) \tag{10.21}$$

$$\text{Matérn } \nu = 3/2 : \quad \kappa(\boldsymbol{x}, \boldsymbol{x}') = \sigma_f^2 \left(1 + \frac{\sqrt{3}\, r}{\ell}\right) \exp\left(-\frac{\sqrt{3}\, r}{\ell}\right) \tag{10.22}$$

$$\text{Périodique} : \quad \kappa(\boldsymbol{x}, \boldsymbol{x}') = \sigma_f^2 \exp\left(-\frac{2\sin^2(\pi \|\boldsymbol{x} - \boldsymbol{x}'\| / p)}{\ell^2}\right) \tag{10.23}$$

$$\text{Linéaire} : \quad \kappa(\boldsymbol{x}, \boldsymbol{x}') = \sigma_f^2\, \boldsymbol{x}^\top \boldsymbol{x}' \tag{10.24}$$

où $r = \|\boldsymbol{x} - \boldsymbol{x}'\|$, ℓ est la longueur caractéristique, σ_f^2 la variance du signal.

Rôle des hyperparamètres

- ℓ (longueur caractéristique) : contrôle la « rugosité » de la fonction. Grand ℓ $\Rightarrow$ fonctions lisses.
- σ_f^2 (variance du signal) : contrôle l'amplitude des variations.
- σ_n^2 (bruit) : variance du bruit d'observation.

10.4.3 Régression par processus gaussien

Soit $\mathcal{D} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$ avec $y_i = f(\boldsymbol{x}_i) + \varepsilon_i$, $\varepsilon_i \sim \mathcal{N}(0, \sigma_n^2)$.

Régression GP — Formules prédictives

Pour un ensemble de points test $\boldsymbol{X}_*$, la distribution prédictive est :

$$\boldsymbol{f}_* \mid \boldsymbol{X}_*, \boldsymbol{X}, \boldsymbol{y} \sim \mathcal{N}(\bar{\boldsymbol{f}}_*, \mathrm{Cov}(\boldsymbol{f}_*)) \tag{10.25}$$

avec :

$$\bar{\boldsymbol{f}}_* = \boldsymbol{K}_*^\top (\boldsymbol{K} + \sigma_n^2 \boldsymbol{I})^{-1} \boldsymbol{y} \tag{10.26}$$

$$\mathrm{Cov}(\boldsymbol{f}_*) = \boldsymbol{K}_{**} - \boldsymbol{K}_*^\top (\boldsymbol{K} + \sigma_n^2 \boldsymbol{I})^{-1} \boldsymbol{K}_* \tag{10.27}$$

où $\boldsymbol{K} = \kappa(\boldsymbol{X}, \boldsymbol{X}) \in \mathbb{R}^{n \times n}$, $\boldsymbol{K}_* = \kappa(\boldsymbol{X}, \boldsymbol{X}_*) \in \mathbb{R}^{n \times n_*}$, $\boldsymbol{K}_{**} = \kappa(\boldsymbol{X}_*, \boldsymbol{X}_*) \in \mathbb{R}^{n_* \times n_*}$.

Dérivation. Par définition du GP, la distribution jointe est :

$$\begin{pmatrix} \boldsymbol{y} \\ \boldsymbol{f}_* \end{pmatrix} \sim \mathcal{N}\left(\boldsymbol{0}, \begin{pmatrix} \boldsymbol{K} + \sigma_n^2 \boldsymbol{I} & \boldsymbol{K}_* \\ \boldsymbol{K}_*^\top & \boldsymbol{K}_{**} \end{pmatrix}\right) \tag{10.28}$$

Par la formule de conditionnement gaussien, si $\begin{pmatrix} \boldsymbol{a} \\ \boldsymbol{b} \end{pmatrix} \sim \mathcal{N}\left(\boldsymbol{0}, \begin{pmatrix} \boldsymbol{A} & \boldsymbol{C} \\ \boldsymbol{C}^\top & \boldsymbol{B} \end{pmatrix}\right)$, alors :

$$\boldsymbol{b} \mid \boldsymbol{a} \sim \mathcal{N}(\boldsymbol{C}^\top \boldsymbol{A}^{-1} \boldsymbol{a},\ \boldsymbol{B} - \boldsymbol{C}^\top \boldsymbol{A}^{-1} \boldsymbol{C}) \tag{10.29}$$

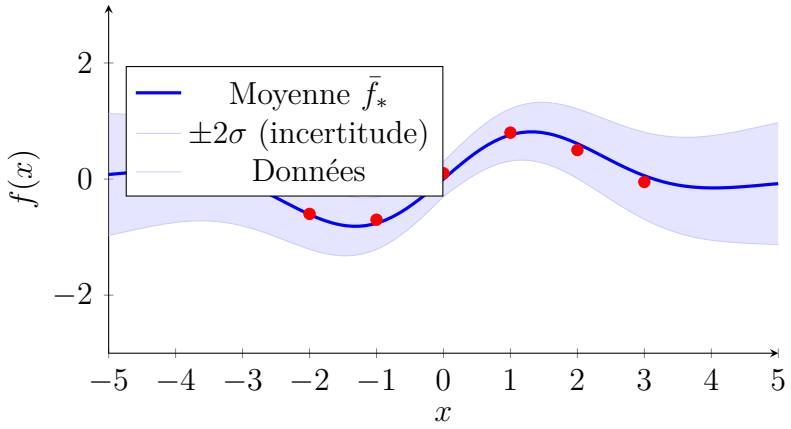
L'application directe donne les équations (10.26) et (10.27). □

10.4.4 Optimisation des hyperparamètres

Définition 10.11 (Log-vraisemblance marginale)**.** Les hyperparamètres $\boldsymbol{\phi} = (\ell, \sigma_f, \sigma_n)$ sont optimisés par maximisation de la log-vraisemblance marginale :

$$\ln p(\boldsymbol{y} \mid \boldsymbol{X}, \boldsymbol{\phi}) = -\frac{1}{2}\boldsymbol{y}^\top (\boldsymbol{K} + \sigma_n^2 \boldsymbol{I})^{-1}\boldsymbol{y} - \frac{1}{2}\ln|\boldsymbol{K} + \sigma_n^2 \boldsymbol{I}| - \frac{n}{2}\ln(2\pi) \tag{10.30}$$

Les trois termes représentent respectivement l'ajustement aux données, la pénalité de complexité, et une constante de normalisation.



10.5 Implémentation en Python

Régression linéaire bayésienne from scratch

```python
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(42)

# Donnees synthetiques
n, d = 20, 1
X = np.sort(np.random.uniform(-3, 3, (n, 1)), axis=0)
w_true = np.array([1.5])
sigma = 0.5
y = X @ w_true + sigma * np.random.randn(n, 1)

# Ajouter biais (colonne de 1)
Phi = np.hstack([np.ones((n, 1)), X])  # n x 2

# Prior: w ~ N(m0, S0)
m0 = np.zeros((2, 1))
S0 = 2.0 * np.eye(2)

# Posterieur analytique
S0_inv = np.linalg.inv(S0)
Sn_inv = S0_inv + (1/sigma**2) * Phi.T @ Phi
Sn = np.linalg.inv(Sn_inv)
```

```
mn = Sn @ (S0_inv @ m0 + (1/sigma**2) * Phi.T @ y)

print(f"Posterieur: m_n = {mn.ravel()}")
print(f"Posterieur: diag(S_n) = {np.diag(Sn)}")

# Distribution predictive
X_test = np.linspace(-4, 4, 200).reshape(-1, 1)
Phi_test = np.hstack([np.ones((200, 1)), X_test])

y_pred = Phi_test @ mn
y_var = sigma**2 + np.sum(Phi_test @ Sn * Phi_test, axis=1,
                          keepdims=True)
y_std = np.sqrt(y_var)

plt.figure(figsize=(8, 5))
plt.scatter(X, y, c='red', zorder=5, label='Donnees')
plt.plot(X_test, y_pred, 'b-', label='Moyenne predictive')
plt.fill_between(X_test.ravel(),
                 (y_pred - 2*y_std).ravel(),
                 (y_pred + 2*y_std).ravel(),
                 alpha=0.2, color='blue',
                 label='$\pm 2\sigma$')
plt.xlabel('x'); plt.ylabel('y')
plt.legend(); plt.title('Regression lineaire bayesienne')
plt.savefig('blr_result.pdf')
```

Sortie

```
Posterieur: m_n = [0.041 1.473]
Posterieur: diag(S_n) = [0.0124 0.0054]
```

Régression par processus gaussien avec scikit-learn

```
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import (
    RBF, ConstantKernel, WhiteKernel, Matern
)

# Donnees
np.random.seed(42)
X_train = np.array([-4, -3, -2, -1, 1, 2, 3]).reshape(-1, 1)
y_train = np.sin(X_train).ravel() + 0.1 * np.random.randn(7)

# Noyau: amplitude * RBF + bruit
kernel = (ConstantKernel(1.0) * RBF(length_scale=1.0)
          + WhiteKernel(noise_level=0.1))

gp = GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=10,
                              random_state=42)
```

```
gp.fit(X_train, y_train)

print(f"Noyau optimise: {gp.kernel_}")
print(f"Log-vraisemblance: {gp.log_marginal_likelihood_value_:.3f}")

# Prediction
X_test = np.linspace(-5, 5, 200).reshape(-1, 1)
y_mean, y_std = gp.predict(X_test, return_std=True)

plt.figure(figsize=(8, 5))
plt.scatter(X_train, y_train, c='red', zorder=5,
            label='Donnees')
plt.plot(X_test, y_mean, 'b-', label='Moyenne GP')
plt.fill_between(X_test.ravel(),
                 y_mean - 2*y_std, y_mean + 2*y_std,
                 alpha=0.2, color='blue',
                 label='$\pm 2\sigma$')
plt.plot(X_test, np.sin(X_test), 'k--', alpha=0.5,
         label='$\sin(x)$ (vrai)')
plt.xlabel('x'); plt.ylabel('y')
plt.legend(); plt.title('Regression par processus gaussien')
plt.savefig('gp_regression.pdf')
```

Effet de la longueur caractéristique

```
fig, axes = plt.subplots(1, 3, figsize=(15, 4))

for ax, length_scale in zip(axes, [0.3, 1.0, 3.0]):
    kernel = ConstantKernel(1.0) * RBF(length_scale=length_scale)
    gp = GaussianProcessRegressor(kernel=kernel, alpha=0.01,
                                  optimizer=None)
    gp.fit(X_train, y_train)
    y_mean, y_std = gp.predict(X_test, return_std=True)

    ax.scatter(X_train, y_train, c='red', zorder=5)
    ax.plot(X_test, y_mean, 'b-')
    ax.fill_between(X_test.ravel(),
                    y_mean - 2*y_std, y_mean + 2*y_std,
                    alpha=0.2, color='blue')
    ax.set_title(f'$\\ell = {length_scale}$')
    ax.set_ylim(-2.5, 2.5)

plt.tight_layout()
plt.savefig('gp_lengthscale.pdf')
```

Échantillonnage du prior GP

```
# Echantillonner des fonctions du prior
X_grid = np.linspace(-5, 5, 200).reshape(-1, 1)
kernel_rbf = RBF(length_scale=1.0)
K_prior = kernel_rbf(X_grid)
K_prior += 1e-8 * np.eye(200)  # stabilite numerique

samples = np.random.multivariate_normal(
    np.zeros(200), K_prior, size=5
)

plt.figure(figsize=(8, 4))
for i, s in enumerate(samples):
    plt.plot(X_grid, s, alpha=0.7, label=f'Sample {i+1}')
plt.xlabel('x'); plt.ylabel('f(x)')
plt.title('Echantillons du prior GP (noyau RBF, $\\ell=1$)')
plt.legend()
plt.savefig('gp_prior_samples.pdf')
```

10.6 Exercices

Exercice 10.1 (Niveau 1 — Conjugaison Beta-Bernoulli)**.** On observe $n = 10$ lancers d'une pièce avec $s = 7$ succès. Le prior est $p \sim \text{Beta}(2, 2)$.

1. Donner la distribution a posteriori.
2. Calculer la moyenne et la variance du postérieur.
3. Comparer avec l'estimateur MLE $\hat{p} = s/n$.
4. Tracer le prior et le postérieur sur un même graphique.

Exercice 10.2 (Niveau 2 — Régression linéaire bayésienne)**.**

1. Dériver les équations (10.12)–(10.13) en détail en partant de la règle de Bayes.
2. Montrer que la distribution prédictive (10.19) est gaussienne et dériver σ_*^2.
3. Implémenter la régression linéaire bayésienne from scratch. Générer des données avec $y = 1.5x + 0.3 + \varepsilon$ et :
   (a) Tracer l'évolution du postérieur $p(\boldsymbol{w} \mid \mathcal{D})$ après avoir observé $n \in \{0, 1, 5, 20\}$ points.
   (b) Tracer les bandes de confiance prédictives.
   (c) Comparer la moyenne du postérieur avec la solution OLS et Ridge.

Exercice 10.3 (Niveau 3 — Processus gaussiens)**.**

1. Implémenter la régression GP from scratch (équations (10.26)–(10.27)) avec le noyau RBF.
2. Générer $n = 15$ points à partir de $f(x) = \sin(x) + 0.2x$ avec bruit $\sigma_n = 0.2$.

   (a) Tracer la moyenne prédictive et les bandes $\pm 2\sigma$.
   (b) Vérifier que l'incertitude augmente loin des données.

3. Implémenter l'optimisation de la log-vraisemblance marginale (10.30) par descente de gradient pour trouver $(\ell, \sigma_f, \sigma_n)$ optimaux. Comparer avec les résultats de `GaussianProcessReg`

4. (*Bonus*) Comparer les noyaux RBF, Matérn 3/2 et périodique sur des données sinusoïdales. Discuter l'impact du choix du noyau sur les prédictions et l'incertitude.

5. (*Bonus*) Montrer que la régression linéaire bayésienne est un cas particulier du GP avec noyau linéaire $\kappa(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{x}^\top \boldsymbol{S}_0 \boldsymbol{x}'$.

Chapitre 11

Sélection de Modèles et Théorie de l'Apprentissage

Ce chapitre traite de la sélection de modèles (critères d'information, validation croisée), du réglage des hyperparamètres (grid search, random search, optimisation bayésienne), et des fondements théoriques de l'apprentissage : cadre PAC, dimension VC, bornes de généralisation et théorème « no free lunch ».

11.1 Critères de sélection de modèles

11.1.1 Critère d'information d'Akaike (AIC)

Définition 11.1 (AIC)**.** Pour un modèle à k paramètres avec log-vraisemblance maximale $\hat{\ell}$, le **critère d'information d'Akaike** est :

$$\mathrm{AIC} = -2\hat{\ell} + 2k \tag{11.1}$$

On choisit le modèle qui *minimise* l'AIC.

Interprétation de l'AIC

Le terme $-2\hat{\ell}$ mesure l'adéquation aux données : plus la vraisemblance est grande, plus ce terme est petit. Le terme $2k$ pénalise la complexité. L'AIC réalise un compromis : ajouter un paramètre n'est justifié que s'il améliore suffisamment l'ajustement.

Remarque 11.2. L'AIC est dérivé d'une approximation asymptotique de la divergence de Kullback-Leibler $\mathrm{KL}(p_{\text{vrai}} \| p_{\hat{\theta}})$ entre la distribution réelle et le modèle estimé. Il est asymptotiquement équivalent à la validation croisée leave-one-out.

11.1.2 Critère d'information bayésien (BIC)

Définition 11.3 (BIC)**.** Le **critère d'information bayésien** (ou critère de Schwarz) est :

$$\mathrm{BIC} = -2\hat{\ell} + k \ln n \tag{11.2}$$

où n est le nombre d'observations.

Différences AIC vs BIC

- Le BIC pénalise davantage la complexité dès que $\ln n > 2$, c'est-à-dire $n \geq 8$.
- L'AIC tend à sélectionner des modèles plus complexes (meilleure prédiction). Le BIC favorise des modèles plus parcimonieux (consistance : il retrouve le vrai modèle si celui-ci est dans $\mathcal{H}$).
- En pratique, utiliser les deux et comparer.

11.1.3 Validation croisée comme critère de sélection

Validation croisée k-fold pour la sélection

Soit $\mathcal{M}_1, \ldots, \mathcal{M}_p$ des modèles candidats. Pour chaque modèle $\mathcal{M}_j$, calculer :

$$\mathrm{CV}_k(\mathcal{M}_j) = \frac{1}{k} \sum_{i=1}^{k} \hat{R}^{(i)}(\mathcal{M}_j) \tag{11.3}$$

Sélectionner $\mathcal{M}^* = \arg\min_j \mathrm{CV}_k(\mathcal{M}_j)$.

Proposition 11.4 (Règle du « one standard error »)**.** Au lieu de choisir le modèle minimisant CV_k, on peut choisir le modèle le plus parcimonieux dont l'erreur CV est au plus à un écart-type de l'erreur minimale :

$$\mathcal{M}^*_{\mathrm{1SE}} = \text{le plus simple } \mathcal{M}_j \text{ tel que } \mathrm{CV}_k(\mathcal{M}_j) \leq \mathrm{CV}_k(\mathcal{M}^*) + \mathrm{SE}(\mathcal{M}^*) \tag{11.4}$$

Cette règle favorise des modèles plus interprétables.

11.2 Réglage des hyperparamètres

Définition 11.5 (Hyperparamètres)**.** Les **hyperparamètres** sont les paramètres du modèle qui ne sont pas appris par l'algorithme d'entraînement : nombre de voisins k dans k-NN, force de régularisation λ dans la régression Ridge, profondeur maximale d'un arbre de décision, etc.

11.2.1 Recherche sur grille (Grid Search)

Grid Search avec validation croisée

1. Définir une grille $\Lambda = \Lambda_1 \times \cdots \times \Lambda_p$ sur l'espace des hyperparamètres.
2. Pour chaque combinaison $\boldsymbol{\lambda} \in \Lambda$:
   (a) Calculer $\mathrm{CV}_k(\boldsymbol{\lambda})$ par validation croisée.
3. Retourner $\boldsymbol{\lambda}^* = \arg\min_{\boldsymbol{\lambda} \in \Lambda} \mathrm{CV}_k(\boldsymbol{\lambda})$.

Complexité : $O(|\Lambda| \cdot k \cdot T_{\mathrm{fit}})$ où T_{fit} est le coût d'entraînement.

11.2.2 Recherche aléatoire (Random Search)

Théorème 11.6 (Efficacité du Random Search — Bergstra & Bengio, 2012)**.** *Si seule une fraction p des hyperparamètres influence significativement la performance, alors pour atteindre un point dans les 5 % meilleurs avec probabilité $\geq 1-\delta$, le random search nécessite :*

$$n \geq \frac{\ln(1/\delta)}{\ln(1/(1-0.05))} \approx \frac{\ln(1/\delta)}{0.0513} \tag{11.5}$$

tirages, indépendamment de la dimension de l'espace de recherche.

Grid Search vs Random Search

- La recherche sur grille explore systématiquement mais souffre de la **malédiction de la dimensionnalité** : $|\Lambda|$ croît exponentiellement avec le nombre d'hyperparamètres.
- La recherche aléatoire est préférable dès que le nombre d'hyperparamètres dépasse 2–3, car elle couvre mieux chaque axe.

11.2.3 Optimisation bayésienne

Définition 11.7 (Optimisation bayésienne)**.** L'optimisation bayésienne modélise la fonction objectif $f(\boldsymbol{\lambda}) = \mathrm{CV}_k(\boldsymbol{\lambda})$ par un **processus gaussien** (surrogate model), puis sélectionne le prochain point $\boldsymbol{\lambda}_{t+1}$ en maximisant une **fonction d'acquisition** $\alpha(\boldsymbol{\lambda})$.

Optimisation bayésienne

1. Initialiser : évaluer f en n_0 points aléatoires.
2. Répéter pour $t = n_0 + 1, \ldots, T$:
   (a) Ajuster un processus gaussien $\mathcal{GP}$ sur les observations $\{(\boldsymbol{\lambda}_i, f_i)\}_{i=1}^{t-1}$.
   (b) Calculer la moyenne a posteriori $\mu(\boldsymbol{\lambda})$ et la variance $\sigma^2(\boldsymbol{\lambda})$.
   (c) Choisir $\boldsymbol{\lambda}_t = \arg\max_{\boldsymbol{\lambda}} \alpha(\boldsymbol{\lambda})$, par exemple l'**Expected Improvement** :
   $$\mathrm{EI}(\boldsymbol{\lambda}) = \mathbb{E}[\max(f^* - f(\boldsymbol{\lambda}), 0)] \tag{11.6}$$
   où f^* est la meilleure valeur observée.
   (d) Évaluer $f(\boldsymbol{\lambda}_t)$.
3. Retourner $\boldsymbol{\lambda}^* = \arg\min_t f(\boldsymbol{\lambda}_t)$.

11.3 Cadre PAC (Probably Approximately Correct)

11.3.1 Définitions fondamentales

Définition 11.8 (Apprentissage PAC)**.** Une classe d'hypothèses $\mathcal{H}$ est **PAC-apprenable** (*Probably Approximately Correct*) s'il existe un algorithme $\mathcal{A}$ et une fonction $m_{\mathcal{H}} : (0,1)^2 \to$

CHAPITRE 11. SÉLECTION DE MODÈLES ET THÉORIE DE L'APPRENTISSAGE

$\mathbb{N}$ tels que, pour tout $\varepsilon > 0$, tout $\delta > 0$ et toute distribution $\mathcal{D}$ sur $\mathcal{X} \times \{0,1\}$, si $m \geq m_{\mathcal{H}}(\varepsilon, \delta)$, alors avec probabilité au moins $1-\delta$ sur le tirage de $S \sim \mathcal{D}^m$:

$$R(h_S) \leq \min_{h \in \mathcal{H}} R(h) + \varepsilon \tag{11.7}$$

où $R(h) = \mathbb{E}_{(\boldsymbol{x},y)\sim\mathcal{D}}[\mathbb{1}[h(\boldsymbol{x}) \neq y]]$ est le risque réel.

Théorème 11.9 (Borne PAC pour $\mathcal{H}$ fini)**.** *Si $\mathcal{H}$ est fini et l'algorithme retourne l'ERM $h_S = \arg\min_{h\in\mathcal{H}} \hat{R}_S(h)$, alors avec probabilité $\geq 1-\delta$:*

$$R(h_S) \leq \hat{R}_S(h_S) + \sqrt{\frac{\ln|\mathcal{H}| + \ln(1/\delta)}{2m}} \tag{11.8}$$

Esquisse de preuve. Par l'inégalité de Hoeffding, pour tout $h \in \mathcal{H}$ fixé :

$$\Pr\Big[|R(h) - \hat{R}_S(h)| > \varepsilon\Big] \leq 2e^{-2m\varepsilon^2} \tag{11.9}$$

Par un argument d'union bound sur les $|\mathcal{H}|$ hypothèses :

$$\Pr\Big[\exists h \in \mathcal{H} : |R(h) - \hat{R}_S(h)| > \varepsilon\Big] \leq 2|\mathcal{H}|\, e^{-2m\varepsilon^2} \tag{11.10}$$

En posant $\delta = 2|\mathcal{H}|\, e^{-2m\varepsilon^2}$ et en résolvant pour ε, on obtient (11.8). □

11.4 Dimension de Vapnik-Chervonenkis

11.4.1 Pulvérisation et dimension VC

Définition 11.10 (Pulvérisation (Shattering))**.** Un ensemble de points $\{x_1, \ldots, x_m\} \subset \mathcal{X}$ est **pulvérisé** par $\mathcal{H}$ si, pour toute affectation de labels $(y_1, \ldots, y_m) \in \{0,1\}^m$, il existe $h \in \mathcal{H}$ tel que $h(x_i) = y_i$ pour tout i. Autrement dit, $\mathcal{H}$ réalise les 2^m dichotomies possibles.

Définition 11.11 (Dimension VC)**.** La **dimension de Vapnik-Chervonenkis** de $\mathcal{H}$, notée $\mathrm{VCdim}(\mathcal{H})$, est la taille maximale d'un ensemble pulvérisé par $\mathcal{H}$:

$$\mathrm{VCdim}(\mathcal{H}) = \max\{m : \exists S \subset \mathcal{X},\ |S| = m,\ \mathcal{H} \text{ pulvérise } S\} \tag{11.11}$$

11.4.2 Exemples de dimension VC

Proposition 11.12 (VC des classifieurs linéaires)**.** La dimension VC des classifieurs linéaires $\mathcal{H} = \{x \mapsto \mathrm{sign}(\boldsymbol{w}^\top \boldsymbol{x} + b) : \boldsymbol{w} \in \mathbb{R}^d, b \in \mathbb{R}\}$ dans $\mathbb{R}^d$ est :

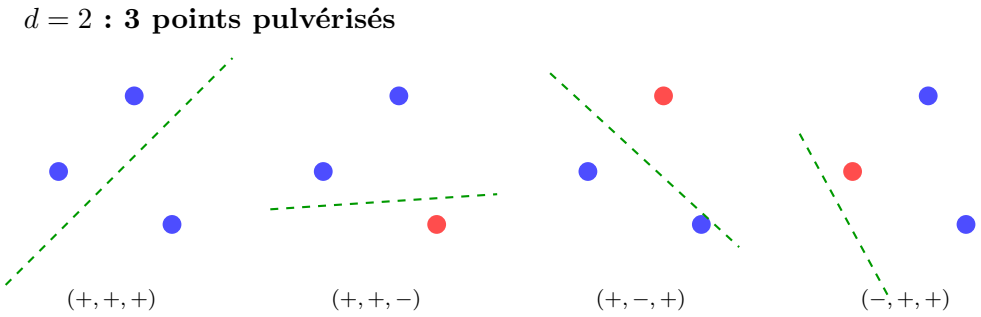
$$\mathrm{VCdim}(\mathcal{H}_{\text{lin}}) = d + 1 \tag{11.12}$$

Esquisse de preuve. **(i)** On montre que $d+1$ points en position générale sont pulvérisés. Prendre les d vecteurs de la base canonique plus l'origine ; toute dichotomie est réalisable par un hyperplan bien choisi.

(ii) On montre qu'aucun ensemble de $d+2$ points ne peut être pulvérisé. Par le théorème de Radon, tout ensemble de $d+2$ points dans $\mathbb{R}^d$ peut être partitionné en deux sous-ensembles dont les enveloppes convexes s'intersectent, ce qui interdit une séparation linéaire. □

Exemple 11.13 (Intervalles sur $\mathbb{R}$)**.** Pour $\mathcal{H} = \{\mathbb{1}[a \leq x \leq b] : a \leq b\}$ sur $\mathbb{R}$, $\mathrm{VCdim}(\mathcal{H}) = 2$. En effet, deux points bien ordonnés sont pulvérisés, mais trois points $x_1 < x_2 < x_3$ ne peuvent réaliser la dichotomie $(1,0,1)$ par un seul intervalle.

11.4.3 Diagramme : pulvérisation par un classifieur linéaire



11.5 Bornes de généralisation

Théorème 11.14 (Borne VC — Vapnik & Chervonenkis)**.** *Soit $\mathcal{H}$ une classe de dimension VC finie d_{VC}. Pour tout $\delta > 0$, avec probabilité $\geq 1 - \delta$ sur un échantillon i.i.d. de taille m :*

$$R(h_S) \leq \hat{R}_S(h_S) + \sqrt{\frac{d_{VC}\left(\ln \frac{2m}{d_{VC}} + 1\right) + \ln \frac{4}{\delta}}{m}} \tag{11.13}$$

pour tout $h_S \in \mathcal{H}$.

Esquisse de preuve. La preuve repose sur trois ingrédients :

1. **Symétrisation :** on remplace la déviation $|R(h) - \hat{R}_S(h)|$ par la déviation entre deux échantillons fantômes (*ghost sample*).
2. **Lemme de Sauer-Shelah :** le nombre de dichotomies réalisables par $\mathcal{H}$ sur m points est borné par :

$$\Pi_{\mathcal{H}}(m) \leq \sum_{i=0}^{d_{\mathrm{VC}}} \binom{m}{i} \leq \left(\frac{em}{d_{\mathrm{VC}}}\right)^{d_{\mathrm{VC}}} \tag{11.14}$$

3. **Inégalité de concentration :** on applique Hoeffding sur les variables de Rademacher, puis on contrôle l'union sur les dichotomies effectives.

□

Lemme 11.15 (Sauer-Shelah)**.** *Si $VCdim(\mathcal{H}) = d$, alors la* ***fonction de croissance*** $\Pi_{\mathcal{H}}(m) = \max_{S \subset \mathcal{X}, |S|=m} |\{(h(x_1), \ldots, h(x_m)) : h \in \mathcal{H}\}|$ *satisfait :*

$$\Pi_{\mathcal{H}}(m) \leq \sum_{i=0}^{d} \binom{m}{i} \tag{11.15}$$

Interprétation de la borne VC

La borne VC dit que l'écart de généralisation décroît en $O\left(\sqrt{d_{\mathrm{VC}}/m}\right)$. Pour garantir une erreur de généralisation $\leq \varepsilon$, il faut un nombre d'exemples $m = O(d_{\mathrm{VC}}/\varepsilon^2)$, proportionnel à la complexité de la classe d'hypothèses.

11.6 Théorème « No Free Lunch »

Théorème 11.16 (No Free Lunch — Wolpert, 1996)**.** *Soit $\mathcal{A}$ un algorithme d'apprentissage quelconque et $\mathcal{F}$ l'ensemble de toutes les fonctions $\{0,1\}^d \to \{0,1\}$. Pour tout algorithme $\mathcal{A}$, il existe une distribution $\mathcal{D}$ telle que :*

1. *Il existe une fonction $f \in \mathcal{F}$ avec $R_{\mathcal{D}}(f) = 0$.*
2. *Avec probabilité $\geq 1/7$ sur le tirage de S de taille $m \leq |\mathcal{X}|/2$:*

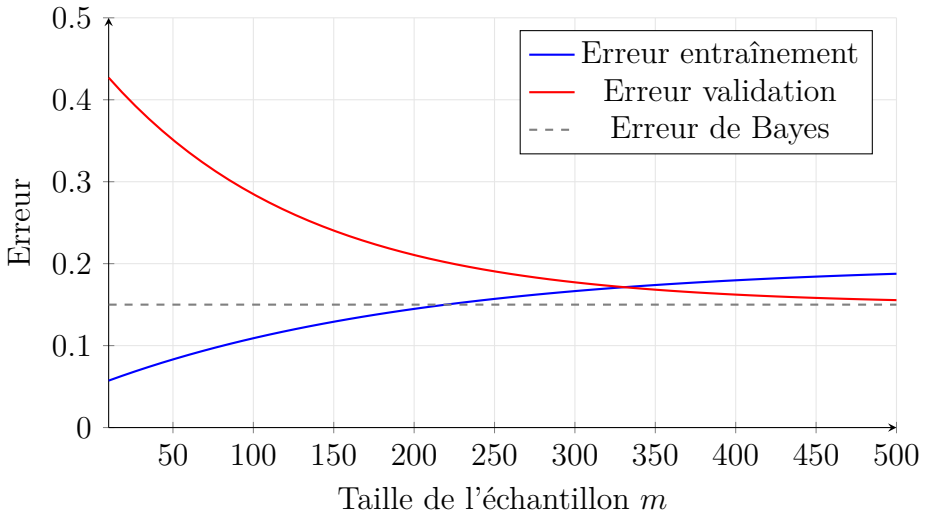
$$R_{\mathcal{D}}(\mathcal{A}(S)) \geq \frac{1}{8} \tag{11.16}$$

Conséquences du No Free Lunch

- Aucun algorithme n'est universellement meilleur que les autres *sur toutes les distributions*.
- Les bonnes performances d'un algorithme reposent sur des **hypothèses implicites** sur la distribution des données (régularité, parcimonie, etc.).
- La sélection de modèles est donc *fondamentale* : il faut choisir la classe $\mathcal{H}$ adaptée au problème.

11.7 Courbes d'apprentissage

Définition 11.17 (Courbes d'apprentissage)**.** Les **courbes d'apprentissage** représentent l'erreur d'entraînement et l'erreur de validation en fonction de la taille m de l'échantillon d'entraînement.



Diagnostic par les courbes d'apprentissage

- **Biais élevé (underfitting)** : les deux courbes convergent vers une erreur élevée. Ajouter des données ne servira pas ; il faut augmenter la complexité du modèle.

- **Variance élevée (overfitting)** : grand écart entre les courbes. Ajouter des données ou régulariser peut aider.
- **Bon compromis** : les courbes convergent vers une erreur proche de l'erreur de Bayes.

11.8 Implémentation : pipeline de sélection de modèles

Comparaison Grid Search, Random Search et courbes d'apprentissage

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import (
    train_test_split, GridSearchCV, RandomizedSearchCV,
    learning_curve
)
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from scipy.stats import uniform, randint

# Charger les donnees
X, y = load_breast_cancer(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

# --- Grid Search sur SVM ---
pipe_svm = Pipeline([
    ('scaler', StandardScaler()),
    ('svm', SVC())
])
param_grid = {
    'svm__C': [0.1, 1, 10, 100],
    'svm__gamma': ['scale', 0.01, 0.001],
    'svm__kernel': ['rbf', 'poly']
}
grid = GridSearchCV(pipe_svm, param_grid, cv=5,
                    scoring='accuracy', n_jobs=-1)
grid.fit(X_train, y_train)
print(f"Grid Search - Meilleurs params: {grid.best_params_}")
print(f"Grid Search - Accuracy CV: {grid.best_score_:.4f}")
print(f"Grid Search - Accuracy test: "
      f"{grid.score(X_test, y_test):.4f}")
```

Sortie

```
Grid Search - Meilleurs params: {'svm__C': 10,
    'svm__gamma': 'scale', 'svm__kernel': 'rbf'}
Grid Search - Accuracy CV: 0.9780
Grid Search - Accuracy test: 0.9825
```

Random Search et courbes d'apprentissage

```python
# --- Random Search sur Random Forest ---
pipe_rf = Pipeline([
    ('scaler', StandardScaler()),
    ('rf', RandomForestClassifier(random_state=42))
])
param_dist = {
    'rf__n_estimators': randint(50, 300),
    'rf__max_depth': [None, 5, 10, 20, 30],
    'rf__min_samples_split': randint(2, 20),
    'rf__max_features': ['sqrt', 'log2']
}
random_search = RandomizedSearchCV(
    pipe_rf, param_dist, n_iter=50, cv=5,
    scoring='accuracy', random_state=42, n_jobs=-1
)
random_search.fit(X_train, y_train)
print(f"Random Search - Accuracy CV: "
      f"{random_search.best_score_:.4f}")
print(f"Random Search - Accuracy test: "
      f"{random_search.score(X_test, y_test):.4f}")

# --- Courbes d'apprentissage ---
train_sizes, train_scores, val_scores = learning_curve(
    grid.best_estimator_, X_train, y_train,
    train_sizes=np.linspace(0.1, 1.0, 10),
    cv=5, scoring='accuracy', n_jobs=-1
)
plt.figure(figsize=(8, 5))
plt.plot(train_sizes, train_scores.mean(axis=1),
         'o-', label="Entrainement")
plt.plot(train_sizes, val_scores.mean(axis=1),
         's-', label="Validation")
plt.xlabel("Taille d'entrainement")
plt.ylabel("Accuracy")
plt.title("Courbes d'apprentissage (SVM)")
plt.legend()
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.savefig("learning_curves.pdf")
plt.show()
```

Sortie

```
Random Search - Accuracy CV: 0.9648
Random Search - Accuracy test: 0.9649
```

11.9 Exercices

Exercice 11.1 (Critères d'information — niveau 1)**.** On compare trois modèles de régression linéaire sur $n = 100$ observations :

Modèle	k (paramètres)	$\hat{\ell}$ (log-vraisemblance)
$\mathcal{M}_1$	3	-120
$\mathcal{M}_2$	5	-115
$\mathcal{M}_3$	10	-110

1. Calculer l'AIC et le BIC pour chaque modèle.
2. Quel modèle est sélectionné par chaque critère ?
3. Commenter les différences.

Exercice 11.2 (Dimension VC — niveau 2)**.** 1. Montrer que $\mathrm{VCdim}(\mathcal{H}) = 2$ pour la classe des classifieurs par intervalle $\mathcal{H} = \{\mathbb{1}[a \leq x \leq b] : a, b \in \mathbb{R}\}$.

2. Soit $\mathcal{H}_r = \{h_{a,r} : \boldsymbol{x} \mapsto \mathbb{1}[\|\boldsymbol{x} - \boldsymbol{a}\| \leq r]\}$ la classe des boules dans $\mathbb{R}^d$. Montrer que $\mathrm{VCdim}(\mathcal{H}_r) = d + 1$.
3. Déterminer la dimension VC de la classe des rectangles alignés sur les axes dans $\mathbb{R}^2$.

Exercice 11.3 (Pipeline complet de sélection — niveau 3)**.** 1. Charger le jeu de données `digits` de scikit-learn.

2. Comparer au moins 4 classifieurs (k-NN, SVM, Random Forest, régression logistique) en utilisant `GridSearchCV` ou `RandomizedSearchCV` pour chacun.
3. Pour le meilleur modèle, tracer les courbes d'apprentissage et diagnostiquer biais/variance.
4. Implémenter une optimisation bayésienne simple avec `scikit-optimize` (`BayesSearchCV`) et comparer le nombre d'évaluations nécessaires.
5. Calculer la borne de généralisation VC pour un classifieur linéaire sur ce problème ($d = 64$ features). La borne est-elle informative avec $n = 1797$ exemples ? Discuter.

Chapitre 12

Méthodes à Noyaux

*Ce chapitre présente les méthodes à noyaux : l'astuce du noyau (*kernel trick*), le théorème de Mercer, les noyaux classiques, la régression Ridge à noyau, l'ACP à noyau, le k-means à noyau, les espaces de Hilbert à noyau reproduisant (RKHS) et le théorème du représentant. Nous concluons par les liens avec les processus gaussiens.*

12.1 Applications de features et astuce du noyau

12.1.1 Motivation : séparabilité en grande dimension

Exemple 12.1 (XOR non linéairement séparable)**.** Considérons les points $\{(\pm 1, \pm 1)\}$ dans $\mathbb{R}^2$ avec étiquettes $y = x_1 \cdot x_2$. Aucun hyperplan dans $\mathbb{R}^2$ ne peut les séparer, mais en ajoutant la feature $\phi_3(\boldsymbol{x}) = x_1 x_2$, les données deviennent linéairement séparables dans $\mathbb{R}^3$.

Définition 12.2 (Application de features)**.** Une **application de features** (ou *feature map*) est une fonction $\phi : \mathcal{X} \to \mathcal{F}$ qui envoie les données d'un espace d'entrée $\mathcal{X}$ vers un espace de features $\mathcal{F}$ (possiblement de dimension infinie).

12.1.2 Dérivation formelle de l'astuce du noyau

De nombreux algorithmes (SVM, Ridge, PCA, k-means) ne dépendent des données qu'à travers des **produits scalaires** $\langle \boldsymbol{x}_i, \boldsymbol{x}_j \rangle$. Si l'on travaille dans l'espace des features, on remplace ces produits par $\langle \phi(\boldsymbol{x}_i), \phi(\boldsymbol{x}_j) \rangle_{\mathcal{F}}$, ce qui peut être coûteux si $\dim(\mathcal{F})$ est grand.

Définition 12.3 (Fonction noyau)**.** Une **fonction noyau** est une fonction $\kappa : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ telle qu'il existe un espace de Hilbert $\mathcal{F}$ et une application $\phi : \mathcal{X} \to \mathcal{F}$ vérifiant :

$$\kappa(\boldsymbol{x}, \boldsymbol{x}') = \langle \phi(\boldsymbol{x}), \phi(\boldsymbol{x}') \rangle_{\mathcal{F}} \tag{12.1}$$

L'astuce du noyau

Au lieu de :

1. calculer explicitement $\phi(\boldsymbol{x}_i)$ pour tout i,
2. former les produits scalaires $\langle \phi(\boldsymbol{x}_i), \phi(\boldsymbol{x}_j) \rangle$,

on évalue directement $\kappa(\boldsymbol{x}_i, \boldsymbol{x}_j)$, ce qui évite de travailler dans l'espace $\mathcal{F}$ (qui peut être de dimension infinie).
Matrice de Gram : $\boldsymbol{K} \in \mathbb{R}^{n \times n}$ avec $K_{ij} = \kappa(\boldsymbol{x}_i, \boldsymbol{x}_j)$.

Exemple 12.4 (Noyau polynomial — feature map explicite)**.** Pour $\boldsymbol{x}, \boldsymbol{x}' \in \mathbb{R}^2$ et $\kappa(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x}^\top \boldsymbol{x}')^2$:

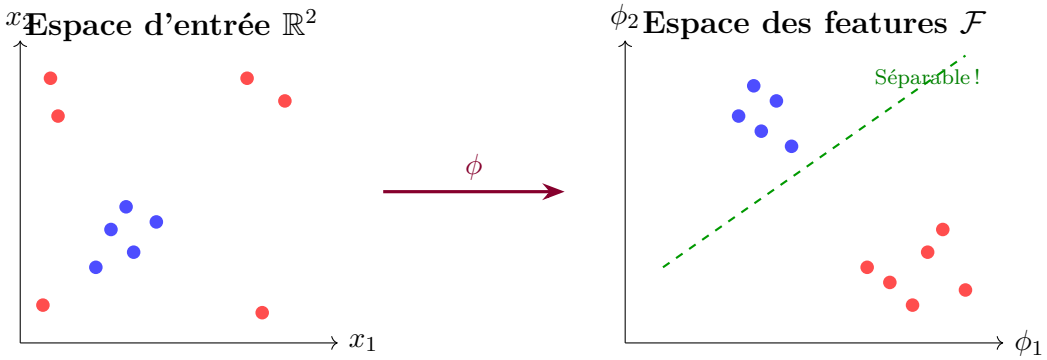
$$(\boldsymbol{x}^\top \boldsymbol{x}')^2 = (x_1 x_1' + x_2 x_2')^2 \tag{12.2}$$

$$= x_1^2 {x_1'}^2 + 2 x_1 x_2 x_1' x_2' + x_2^2 {x_2'}^2 \tag{12.3}$$

$$= \langle \phi(\boldsymbol{x}), \phi(\boldsymbol{x}') \rangle \tag{12.4}$$

où $\phi(\boldsymbol{x}) = (x_1^2, \sqrt{2}\, x_1 x_2, x_2^2)^\top \in \mathbb{R}^3$. Le noyau calcule le produit scalaire dans $\mathbb{R}^3$ sans construire ϕ explicitement.

12.1.3 Diagramme : projection dans l'espace des features



12.2 Théorème de Mercer et noyaux classiques

Théorème 12.5 (Mercer)**.** *Soit $\kappa : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ une fonction continue et symétrique sur un compact $\mathcal{X} \subset \mathbb{R}^d$. κ est un **noyau défini positif** si et seulement si, pour tout échantillon fini $\{x_1, \ldots, x_n\} \subset \mathcal{X}$, la matrice de Gram $\boldsymbol{K}$ avec $K_{ij} = \kappa(x_i, x_j)$ est semi-définie positive :*

$$\forall \boldsymbol{c} \in \mathbb{R}^n, \quad \sum_{i=1}^{n} \sum_{j=1}^{n} c_i c_j \, \kappa(x_i, x_j) \geq 0 \tag{12.5}$$

De plus, κ admet une décomposition :

$$\kappa(\boldsymbol{x}, \boldsymbol{x}') = \sum_{k=1}^{\infty} \lambda_k \, \psi_k(\boldsymbol{x}) \, \psi_k(\boldsymbol{x}') \tag{12.6}$$

où $\lambda_k \geq 0$ sont les valeurs propres et $\{\psi_k\}$ les fonctions propres de l'opérateur intégral associé à κ.

Remarque 12.6. La décomposition de Mercer généralise la décomposition spectrale des matrices symétriques définies positives au cadre fonctionnel. Elle justifie l'existence de ϕ dans (12.1) : on peut poser $\phi(\boldsymbol{x}) = (\sqrt{\lambda_1}\, \psi_1(\boldsymbol{x}), \sqrt{\lambda_2}\, \psi_2(\boldsymbol{x}), \ldots)$.

12.2.1 Noyaux classiques

Noyaux couramment utilisés

Noyau	Expression	Paramètres
Linéaire	$\kappa(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{x}^\top \boldsymbol{x}'$	—
Polynomial	$\kappa(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x}^\top \boldsymbol{x}' + c)^p$	$c \geq 0$, $p \in \mathbb{N}^*$
RBF (Gaussien)	$\kappa(\boldsymbol{x}, \boldsymbol{x}') = \exp\left(-\frac{\|\boldsymbol{x} - \boldsymbol{x}'\|^2}{2\sigma^2}\right)$	$\sigma > 0$
Laplacien	$\kappa(\boldsymbol{x}, \boldsymbol{x}') = \exp\left(-\frac{\|\boldsymbol{x} - \boldsymbol{x}'\|_1}{\sigma}\right)$	$\sigma > 0$

Proposition 12.7 (Dimension de l'espace des features du noyau RBF)**.** Le noyau RBF correspond à un espace de features $\mathcal{F}$ de **dimension infinie**. En effet, par le développement en série de Taylor de l'exponentielle :

$$\exp\left(-\frac{\|\boldsymbol{x} - \boldsymbol{x}'\|^2}{2\sigma^2}\right) = e^{-\frac{\|\boldsymbol{x}\|^2}{2\sigma^2}} e^{-\frac{\|\boldsymbol{x}'\|^2}{2\sigma^2}} \sum_{k=0}^{\infty} \frac{(\boldsymbol{x}^\top \boldsymbol{x}')^k}{k!\,\sigma^{2k}} \tag{12.7}$$

Chaque terme $(\boldsymbol{x}^\top \boldsymbol{x}')^k$ engendre un espace de dimension $\binom{d+k-1}{k}$, et la somme infinie produit un espace de dimension infinie.

Choix du noyau

- Le **noyau linéaire** est adapté lorsque $d \gg n$ (texte, génomique).
- Le **noyau RBF** est un bon choix par défaut ; le paramètre σ (ou $\gamma = 1/(2\sigma^2)$) contrôle la « portée » du noyau.
- Le **noyau polynomial** capture les interactions entre features jusqu'à l'ordre p.

12.3 Régression Ridge à noyau

Définition 12.8 (Kernel Ridge Regression)**.** La **régression Ridge à noyau** résout :

$$\min_{f \in \mathcal{H}_\kappa} \frac{1}{n} \sum_{i=1}^{n} (y_i - f(\boldsymbol{x}_i))^2 + \lambda \|f\|_{\mathcal{H}_\kappa}^2 \tag{12.8}$$

Par le théorème du représentant (voir §12.7), la solution s'écrit $f^*(\boldsymbol{x}) = \sum_{i=1}^{n} \alpha_i \, \kappa(\boldsymbol{x}_i, \boldsymbol{x})$ avec :

$$\boldsymbol{\alpha}^* = (\boldsymbol{K} + \lambda n \, \boldsymbol{I})^{-1} \boldsymbol{y} \tag{12.9}$$

où $\boldsymbol{K}$ est la matrice de Gram, $K_{ij} = \kappa(\boldsymbol{x}_i, \boldsymbol{x}_j)$.

Remarque 12.9. Le coût de calcul est $O(n^3)$ pour l'inversion matricielle et $O(n^2 d)$ pour le calcul de $\boldsymbol{K}$. Pour n grand, des approximations comme Nyström ou les features aléatoires de Rahimi-Recht sont utilisées.

12.4 ACP à noyau (Kernel PCA)

Définition 12.10 (Kernel PCA)**.** L'**ACP à noyau** effectue une analyse en composantes principales dans l'espace des features $\mathcal{F}$. Les composantes principales sont les vecteurs propres de la matrice de covariance dans $\mathcal{F}$:

$$\boldsymbol{C}_\phi = \frac{1}{n}\sum_{i=1}^{n} \phi(\boldsymbol{x}_i)\,\phi(\boldsymbol{x}_i)^\top \tag{12.10}$$

Proposition 12.11 (Calcul via la matrice de Gram)**.** Les projections sur les composantes principales dans $\mathcal{F}$ se calculent à partir de la décomposition spectrale de la matrice de Gram centrée $\tilde{\boldsymbol{K}}$:

$$\tilde{\boldsymbol{K}} = \boldsymbol{K} - \mathbf{1}_n\boldsymbol{K} - \boldsymbol{K}\mathbf{1}_n + \mathbf{1}_n\boldsymbol{K}\mathbf{1}_n \tag{12.11}$$

où $\mathbf{1}_n = \frac{1}{n}\mathbf{1}\mathbf{1}^\top$. Si $\tilde{\boldsymbol{K}}\boldsymbol{v}_k = n\mu_k\boldsymbol{v}_k$, la projection du point $\boldsymbol{x}$ sur la k-ième composante est :

$$z_k(\boldsymbol{x}) = \sum_{i=1}^{n} (v_k)_i\, \kappa(\boldsymbol{x}_i, \boldsymbol{x}) \tag{12.12}$$

(après centrage dans $\mathcal{F}$).

12.5 k-means à noyau

Définition 12.12 (Kernel k-means)**.** Le **k-means à noyau** minimise :

$$J = \sum_{c=1}^{C}\sum_{i\in\mathcal{C}_c} \left\|\phi(\boldsymbol{x}_i) - \boldsymbol{\mu}_c^\phi\right\|_{\mathcal{F}}^2 \tag{12.13}$$

où $\boldsymbol{\mu}_c^\phi = \frac{1}{|\mathcal{C}_c|}\sum_{j\in\mathcal{C}_c}\phi(\boldsymbol{x}_j)$ est le centroïde du cluster c dans $\mathcal{F}$.

Calcul des distances dans $\mathcal{F}$

La distance au carré entre $\phi(\boldsymbol{x}_i)$ et $\boldsymbol{\mu}_c^\phi$ s'exprime uniquement en termes de noyaux :

$$\left\|\phi(\boldsymbol{x}_i) - \boldsymbol{\mu}_c^\phi\right\|^2 = \kappa(\boldsymbol{x}_i,\boldsymbol{x}_i) - \frac{2}{|\mathcal{C}_c|}\sum_{j\in\mathcal{C}_c}\kappa(\boldsymbol{x}_i,\boldsymbol{x}_j) + \frac{1}{|\mathcal{C}_c|^2}\sum_{j,l\in\mathcal{C}_c}\kappa(\boldsymbol{x}_j,\boldsymbol{x}_l) \tag{12.14}$$

L'algorithme itère affectation et mise à jour comme le k-means classique, mais dans $\mathcal{F}$.

12.6 Espaces de Hilbert à noyau reproduisant (RKHS)

12.6.1 Définition et intuition

Définition 12.13 (RKHS)**.** Un **espace de Hilbert à noyau reproduisant** $\mathcal{H}_\kappa$ associé à un noyau défini positif κ est un espace de Hilbert de fonctions $f : \mathcal{X} \to \mathbb{R}$ satisfaisant :

1. **Feature map canonique :** pour tout $\boldsymbol{x} \in \mathcal{X}$, la fonction $\kappa(\boldsymbol{x}, \cdot) \in \mathcal{H}_\kappa$.

2. **Propriété de reproduction :** pour tout $f \in \mathcal{H}_\kappa$ et tout $\boldsymbol{x} \in \mathcal{X}$:

$$f(\boldsymbol{x}) = \langle f, \kappa(\boldsymbol{x}, \cdot)\rangle_{\mathcal{H}_\kappa} \tag{12.15}$$

En particulier, $\kappa(\boldsymbol{x}, \boldsymbol{x}') = \langle \kappa(\boldsymbol{x}, \cdot), \kappa(\boldsymbol{x}', \cdot)\rangle_{\mathcal{H}_\kappa}$.

Interprétation du RKHS

- Le RKHS est l'espace de *toutes les fonctions* que l'on peut construire comme combinaisons linéaires (et limites) de $\kappa(\boldsymbol{x}, \cdot)$, appelées « features canoniques ».
- La norme $\|f\|_{\mathcal{H}_\kappa}$ mesure la **régularité** de f : les fonctions à petite norme sont « lisses » au sens du noyau.
- Évaluer f en un point $\boldsymbol{x}$ revient à projeter f sur $\kappa(\boldsymbol{x}, \cdot)$: c'est la propriété de reproduction.

12.6.2 Construction formelle

Proposition 12.14 (Construction du RKHS)**.** Le RKHS $\mathcal{H}_\kappa$ est la complétion de l'espace pré-hilbertien :

$$\mathcal{H}_0 = \left\{ f = \sum_{i=1}^{m} \alpha_i \, \kappa(\boldsymbol{x}_i, \cdot) : m \in \mathbb{N},\ \alpha_i \in \mathbb{R},\ \boldsymbol{x}_i \in \mathcal{X} \right\} \tag{12.16}$$

muni du produit scalaire :

$$\left\langle \sum_i \alpha_i \, \kappa(\boldsymbol{x}_i, \cdot),\ \sum_j \beta_j \, \kappa(\boldsymbol{x}'_j, \cdot) \right\rangle_{\mathcal{H}_\kappa} = \sum_{i,j} \alpha_i \beta_j \, \kappa(\boldsymbol{x}_i, \boldsymbol{x}'_j) \tag{12.17}$$

12.7 Théorème du représentant

Théorème 12.15 (Théorème du représentant — Kimeldorf & Wahba, 1971)**.** *Soit* $\Omega : \mathbb{R}_+ \to \mathbb{R}$ *une fonction croissante et* $L : (\mathcal{X} \times \mathbb{R} \times \mathbb{R})^n \to \mathbb{R} \cup \{+\infty\}$ *une fonction de perte quelconque. Le problème d'optimisation :*

$$\min_{f \in \mathcal{H}_\kappa} L((\boldsymbol{x}_1, y_1, f(\boldsymbol{x}_1)), \ldots, (\boldsymbol{x}_n, y_n, f(\boldsymbol{x}_n))) + \Omega(\|f\|^2_{\mathcal{H}_\kappa}) \tag{12.18}$$

admet un minimiseur de la forme :

$$f^*(\boldsymbol{x}) = \sum_{i=1}^{n} \alpha_i \, \kappa(\boldsymbol{x}_i, \boldsymbol{x}) \tag{12.19}$$

pour certains coefficients $\alpha_1, \ldots, \alpha_n \in \mathbb{R}$.

Esquisse de preuve. Décomposons $f \in \mathcal{H}_\kappa$ en une partie dans le sous-espace engendré par $\{\kappa(\boldsymbol{x}_i, \cdot)\}_{i=1}^n$ et son complément orthogonal :

$$f = \underbrace{\sum_{i=1}^{n} \alpha_i \, \kappa(\boldsymbol{x}_i, \cdot)}_{f_\parallel} + \underbrace{f_\perp}_{f_\perp \perp \operatorname{span}\{\kappa(\boldsymbol{x}_i, \cdot)\}} \tag{12.20}$$

Par la propriété de reproduction, $f(\boldsymbol{x}_j) = f_\parallel(\boldsymbol{x}_j) + \langle f_\perp, \kappa(\boldsymbol{x}_j, \cdot)\rangle = f_\parallel(\boldsymbol{x}_j)$, donc $f_\perp$ n'affecte pas le terme de perte L. Par ailleurs :

$$\|f\|_{\mathcal{H}_\kappa}^2 = \|f_\parallel\|^2 + \|f_\perp\|^2 \geq \|f_\parallel\|^2 \tag{12.21}$$

Puisque Ω est croissante, ajouter $f_\perp \neq 0$ augmente la pénalité sans diminuer la perte. Donc le minimiseur vérifie $f_\perp = 0$. □

Portée du théorème du représentant

Ce théorème est fondamental car il transforme un problème d'optimisation en **dimension infinie** (minimiser sur $\mathcal{H}_\kappa$) en un problème en **dimension finie** (trouver $\boldsymbol{\alpha} \in \mathbb{R}^n$). C'est ce qui rend les méthodes à noyaux calculatoirement réalisables.

12.8 Connexions avec les processus gaussiens

Définition 12.16 (Processus gaussien)**.** Un **processus gaussien** (GP) est une collection de variables aléatoires dont toute sous-collection finie suit une distribution gaussienne multivariée. Un GP est entièrement défini par sa fonction moyenne $\mu(\boldsymbol{x}) = \mathbb{E}[f(\boldsymbol{x})]$ et sa fonction de covariance $\kappa(\boldsymbol{x}, \boldsymbol{x}') = \mathrm{Cov}[f(\boldsymbol{x}), f(\boldsymbol{x}')]$.

Proposition 12.17 (Lien GP / Kernel Ridge Regression)**.** La moyenne a posteriori du GP avec prior $f \sim \mathcal{GP}(0, \kappa)$ et vraisemblance gaussienne $y_i|f \sim \mathcal{N}(f(\boldsymbol{x}_i), \sigma^2)$ coïncide avec la solution de la régression Ridge à noyau :

$$\mathbb{E}[f(\boldsymbol{x})|\boldsymbol{y}] = \boldsymbol{k}(\boldsymbol{x})^\top (\boldsymbol{K} + \sigma^2 \boldsymbol{I})^{-1} \boldsymbol{y} \tag{12.22}$$

où $k(\boldsymbol{x})_i = \kappa(\boldsymbol{x}_i, \boldsymbol{x})$ et $\lambda = \sigma^2/n$ dans (12.9).

Remarque 12.18. Le GP fournit en plus une **incertitude prédictive** :

$$\mathrm{Var}[f(\boldsymbol{x})|\boldsymbol{y}] = \kappa(\boldsymbol{x}, \boldsymbol{x}) - \boldsymbol{k}(\boldsymbol{x})^\top (\boldsymbol{K} + \sigma^2 \boldsymbol{I})^{-1} \boldsymbol{k}(\boldsymbol{x}) \tag{12.23}$$

C'est un avantage majeur des GP par rapport à la Kernel Ridge Regression : on obtient non seulement une prédiction ponctuelle, mais aussi une barre d'erreur.

12.9 Implémentation : méthodes à noyaux avec scikit-learn

SVM à noyau, Kernel Ridge et Kernel PCA

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_moons, make_circles
from sklearn.svm import SVC
from sklearn.kernel_ridge import KernelRidge
from sklearn.decomposition import KernelPCA
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import GridSearchCV
```

```python
from sklearn.pipeline import Pipeline

# --- Donnees non lineairement separables ---
X, y = make_moons(n_samples=300, noise=0.2, random_state=42)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# --- SVM avec differents noyaux ---
fig, axes = plt.subplots(1, 3, figsize=(15, 4))
kernels = ['linear', 'poly', 'rbf']
for ax, kernel in zip(axes, kernels):
    svm = SVC(kernel=kernel, degree=3, gamma='scale', C=1.0)
    svm.fit(X_scaled, y)
    # Grille de decision
    xx, yy = np.meshgrid(
        np.linspace(X_scaled[:,0].min()-0.5,
                    X_scaled[:,0].max()+0.5, 200),
        np.linspace(X_scaled[:,1].min()-0.5,
                    X_scaled[:,1].max()+0.5, 200)
    )
    Z = svm.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    ax.contourf(xx, yy, Z, alpha=0.3, cmap='coolwarm')
    ax.scatter(X_scaled[:,0], X_scaled[:,1], c=y,
               cmap='coolwarm', edgecolors='k', s=20)
    acc = svm.score(X_scaled, y)
    ax.set_title(f"{kernel} (acc={acc:.2f})")
plt.tight_layout()
plt.savefig("svm_kernels.pdf")
plt.show()
```

Kernel PCA : réduction non linéaire

```python
# --- Kernel PCA sur donnees circulaires ---
X_circ, y_circ = make_circles(n_samples=400, factor=0.3,
                              noise=0.1, random_state=42)

fig, axes = plt.subplots(1, 3, figsize=(15, 4))

# PCA lineaire
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_circ)
axes[0].scatter(X_pca[:,0], X_pca[:,1], c=y_circ,
                cmap='coolwarm', s=15)
axes[0].set_title("PCA lineaire")

# Kernel PCA avec noyau RBF
kpca_rbf = KernelPCA(n_components=2, kernel='rbf', gamma=5)
X_kpca_rbf = kpca_rbf.fit_transform(X_circ)
```

```
axes[1].scatter(X_kpca_rbf[:,0], X_kpca_rbf[:,1],
                c=y_circ, cmap='coolwarm', s=15)
axes[1].set_title("Kernel PCA (RBF, $\\gamma=5$)")

# Kernel PCA avec noyau polynomial
kpca_poly = KernelPCA(n_components=2, kernel='poly',
                      degree=3, gamma=1)
X_kpca_poly = kpca_poly.fit_transform(X_circ)
axes[2].scatter(X_kpca_poly[:,0], X_kpca_poly[:,1],
                c=y_circ, cmap='coolwarm', s=15)
axes[2].set_title("Kernel PCA (poly, $p=3$)")

plt.tight_layout()
plt.savefig("kernel_pca.pdf")
plt.show()
```

Kernel Ridge Regression

```
# --- Kernel Ridge Regression sur donnees 1D ---
np.random.seed(42)
X_train = np.sort(np.random.uniform(-3, 3, 50))[:, None]
y_train = np.sin(X_train).ravel() + 0.3 * np.random.randn(50)
X_plot = np.linspace(-3.5, 3.5, 300)[:, None]

fig, axes = plt.subplots(1, 3, figsize=(15, 4))
gammas = [0.5, 2.0, 10.0]

for ax, gamma in zip(axes, gammas):
    krr = KernelRidge(kernel='rbf', alpha=0.1, gamma=gamma)
    krr.fit(X_train, y_train)
    y_pred = krr.predict(X_plot)

    ax.scatter(X_train, y_train, c='steelblue', s=20,
               label="Donnees")
    ax.plot(X_plot, np.sin(X_plot), 'g--', label="$\\sin(x)$")
    ax.plot(X_plot, y_pred, 'r-', lw=2,
            label=f"KRR ($\\gamma={gamma}$)")
    ax.set_title(f"$\\gamma = {gamma}$")
    ax.legend(fontsize=8)
    ax.set_ylim(-2, 2)

plt.tight_layout()
plt.savefig("kernel_ridge.pdf")
plt.show()
```

Sortie

```
[Figures sauvegardees: svm_kernels.pdf, kernel_pca.pdf,
 kernel_ridge.pdf]
```

12.10 Exercices

Exercice 12.1 (Feature maps explicites — niveau 1).

1. Pour le noyau polynomial $\kappa(\boldsymbol{x}, \boldsymbol{x}') = (1 + \boldsymbol{x}^\top \boldsymbol{x}')^2$ avec $\boldsymbol{x} \in \mathbb{R}^2$, déterminer explicitement la feature map $\phi : \mathbb{R}^2 \to \mathbb{R}^p$. Quelle est la valeur de p ?
2. Vérifier que $\kappa(\boldsymbol{x}, \boldsymbol{x}') = \langle \phi(\boldsymbol{x}), \phi(\boldsymbol{x}') \rangle$ sur l'exemple numérique $\boldsymbol{x} = (1, 2)^\top$, $\boldsymbol{x}' = (3, -1)^\top$.
3. La fonction $\kappa(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x}^\top \boldsymbol{x}')^2 - 1$ est-elle un noyau valide ? Justifier.

Exercice 12.2 (Théorème du représentant et Kernel Ridge — niveau 2).

1. Montrer que le problème de régression Ridge à noyau (12.8), par application du théorème du représentant, se réduit à :
$$\min_{\boldsymbol{\alpha} \in \mathbb{R}^n} \frac{1}{n} \|\boldsymbol{y} - \boldsymbol{K}\boldsymbol{\alpha}\|^2 + \lambda \, \boldsymbol{\alpha}^\top \boldsymbol{K} \boldsymbol{\alpha}$$
2. En dérivant par rapport à $\boldsymbol{\alpha}$ et en annulant le gradient, retrouver la solution (12.9).
3. Implémenter la Kernel Ridge Regression à partir de zéro (sans `KernelRidge`) avec un noyau RBF. Comparer vos prédictions avec celles de scikit-learn.

Exercice 12.3 (Kernel PCA et processus gaussiens — niveau 3).

1. Générer un jeu de données en forme de spirale ($n = 500$ points, 2 classes). Appliquer la Kernel PCA avec différents noyaux et valeurs de γ. Visualiser les résultats et identifier la meilleure combinaison.
2. Implémenter le k-means à noyau à partir de zéro en utilisant uniquement la matrice de Gram. Comparer avec `SpectralClustering` de scikit-learn.
3. Utiliser `GaussianProcessRegressor` de scikit-learn pour régresser $y = \sin(x) + \varepsilon$ avec un noyau RBF. Tracer la moyenne a posteriori et l'intervalle de confiance à $\pm 2\sigma$. Vérifier numériquement que la moyenne a posteriori coïncide avec la prédiction de `KernelRidge` (avec $\alpha = \sigma^2_{\text{bruit}}$).
4. (**Théorique**) Montrer que la somme de deux noyaux définis positifs est un noyau défini positif. Le produit de deux noyaux définis positifs est-il aussi un noyau défini positif ? Prouver ou donner un contre-exemple.

Bibliographie

[1] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[2] T. Hastie, R. Tibshirani, J. Friedman. *The Elements of Statistical Learning*. Springer, 2nd edition, 2009.

[3] K. P. Murphy. *Probabilistic Machine Learning : An Introduction*. MIT Press, 2022.

[4] S. Shalev-Shwartz, S. Ben-David. *Understanding Machine Learning : From Theory to Algorithms*. Cambridge University Press, 2014.

[5] M. Mohri, A. Rostamizadeh, A. Talwalkar. *Foundations of Machine Learning*. MIT Press, 2nd edition, 2018.

[6] G. James, D. Witten, T. Hastie, R. Tibshirani. *An Introduction to Statistical Learning*. Springer, 2nd edition, 2021.

[7] C. E. Rasmussen, C. K. I. Williams. *Gaussian Processes for Machine Learning*. MIT Press, 2006.

[8] B. Schölkopf, A. J. Smola. *Learning with Kernels*. MIT Press, 2002.

[9] V. N. Vapnik. *Statistical Learning Theory*. Wiley, 1998.

[10] F. Pedregosa et al. Scikit-learn : Machine Learning in Python. *JMLR*, 12 :2825–2830, 2011.

[11] D. Arthur, S. Vassilvitskii. k-means++ : The Advantages of Careful Seeding. *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1027–1035, 2007.