Intelligence Artificielle

Fabien Lauer

2025

Table des matières

Préface **3**
Option Apprentissage supervisé (L3) . . . 3
Cours Intelligence Artificielle & Machine Learnig (Polytech 5A FISA) . . . 3
Cours Intelligence Artificielle (M1) . . . 4
Cours Intelligence Artificielle & Machine Learnig (Polytech 5A FISE) . . . 4
Cours Apprentissage statistique et deep learning (M2 IA2VR) . . . 5

Qu'est-ce que l'IA ? **6**
Définitions de l'IA . . . 6
Première définition (inutile) . . . 6
Définition « historique » . . . 6
Définitions construites selon le but . . . 6
Une autre vision : Découpage « chronologique » des approches d'IA . . . 7
Apprentissage machine . . . 8

I Algorithmes d'IA sans apprentissage **9**

1 Résolution de problèmes **10**
1.1 Formalisation d'un problème . . . 10
1.2 Arbre de recherche . . . 11
1.3 Exemple : le taquin . . . 11
1.4 Algorithmes de recherche « aveugles » . . . 12
1.4.1 Recherche en *Largeur d'abord* . . . 12
1.4.2 Recherche en *Profondeur d'abord* . . . 13
1.4.3 Recherche *mixte/en profondeur d'abord itérée* . . . 13
1.4.4 Recherche à *Coût uniforme* . . . 13
1.5 Algorithmes de recherche « informés » . . . 14
1.5.1 Recherche gloutonne . . . 14
1.5.2 A* . . . 14
1.6 Algorithme générique . . . 14

2 Jeux **16**
2.1 Formalisation d'un jeu . . . 16
2.2 Arbre de jeu . . . 16
2.3 Minimax et ses variantes . . . 17
2.3.1 Algorithme minimax . . . 17
2.3.2 Algorithme alpha-beta (réduire b) . . . 18

2.3.3 Limitation de profondeur (réduire n) . . . 19
2.3.4 Expectimax (pour les jeux avec du hasard) . . . 20
2.4 Algorithme MCTS . . . 20

3 Bandits manchots **21**
3.1 Modélisation probabiliste . . . 21
3.2 Algorithme glouton (*exploiter au maximum*) . . . 23
3.3 Algorithme ϵ-glouton (*explorer à l'infini*) . . . 24
3.4 Algorithme UCB . . . 25
3.4.1 Interprétation du choix basé sur $B_t(i)$ . . . 26
3.4.2 Explication de la formule de $B_t(i)$ . . . 26
3.4.3 Regret de l'algorithme UCB . . . 27

4 Réseaux bayésiens **30**
4.1 Représetation directe d'une loi jointe . . . 30
4.2 Représentation graphique d'une loi jointe . . . 31
4.2.1 Factorisation de la loi jointe . . . 31
4.2.2 Tables de probabilités conditionnelles . . . 32
4.3 Construction d'un réseau bayésien . . . 34
4.4 Analyse des liens d'indépendances conditionnelles . . . 36
4.4.1 Triplet en forme de chaîne de cause à effet . . . 36
4.4.2 Triplet en forme de V inversé : cause commune . . . 37
4.4.3 Triplet en forme de V : effet commun . . . 38
4.4.4 Effect commun observé à distance . . . 39
4.4.5 Cas général . . . 41
4.5 Inférence . . . 41
4.6 Inférence approximative . . . 43
4.6.1 Estimation des probabilités conditionnelles . . . 44
4.6.2 Pondération par la vraisemblance . . . 45

II Apprentissage supervisé **46**

5 Bases de l'apprentissage supervisé **48**
5.1 Données disponibles . . . 48
5.2 Modèle de prédiction . . . 49
5.3 Modélisation probabiliste de l'apprentissage . . . 49
5.4 Fonction de perte et risque . . . 50
5.5 Objectif de l'apprentissage et risque empirique . . . 50
5.6 Paramètres et hyperparamètres . . . 51

6 Surapprentissage et Régularisation **52**
6.1 Surapprentissage . . . 52
6.2 Erreur vs complexité . . . 55
6.3 Décomposition de l'erreur . . . 55
6.4 Réglage des hyperparamètres et validation . . . 57

6.5 Régularisation 57

7 Estimation du risque **58**
7.1 Erreur de test 58
7.1.1 Quelles garanties sur les performances en généralisation ? 59
7.1.2 Combien d'exemples faut-il dans la base de test pour garantir une bonne estimation de $R(f)$? 60
7.1.3 Que gagne-t-on à avoir une plus grande base de test ? 60
7.1.4 Applicabilité de ces courbes et de la borne Équation 7.1 61
7.2 Validation croisée 62
7.2.1 K-fold 62
7.2.2 *Leave-one-out* (LOO) 62

8 Classification **64**
8.1 Classifieur optimal : le classifieur de Bayes 64
8.2 Classification linéaire 65
8.3 Classification multi-classe 69
8.3.1 Décomposition un-contre-tous 69
8.3.2 Décomposition un-contre-un 70

9 Arbres de décision, de régression et forêts aléatoires **71**
9.1 Arbres de décision (pour la classification) 71
9.2 Apprentissage 71
9.2.1 Recherche de la coupe optimale 72
9.2.2 Variables qualitatives 73
9.3 Partition de l'espace de représentation 73
9.4 Arbres de régression 73
9.5 Forêts aléatoires 74
9.5.1 Bootstrap 75
9.5.2 Bagging 75
9.5.3 Forêts aléatoires (*Random forests*) 75

10 Analyse discriminante Linéaire (ADL) **76**
10.1 Hypothèses sur le modèle génératif 76
10.2 ADL et classification linéaire dans le cas binaire 77
10.3 Apprentissage 78

11 Classifieur naïf de Bayes **81**
11.1 Hypothèses sur le modèle génératif 81
11.2 Application à des données binaires 82

12 Perceptron **84**
12.1 Apprentissage des poids (à seuil fixe) 84
12.2 Convergence de l'algorithme 85
12.3 Apprentissage des poids et du seuil 86

13 K-plus proches voisins **87**
13.1 K-plus proches voisins pour la classification . . . 87
13.2 K-plus proches voisins pour la régression . . . 88
13.3 Optimisation des calculs et KD-tree . . . 90

14 Machines à Vecteurs Supports (SVM) **92**
14.1 Hyperplan de marge maximale . . . 92
14.2 Algorithme d'apprentissage SVM linéaire . . . 95
14.3 SVM et régularisation . . . 98
14.4 Version duale de l'apprentissage SVM . . . 99
14.4.1 Vecteurs supports et forme parcimonieuse du modèle . . . 101
14.4.2 Calcul de b . . . 102
14.5 SVM non linéaires . . . 102

15 Régression **104**
15.1 Modèle optimal : la fonction de régression . . . 104
15.2 Régression linéaire . . . 105

16 Méthode des moindres carrés **106**
16.1 Démonstration de la formule des moindres carrés . . . 106

17 Moindres carrés régularisés (régression *ridge*) **108**
17.1 Lien entre la complexité du modèle et la norme des paramètres . . . 109

18 LASSO **110**
18.1 Modèles parcimonieux . . . 110

19 Méthodes à noyaux **114**
19.1 Projection non linéaire . . . 114
19.2 Le problème de la dimension . . . 117
19.3 Noyaux . . . 119
19.3.1 Noyaux définis positifs . . . 120
19.3.2 Construction de noyaux . . . 121
19.4 Astuce du noyau . . . 122
19.5 Cadre fonctionnel et espace de Hilbert à noyau reproduisant . . . 123
19.5.1 Espace de Hilbert . . . 123
19.5.2 Propriété de reproduction . . . 126
19.5.3 Apprentissage dans un RKHS . . . 126

20 Régression ridge à noyau (KRR) **128**
20.1 Formulation dans le RKHS . . . 128
20.2 Formulation duale . . . 128
20.3 Solution . . . 129

21 Réseaux de neurones **132**
21.1 Unité de calcul de base : le neurone (artificiel) . . . 132

21.2 Perceptron multicouches (MLP) . . . 133
21.2.1 Apprentissage des poids : minimisation de l'erreur sur les données . . 134
21.2.2 Descente de gradient stochastique . . . 134
21.2.3 Calcul des dérivées et rétropropagation du gradient . . . 135
21.2.4 Variantes de l'algorithme . . . 141
21.3 Réseaux de neurones pour la classification . . . 141

III Apprentissage non supervisé 143

22 Bases de l'apprentissage non supervisé 145
22.1 Données disponibles . . . 145
22.2 Clustering . . . 146
22.3 Estimation de densité . . . 149

23 Algorithme K-means 150
23.1 Détails de l'algorithme . . . 150
23.2 Critère optimisé par K-means . . . 155
23.3 Amélioration par multiples initialisations . . . 156
23.4 K-means++ : initialisation dispersée . . . 156

24 Clustering spectral 158
24.1 Construction du graphe . . . 158
24.2 Algorithme . . . 160
24.3 Justification théorique . . . 163

IV Théorie de l'apprentissage 168

25 Apprentissage PAC 170
25.1 Borne PAC . . . 171
25.2 Complexité en échantillon (*sample complexity*) . . . 171
25.3 Apprenabilité . . . 171
25.4 Apprenabilité efficace . . . 173

26 Apprentissage PAC agnostic 174
26.1 Borne PAC agnostique . . . 174
26.2 Complexité en échantillon agnostique (*sample complexity*) . . . 175
26.3 Apprenabilité . . . 175
26.4 Apprenabilité et bornes uniformes . . . 175

27 Bornes sur l'erreur de généralisation 177
27.1 Borne pour les classes de fonctions finies . . . 177
27.1.1 Interprétation de la borne . . . 178
27.1.2 Complexité en échantillon . . . 179

28 Bornes pour les classes de fonctions infinies **180**
28.1 Fonction de croissance . . . 180
28.1.1 Borne sur l'erreur de généralisation . . . 181
28.1.2 Symétrisation . . . 182
28.1.3 Reste de la preuve du Théorème 28.1 . . . 182
28.2 Dimension de Vapnik-Chervonenkis . . . 184
28.2.1 Borne à base de VC dimension . . . 184
28.2.2 VC dimension des classifieurs linéaires . . . 187
28.2.3 VC dimension infinie . . . 188

29 Complexité de Rademacher **199**
29.1 Borne à base de complexité de Rademacher . . . 200
29.2 Borne pour la classification (sans marge) . . . 202
29.3 Classifieurs à marge . . . 205
29.3.1 Fonction de perte et risque à marge . . . 206
29.3.2 Borne sur le risque à marge . . . 207
29.3.3 Version uniforme en γ . . . 208
29.4 Borne pour les classifieurs linéaires régularisés . . . 208

V Réduction de dimension **211**

30 Analyse en composantes principales (ACP) **212**
30.1 Analyse statistique . . . 212
30.1.1 Obtention de la première composante principale . . . 212
30.1.2 Obtention des composantes principales suivantes . . . 213
30.2 En pratique . . . 214

31 Analyse en composantes principales kernelisée (Kernel PCA) **217**
31.1 Version duale de la PCA . . . 218
31.1.1 Astuce du noyau . . . 219

Références **221**

Annexes **222**

A Python **222**
A.1 Bases de Python . . . 222
A.1.1 Conditionnelles . . . 223
A.1.2 Boucles . . . 223
A.1.3 Fonctions . . . 224
A.2 Graphiques . . . 224

B Probabilités **226**
B.1 Espace probabilisé . . . 226
B.2 Variables aléatoires (v.a.) . . . 227

B.3 Espérance . . . 228
B.3.1 Propriétés . . . 228
B.3.2 Inégalité de Jensen . . . 229
B.4 Variance et covariance . . . 229
B.4.1 Propriétés . . . 229
B.5 Indicatrice . . . 231
B.6 Couples de variables aléatoires . . . 231
B.7 Probabilités conditionnelles . . . 232
B.8 Factorisation d'une loi jointe . . . 232
B.9 Lois marginales . . . 232
B.10 Règle de Bayes . . . 232
B.11 Indépendance . . . 233
B.12 Espérance conditionnelle . . . 233

C Algèbre linéaire **234**
C.1 Vecteurs et matrices . . . 234
C.1.1 Transposée . . . 235
C.1.2 Addition . . . 235
C.1.3 Produit scalaire . . . 236
C.1.4 Multiplication matricielle . . . 236
C.1.5 Normes . . . 237
C.1.6 Vecteurs et matrices particuliers . . . 237
C.2 Inégalité de Cauchy-Schwarz . . . 238
C.3 Matrice inverse . . . 238
C.4 Rang d'une matrice . . . 239
C.5 Décomposition en vecteurs et valeurs propres . . . 239
C.6 Décomposition en valeurs singulières (SVD) . . . 240
C.6.1 Lien avec les vecteurs et valeurs propres . . . 242
C.6.2 SVD fine ou tronquée . . . 242

D Calcul différentiel **244**
D.1 Dérivée d'une fonction d'une variable . . . 244
D.2 Dérivées partielles et gradient d'une fonction de plusieurs variables . . . 244
D.3 Matrice jacobienne . . . 245
D.4 Quelques règles utiles . . . 245
D.4.1 Dérivation en chaîne . . . 245
D.4.2 Gradient d'un produit scalaire . . . 246
D.4.3 Gradient d'une fonction quadratique . . . 246
D.4.4 Matrice jacobienne d'un produit matrice-vecteur . . . 246

E Optimisation **247**
E.1 Optimisation sans contrainte . . . 247
E.1.1 Descente de gradient . . . 247
E.1.2 Optimisation locale vs globale . . . 252
E.1.3 Fonctions convexes et concaves . . . 253
E.2 Optimisation sous contraintes . . . 254

F Inégalités de concentration **255**
F.1 Inégalité de Markov 255
F.2 Inégalité de Bienaymé-Chebyshev 255
F.3 Inégalité de Chebyshev pour les moyennes 256
F.4 Borne de Chernoff 256
F.5 Inégalité de Hoeffding 257
F.6 Inégalité des différences bornées 260

Préface

Ce document présente des notes de cours sur l'Intelligence Artificielle.

Vous trouverez ci-dessous la liste des chapitres spécifiques à chaque cours.

Option Apprentissage supervisé (L3)

Les chapitres à étudier :

1. Qu'est-ce que l'IA
2. Bases de l'apprentissage supervisé
3. Classification
4. Arbres de décision
5. K-plus proches voisins
6. Perceptron
7. Méthode des moindres carrés

Les annexes suivants peuvent être utiles :

1. Probabilités
2. Algèbre linéaire

Cours Intelligence Artificielle & Machine Learnig (Polytech 5A FISA)

1. Qu'est-ce que l'IA
2. Résolution de problèmes
3. Jeux
4. Bandits manchots
5. Réseaux bayésiens
6. Apprentissage supervisé
7. Régression
8. Méthode des moindres carrés
9. Perceptron
10. K-plus proches voisins
11. SVM
12. Surapprentissage et régularisation
13. Régression ridge
14. LASSO
15. Méthodes à noyaux

16. Régression ridge à noyaux

Les annexes suivants peuvent être utiles :

1. Probabilités
2. Algèbre linéaire
3. Optimisation

Cours Intelligence Artificielle (M1)

1. Qu'est-ce que l'IA
2. Résolution de problèmes
3. Jeux
4. Bandits manchots
5. Réseaux bayésiens
6. Apprentissage non supervisé
7. K-means
8. Clustering spectral
9. Apprentissage supervisé
10. Régression
11. Méthode des moindres carrés
12. Analyse discriminante linéaire
13. Classifieur naïf de Bayes
14. K-plus proches voisins
15. Perceptron
16. Réseaux de neurones

Les annexes suivants peuvent être utiles :

1. Probabilités
2. Algèbre linéaire
3. Calcul différentiel
4. Optimisation

Cours Intelligence Artificielle & Machine Learnig (Polytech 5A FISE)

1. Apprentissage supervisé
2. Régression
3. Méthode des moindres carrés
4. Classification
5. Perceptron
6. K-plus proches voisins
7. SVM
8. Surapprentissage et régularisation
9. Régression ridge

10. LASSO
11. Méthodes à noyaux
12. Régression ridge à noyaux
13. Estimation du risque
14. Apprentissage non supervisé
15. K-means
16. Clustering spectral

Les annexes suivants peuvent être utiles :

1. Probabilités
2. Algèbre linéaire
3. Optimisation

Cours Apprentissage statistique et deep learning (M2 IA2VR)

1. Bases de l'apprentissage supervisé
2. Classification
3. Régression
4. Estimation du risque
5. Surapprentissage et régularisation
6. Régression ridge
7. LASSO
8. SVM
9. Méthodes à noyaux
10. Régression ridge à noyaux

ainsi que toute la partie Théorie de l'apprentissage.

Les annexes suivants peuvent être utiles :

1. Probabilités
2. Algèbre linéaire
3. Optimisation
4. Inégalités de concentration

> Mise en garde
>
> Pour plus d'informations, certaines démontrations détaillées sont données dans des cadres de ce type. La lecture de celles qui n'ont pas été vues en cours peut être considérée comme optionnelle.

Qu'est-ce que l'IA ?

L'intelligence artificielle n'est pas un domaine aisé à définir, et ses contours restent assez flous. Voici quelques définitions pour aider à y voir plus clair.

Définitions de l'IA

Première définition (inutile)

L'Intelligence Artificielle (IA) est l'étude, l'analyse et la conception de systèmes intelligents.

Cette définition est inutile car elle ne définit pas ce qu'est un système intelligent.

Définition « historique »

L'intelligence artificielle englobe tout ce qui se rapproche de l'étude d'une liste de problèmes qu'un « bête » ordinateur ne peut pas résoudre.

Par exemple : *résoudre un casse-têtes ; gagner contre le champion du monde aux dames, aux échecs, au go ; garer une voiture ; reconnaître l'écriture, la parole, un chien dans une photo ; comprendre une phrase ; attraper des objets ; écriture de la musique commme Bach...*

Le problème avec cette définition est qu'il suffise qu'un terme soit ajouté à la liste pour que les ordinateurs lui trouve une solution quelques années plus tard...

Définitions construites selon le but

On peut définir un système intelligent de différentes manières selon le but recherché. Voici quatre catégories d'approches divisées selon deux axes pour déterminer si un système intelligent doit être capable de *penser* ou *agir comme un humain* ou *non.*

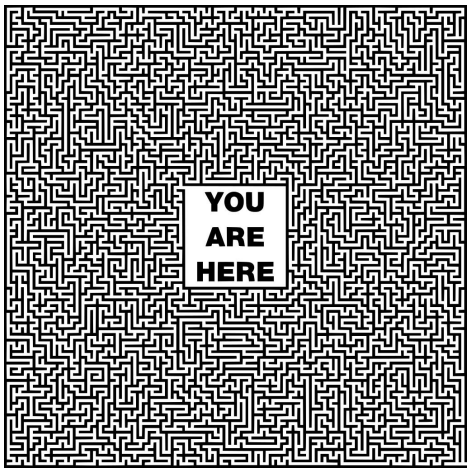
	comme un humain	**de manière rationnelle**
penser	Neurosciences / modélisation du cerveau	Logique, déductions automatiques
agir	Test(s) de Turing	Théorie de la décision, apprentissage...

Une autre vision : Découpage « chronologique » des approches d'IA

L'IA des années 50 à 80

Aux débuts de l'IA, celle-ci s'attachait surtout à *résoudre des problèmes complexes pour les humains avec des algorithmes simples.*

Exemples de problèmes complexes :



Pour résoudre ces types de problèmes, nous pouvons développer des « agents » qui chercheront à prendre les décisions permettant de maximiser un certain critère de performance étant donné leurs perceptions de l'environnement dans lequel ils évoluent. Nous verrons ici deux types d'agents et d'algorithmes classiques d'IA :

- des agents pour la résolution de problèmes ;
- des agents pour jouer à des jeux.

L'IA des années 80 à aujourd'hui

À partir des années 80, l'IA s'est tournée vers le *calcul de choses évidentes avec des algorithmes complexes.*

Par exemple, pour répondre à la question « Est-ce un chat ou un chien ? »

les méthodes mises en œuvre sont plus complexes, et demandent bien plus de manipulations mathématiques que celles utilisées pour vaincre le champion du monde aux échecs.

Ces problèmes sont généralement traités par des techniques d'**apprentissage machine**, qui, rassurons-nous, peuvent aussi servir à calculer des réponses à des questions bien moins évidentes pour un humain.

Apprentissage machine

De manière générale, l'apprentissage machine cherche à résoudre des problèmes qui n'ont pas de solution explicite (formule mathématique connue, algorithme...) ou dont la solution explicite est trop complexe pour être utilisée.

Pour résoudre ces problèmes, l'apprentissage machine repose sur des **données**.

Par exemple, si l'on cherche à construire un système de reconnaissance de caractères manuscrits, la fonction qui permet de passer de l'image au caractère est inconnue (car chaque image du même caractère peut être très différente au niveau des pixels), mais il est aisé de récolter des données sous la forme d'une collection d'images de chaque caractère considéré.

Le système sera ensuite capable « d'apprendre » à partir de ces données sa propre représentation des différents caractères et de leurs différences pour pouvoir ensuite les reconnaître dans de nouvelles images.

Il existe différentes formes d'apprentissage machine :

- l'apprentissage supervisé qui, comme dans l'exemple ci-dessus, travaille à partir de données étiquetées pour lesquelles la bonne réponse est connue ;
- l'apprentissage non supervisé qui, à l'inverse, extrait de l'information à partir de données non étiquetées (par exemple, un ensemble d'images sans information sur les objets qu'elles représentent) ;
- l'apprentissage par renforcement qui cherche à résoudre un problème de prise de décisions séquentielles.

partie I

Algorithmes d'IA sans apprentissage

1 Résolution de problèmes

Nous présentons ici les algorithmes d'intelligence artificielle dédiés à la résolution de problèmes. Il s'agit ici de développer des « agents intelligents », au sens où ceux-ci devraient être capables d'*agir de manière rationnelle*.

Un agent

- vie/évolue dans un **environnement**
- **perçoit** son environnement grâce à ses capteurs
- **agit** sur cet environnement grâce à des actionneurs/effecteurs.

Un agent intelligent est un agent qui exécute la meilleure action possible étant donné sa perception du monde.

Dans cette définition, il faut bien faire la différence entre l'agent intelligent et l'omniscient. On ne demande pas à l'agent de prendre la meilleure décision dans l'absolu, mais uniquement la meilleure à la vue de ce qu'il a pu observé du monde.

1.1 Formalisation d'un problème

Avant de pouvoir appliquer un algorithme de résolution de problème, il faut commencer par formaliser le problème. Cela se fait avec les composants suivants :

- un **objectif** que l'on cherche atteindre ;
- une liste d'**actions** possibles ;
- une définition de l'**état** du monde dans lequel l'agent évolue (la concaténation des variables qui représentent ce monde) ;
- l'**état initial** au début de la réflexion ;
- une fonction de transition qui définit l'état dans lequel le monde se retrouve après avoir pris telle action dans tel état ;
- (éventuellement) le **coût** de chaque action.

A partir de ces éléments, on définit également le **but** qui est l'ensemble des états dans lesquels l'objectif est atteint.

Les algorithmes de résolution de problèmes recherchent donc une **solution** qui est une suite d'actions garantissant d'atteindre le *but* à partir de l'*état initial*. Il peut exister plusieurs solutions pour le même problème, et dans ce cas on cherchera à obtenir si possible une solution de *coût* minimal.

1.2 Arbre de recherche

Les algorithmes de recherche pour la résolution de problèmes sont basés sur l'exploration d'un arbre de recherche dont les éléments ont la signification suivante :

- chaque nœud contient un *état* du problème ;
- chaque branche correspond à une *action* possible à partir de cet état (éventuellement pondérée par le coût de l'action) ;
- la racine contient l'*état initial*.

Une solution du problème est un chemin correspondant à une séquence d'actions qui mène de la racine jusqu'à un nœud dont l'état appartient au *but*. Le coût d'une solution est la somme des pondérations des branches traversées par le chemin.

1.3 Exemple : le taquin

Figure 1.1: taquin

- état : position des pièces et du trou ;
- actions possibles : déplacer le trou vers le haut, le bas, la gauche ou la droite *(en fonction de l'état, certaines actions sont impossibles)* ;
- coût d'une action = 1 pour toutes les actions car *on cherche à minimiser le nombre de coups pour arriver au but.*

Arbre de recherche :

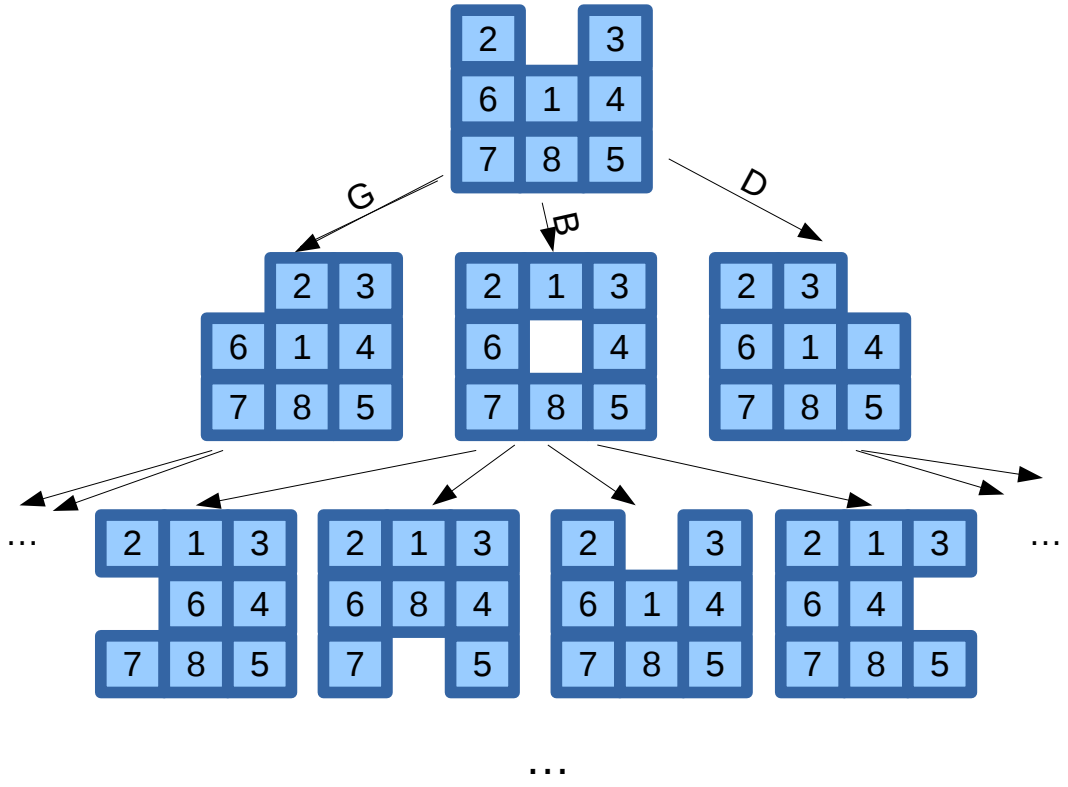


Figure 1.2: Arbre de recherche

1.4 Algorithmes de recherche « aveugles »

Tous les algorithmes de recherche explorent l'arbre de recherche dans des directions différentes et s'arrêtent dès qu'ils trouvent un nœud dont l'état appartient au *but* et retourne la solution correspondant au chemin de la racine à ce nœud.

1.4.1 Recherche en *Largeur d'abord*

Dans une recherche en largeur d'abord, l'arbre est exploré couche par couche en partant de la racine.

Cela garantie de trouver une solution avec le **moins d'actions possibles** mais demande beaucoup de mémoire. En effet, pour pouvoir explorer couche par couche, il faut au minimum retenir l'équivalent d'une couche en mémoire.

Si n est la longueur de la solution la plus courte (en nombre d'actions), alors l'arbre construit sera de profondeur n. Et si le facteur de branchement (le nombre de fils de chaque nœud) est b, alors le nombre de nœuds sur la dernière couche est b^n. Si $b \geq 10$ et que la solution est de longueur $n \geq 12$, alors la taille de cette couche dans la mémoire est de l'ordre du Teraoctet.

1.4.2 Recherche en *Profondeur d'abord*

Dans la recherche en profondeur d'abord, chaque branche est explorée jusqu'à la profondeur maximale avant de passer à la branche suivante, qui correspond à la plus profonde non encore explorée.

Cette recherche est plus efficace en termes de mémoire, car il suffit de retenir la branche courante en mémoire. Pour une profondeur n et un facteur de branchement b, la taille de cette branche est de l'ordre de bn, donc bien inférieur à b^n.

La recherche en profondeur d'abord peut trouver une solution rapidement si elle se trouve dans une des premières branches explorées, mais elle peut aussi perdre beaucoup de temps dans des branches inintéressantes.

En pratique, il est aussi utile d'altérer la liste des actions possibles pour éviter les cycles qui conduisent à creuser une branche de profondeur infinie en tournant en rond.

1.4.3 Recherche *mixte/en profondeur d'abord itérée*

La recherche en profondeur d'abord itérée permet de bénéficier des avantages des deux recherches précédentes. L'idée est d'appliquer la recherche en profondeur d'abord en incrémentant progressivement la profondeur maximale autorisée.

Ainsi, toute la couche de profondeur 1 est explorée avant celle à la profondeur 2 et ainsi de suite, selon le même ordre quand dans une recherche en largeur d'abord, ce qui conduit aux mêmes garanties de trouver la solution la plus courte.

L'avantage ici est que le cœur de l'algorithme fonctionne en profondeur d'abord et ne nécessite de retenir en mémoire que la branche courante. Le prix à payer pour bénéficier de tous ces avantages se compte en temps de calcul : pour explorer la couche à la profondeur n, l'algorithme repart de la racine et doit repasser par toutes les couches précédentes.

1.4.4 Recherche à *Coût uniforme*

La recherche en largeur d'abord donne des garanties uniquement en termes du nombre d'actions. Mais la solution la plus courte en actions n'est pas nécessairement la meilleure au sens du *coût*.

La recherche à coût uniforme explore à chaque itération le nœud qui minimise le coût *g(nœud)* entre la racine et le nœud.

Ainsi, lorsqu'elle trouve une solution, elle garantie qu'aucune autre solution ne possède un coût plus faible.

Si cette recherche est appliquée avec un coût constant pour chaque action (comme pour le taquin), alors le résusltat sera identique à la recherche en largeur d'abord.

1.5 Algorithmes de recherche « informés »

Les algorithmes de recherche informés incluent de la connaissance sur le problème au travers d'une **fonction heuristique** *h(noeud)* qui estime le coût restant pour le meilleur chemin entre le nœud et le *but*. Cette heuristique est utilisée afin d'explorer le nœud qui paraît être le « meilleur » d'abord.

1.5.1 Recherche gloutonne

La recherche gloutonne explore le nœud qui minimise *h(noeud)* pour se rapprocher le plus possible du but, mais sans garantie sur le coût de la solution car l'heuristique n'est qu'une estimation du coût réel.

1.5.2 A*

L'algorithme A* combine le coût uniforme et la recherche gloutonne pour explorer le nœud qui minimise *f(noeud) = g(noeud) + h(noeud)* = estimation du coût total d'une solution passant par *noeud*.

Si l'heuristique est *admissible* (elle ne surestime jamais le coût total d'une solution passant par *noeud*), alors l'algorithme A* est optimal : il retourne une solution qui minimise le coût.

1.6 Algorithme générique

Tous les algorithmes ci-dessus peuvent être implémentés à partir d'un squelette identique :

1. Initialiser la file d'attente avec la racine.
2. Prendre un nœud dans la file d'attente.
3. Vérifier si le but est atteint à ce nœud (si oui, retourner la solution).
4. Développer le nœud et ajouter les fils à la file d'attente.
5. Retourner à l'étape 2.

La seule différence entre tous les algorithmes ci-dessus réside dans la manière d'ordonner la file d'attente :

- Largeur d'abord : ajouter les nouveaux nœud à la fin de la file

- Coût uniforme : trier la file par ordre croissant de *g(noeud)*
- Profondeur d'abord : ajouter les nouveaux nœuds au début de la file
- Glouton : trier la file par ordre croissant de *h(noeud)*
- A* : trier la file par ordre croissant de *f(noeud) = g(noeud) + h(noeud)*

2 Jeux

Nous considérons ici les jeux

- déterministes : sans hasard (sauf ici) ;
- à information parfaite : tous les éléments sont connus de tous les joueurs ;
- à deux joueurs : un contre l'autre (le mode coopératif peut se formuler comme une résolution de problème).

Exemples de jeux qui entrent dans ce cadre :

- les dames
- Reversi/Othello
- les échecs
- le Go

2.1 Formalisation d'un jeu

Un jeu comprend les éléments suivants :

- une définition de l'**état** du jeu (la concaténation des variables susceptibles d'évoluer au cours d'une partie, par exemple : la position des pièces, tour du joueur...) ;
- un **état initial** dans lequel la partie commence ;
- une liste de **coups** ou **actions** possibles pour chaque état (ou une fonction de l'état qui retourne cette liste) ;
- un fonction de transition d'un état à l'autre en fonction du coup joué ;
- un **test de terminaison** pour détecter si l'*état* correspond à une fin de partie ;
- une **fonction de récompense** qui retourne le score en fonction de l'*état* atteint en fin de partie (par ex. $\{+1, 0, -1\}$ pour gagné, match nul ou perdu).

2.2 Arbre de jeu

Un arbre de jeu se définit comme un arbre de recherche où

- chaque nœud contient un *état* du problème ;
- chaque branche correspond à une *action* (un coup) possible à partir de cet état ;
- la racine contient l'*état initial de la réflexion* ;

mais avec un élément supplémentaire :

- chaque niveau/profondeur de l'arbre est associé à un joueur (celui qui doit jouer).

Cette différence est essentielle car l'adversaire agit sur le résultat : il ne suffit pas de prendre un chemin qui mène à la meilleure récompense, mais il s'agit de prendre un chemin qui mène à une bonne récompense quelles que soient les actions de l'adversaire.

> **Note**
>
> À chaque tour de jeu de l'ordinateur, celui-ci construit un arbre de jeu dans le but de trouver un tel chemin. Ainsi, l'*état initial* stocké dans la racine de l'arbre n'est pas l'état initial de la partie, mais l'état courant de la partie lorsque c'est à l'ordinateur de jouer.

2.3 Minimax et ses variantes

L'algorithme Minimax permet de calculer le coup à jouer pour l'ordinateur à partir d'un arbre de jeu.

Le problème ici est que l'algorithme ne contrôle pas les décisions de l'adversaire. En fait, l'algorithme ne peut même pas savoir comment va jouer l'adversaire, et il est donc difficile de prédire ce qu'il va se passer si l'on joue tel ou tel coup.

Pour pouvoir s'en sortir, il est donc nécessaire de faire des hypothèses sur la manière de jouer de l'adversaire.

2.3.1 Algorithme minimax

L'algorithme minimax adopte une **stratégie au pire cas**, c'est-à-dire qu'il travaille sous l'hypothèse que l'adversaire joue toujours le mieux possible, et donc de manière optimale.

Faire cette hypothèse permet deux choses :

- l'algorithme peut calculer les coups de l'adversaire à l'avance ;
- si l'adversaire ne joue pas comme prévu, alors il ne peut que jouer moins bien, et donc de manière plus avantageuse pour l'ordinateur que ce que l'algorithme avait prévu.

Cette stratégie permet donc de calculer le coup optimal avec des garanties sur le score obtenu en fin de partie.

Pour cela, l'algorithme calcule, récursivmenet à partir des feuilles, une valeur numérique (la valeur minimax) pour chaque nœud de l'arbre et détermine le coup à jouer comme celui qui maximise la valeur parmi l'ensemble des fils de la racine.

La valeur d'un nœud se calcule de trois manières différentes, en fonction du type de nœud:

- pour un **nœud Max** (*c'est à l'ordinateur de jouer*) : le maximum des valeurs des fils ;
- pour un **nœud Min** (*c'est à l'adversaire de jouer*) : le minimum des valeurs des fils ;

- pour une feuille (fin de partie) : la récompense (score de fin).

2.3.1.1 Complexité du minimax

Puisque l'algorithme effectue un calcul dans chaque nœud, les deux facteurs principaux qui influencent la complexité du minimax sont :

- le facteur de branchement moyen b
  environ 10 au reversi, 35 aux échecs, 360 au Go
- la profondeur de l'arbre n
  60 au reversi, environ 40 aux échecs et 400 au Go

La complexité est donc :

- le nombre de nœuds en $O(b^n)$ pour la complexité en temps ;
- la taille d'une branche en $O(bn)$ pour la complexité en espace (car l'algorithme travaille récursivement branche par branche).

La complexité en temps est ici bien trop grande pour être raisonnable pour la plupart des jeux connus et les variantes ci-dessous permettent de réduire les facteurs b et n.

2.3.2 Algorithme alpha-beta (réduire b)

L'idée de l'algorithme alpha-beta est d'élaguer l'arbre pour éviter d'explorer des branches inutiles. Ces branches inutiles sont celles qui ne seraient jamais jouées par des joueurs qui adoptent une stratégie optimale.

Il peut sembler difficile de les détecter *avant* d'avoir calculé la stratégie optimale, mais en réalité, on peut utiliser le fait que l'algorithme minimax parcourt l'arbre récursivmeent, et donc en profondeur d'abord. Ainsi, certaines informations sur une branche peuvent être obtenues *avant* de descendre dans une autre et utilisées pour couper des branches alternatives inutiles.

L'algorithme Alpha-beta peut s'écrire avec deux fonctions presque identiques à celles utilisées par minimax et simplement deux variables supplémentaires : α et β qui enregistrent les informations utiles des branches précédentes.

```
Initialiser alpha = -inf, beta = inf

Fonction valeurMax ( noeud, alpha, beta )
    v = -inf
    Pour chaque fils de noeud:
        calculer sa valeur et le max courant :
          v = max ( v, valeurMin(fils, alpha, beta) )
        Si v >= beta, retourner v
        alpha = max( alpha, v )
```

```
    Retourner v

Fonction valeurMin ( noeud, alpha, beta )
    v = inf
    Pour chaque fils de noeud
        calculer sa valeur et le min courant :
          v= min ( v, valeurMax(fils, alpha, beta))
        Si v <= alpha, retourner v
        beta = min( beta, v )
    Retourner v
```

> **Note**
>
> L'algorithme alpha-beta permet de gagner du temps pour un coût presque nul (deux variables et quelques tests) sans perdre aucune garantie sur la qualité de la solution. En effet, seules les branches dont les valeurs ne peuvent pas influencer la décision finale sont supprimées.

2.3.3 Limitation de profondeur (réduire n)

La limitation de profondeur est une technique très simple pour gagner énormément de temps : il suffit de ne pas explorer l'arbre jusqu'aux feuilles (fins de parties), mais uniquement jusqu'à une certaine profondeur prédéterminée (10 par exemple).

Le problème ici est que les valeurs des récompenses obtenues pour les feuilles ne sont plus accessibles et aucun calcul n'est possible. Pour contourner ce problème, on utilise une **fonction d'évaluation** qui se charge d'estimer la valeur des nœuds non terminaux au niveau de la profondeur maximale autorisée.

Pour être utile, cette fonction d'évaluation doit avoir les propriétés suivantes :

- être rapidement calculable ;
- être représentative des chances de gagner pour l'ordinateur à partir de l'état considéré.

Dans les jeux à somme nulle (où les joueurs ont des positions symétriques et jouent avec le même type de pièces et d'actions possibles), on découpe généralement cette fonction ainsi :

$$evaluation(etat) = evalJoueur(Max) - evalJoueur(Min)$$

à l'aide d'une fonction intermédiaire chargée d'évaluer la position d'un joueur en particulier. Celle-ci peut par exemple simplement compter le nombre de pièces d'un joueur dans les cas les plus simples.

> **Avertissement**
>
> Contrairement à alpha-beta, utiliser la limitation de profondeur ne permet plus de garantir le score de fin de partie et l'optimalité de la décision.
> En effet, tous les calculs sont basés sur la fonction d'évaluation qui n'est qu'une heuristique particulière.

2.3.4 Expectimax (pour les jeux avec du hasard)

Le hasard peut prendre différentes formes :

- un coup ne donne le résultat prévu qu'avec une certaine probabilité ;
- un tirage aléatoire pré-conditionne chaque coup ;

mais cela revient dans tous les cas à insérer un coup joué par le hasard entre les coups des adversaires.

Ainsi, on peut développer un **arbre de jeu probabiliste** contenant des *nœuds chance* sur des couches intermédiaires associées au hasard entre chaque couche d'un arbre de jeu classique. Dans cet arbre, les branches qui descendent d'un nœud chance sont étiquetées avec les probabilités d'être tirées au hasard.

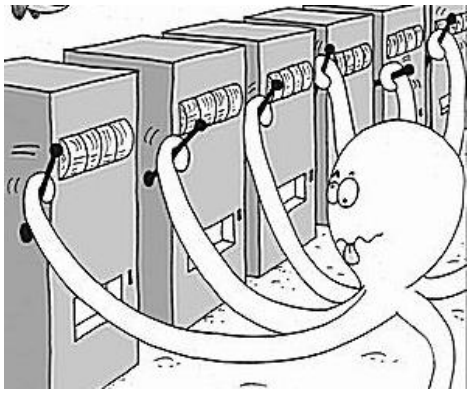
Par exemple, si chaque joueur doit lancer un dé avant de jouer, 6 branches sont crées avec les probabilités 1/6.

L'algorithme **Expectimax** fonctionne ensuite comme l'algorithme minimax avec une nouvelle règle pour ces nœuds chance : la valeur du nœud est la **moyenne pondérée** des valeurs sur l'ensemble des fils, où la pondération est donnée par les probabilités de chacune des actions tirées au sort.

2.4 Algorithme MCTS

À venir...

3 Bandits manchots



- Chaque machine/bras a une probabilité **inconnue** de gagner
- A chaque étape, l'algorithme choisit un bras à tirer
- But : maximiser les gains en essayant de tirer le plus souvent possible le meilleur bras

Ce problème met en évidence le **compromis Exploration / Exploitation** : il faut tirer le plus possible le bras que l'on pense être le meilleur (exploitation) tout en tirant suffisamment les autres pour améliorer notre connaissance des probabilités de gagner (exploration).

3.1 Modélisation probabiliste

Un problème de bandits manchots à K bras est modélisé par :

- les K variables aléatoires X_i représentant le gain du bras i, de **loi inconnue** P_i et de moyenne (espérance) $\mathbb{E}[X_i] = \mu(i)$;
- le meilleur bras i^* qui a la plus grande moyenne de gains :

$$i^* = \arg \max_{i=1,\dots,K} \mu(i)$$

- un algorithme/une politique $A : t \mapsto i$ qui choisit un bras i à chaque étape t.

Ainsi, si on tire T fois le bras i, on gagne

$$\sum_{t=1}^{T} X_{i,t}$$

où $X_{i,t}$ est une copie indépendante de X_i, c'est-à-dire une variable aléatoire avec exactement les mêmes propriétés.[1]

Donc on peut espérer gagner au mieux (en tirant sur le meilleur bras) :

$$Gain^*(T) = \mathbb{E}\left[\sum_{t=1}^{T} X_{i^*,t}\right] = \sum_{t=1}^{T} \mathbb{E}[X_{i^*,t}] = \sum_{t=1}^{T} \mu(i^*) = T\mu(i^*)$$

En pratique, si A est un algorithme déterministe et indépendant des $X_{i,t}$, on espère gagner

$$Gain(T) = \mathbb{E}\left[\sum_{t=1}^{T} X_{A(t),t}\right] = \sum_{t=1}^{T} \mu(A(t))$$

Ceci nous permet de définir le **regret** comme la somme de ce qu'on aurait pu espérer gagner en plus :

$$Regret(T) = \mathbb{E}\left[Gain^*(T) - Gain(T)\right] = \mathbb{E}\left[\sum_{t=1}^{T}(\mu(i^*) - \mu(A(t)))\right]$$

> **Note**
>
> Le regret est défini avec une espérance supplémentaire car en pratique tous les algorithmes intéressants, même s'ils sont déterministes, prennent des décisions aléatoires : la décision prise à l'instant t dépend des résultats aléatoires $X_{i,t-1}$, $X_{i,t-2}, \ldots$ précédents, et donc devient aléatoire.
> Concrètement, si j'applique le même algorithme chaque jour où je vais au casino, les bras qu'il me conseille de tirer ne sont pas les mêmes d'un jour à l'autre car mes gains sont différents.

La question est donc maintenant : comment choisir quel bras tirer pour **minimiser le regret** ?

Au début, on peut choisir n'importe lequel, car aucune information nous permet de choisir. Mais ensuite, différentes stratégies/algorithmes peuvent être mis en place.

[1] Le degré d'un sommet du graphe est la somme des pondérations des arêtes connectées à ce sommet. La matrice des degrés est la matrice diagonale où chaque terme sur la diagonale donne le degré d'un sommet.

Pour donner une indication, on peut reformuler le regret en fonction du nombre $N_T(i)$ d'essais de chaque bras parmi T tirages et des différences $\Delta_i = \mu(i^*) - \mu(i)$ entre les bras :

$$\begin{aligned} Regret(T) &= \mathbb{E}\left[\sum_{t=1}^{T}(\mu(i^*) - \mu(A(t)))\right] \\ &= \mathbb{E}\left[\sum_{i=1}^{K} N_T(i)(\mu(i^*) - \mu(i))\right] \\ &= \sum_{i=1}^{K} \mathbb{E}\left[N_T(i)\Delta_i\right] \end{aligned}$$

et, puisque les moyennes $\mu(i)$ et les différences Δ_i sont constantes (les réglages des machines à sous sont supposés ne pas changer),

$$Regret(T) = \sum_{i=1}^{K} \mathbb{E}\left[N_T(i)\right]\Delta_i \tag{3.1}$$

Ainsi, on voit que les choix de l'algorithme devrait conduire à des petits $N_T(i)$ pour les grands Δ_i, et donc tirer rarement sur les bras très différents du meilleur bras.

3.2 Algorithme glouton (*exploiter au maximum*)

L'algorithme glouton applique les étapes suivantes :

- Jouer au moins une fois chaque bras pour observer les gains $x_{i,j}$
- Estimer $\mu(i)$ par la moyenne des gains sur les tirages du bras i :

$$\hat{\mu}(i) = \frac{1}{N_t(i)} \sum_{j=1}^{t} x_{i,j}\mathbf{1}(A(j) = i)$$

- Choisir ensuite le bras qui maximise $\hat{\mu}(i)$:

$$A(t) = \arg\max_{i=1,\ldots,K} \hat{\mu}(i)$$

Cet algorithme semble pertinent et peut réussir à minimiser le regret si le bras optimal est rapidement trouvé.

Cependant, l'algorithme glouton risque de toujours choisir le même bras i sous-optimal, et donc d'accumuler un $\Delta_i = \mu(i^*) - \mu(i)$ dans le regret à chaque itération, ce qui conduit à un **regret linéaire en** T.

Exemple de mauvais tirage pour l'algorithme glouton avec K=3

Imaginons que les moyennes des bras soient $\mu(1) = 0.5$, $\mu(2) = 0.2$ et $\mu(3) = 0.6$. Le meilleur bras est donc $i^* = 3$.
Après 1 tirage sur chaque bras, on obtient $x_{1,1} = 0.67, x_{2,2} = 0.15, x_{3,3} = 0.4$, et donc les estimations des moyennes $\hat{\mu}(1) = 0.67, \hat{\mu}(2) = 0.15, \hat{\mu}(3) = 0.4$. L'algorithme glouton va donc choisir au prochain tirage pour $t = 4$, $A(t) = 1$ car le 1er bras paraît meilleur que les autres. Les tirages suivants pourront être plus faibles que le premier et la moyenne va tendre vers la vraie valeur : $\hat{\mu}(1) \to \mu(1) = 0.5$. Cependant, si la moyenne estimée converge vers $\mu(1)$ sans jamais repasser en-dessous de $\mu(3) = 0.4$, alors l'algorithme glouton se vérouillera sur le 1er bras et ne découvrira jamais le bras optimal.

3.3 Algorithme ϵ-glouton (*explorer à l'infini*)

À chaque étape,

- avec une probabilité $1 - \epsilon$, appliquer l'algorithme glouton ;
- avec une probabilité ϵ, tirer un bras aléatoirement.

Cela force l'algorithme à explorer suffisamment tous les bras pour finir par déterminer le bras optimal. En effet, si le nombre de tirages T est suffisamment grand, alors tous les bras auront été tirés suffisamment de fois pour que les moyennes estimées $\hat{\mu}(i)$ convergent vers les vraies moyennes $\mu(i)$, et donc pour que l'algorithme glouton choisisse le bras optimal.

Mais à chaque étape, le regret instantané est

$$\begin{aligned}\mathbb{E}[\mu(i^*) - \mu(A(t))] &= \sum_{i=1}^{K} \mathbb{E}\left[(\mu(i^*) - \mu(i))\mathbf{1}(A(t) = i)\right] = \sum_{i=1}^{K} \Delta_i P(A(t) = i) \\ &\geq \frac{\epsilon}{K} \sum_{i=1}^{K} \Delta_i = \textit{constante}\end{aligned}$$

et le **regret est donc linéaire en** T, comme pour l'algorithme glouton classique.

Cela est dû au fait que l'algorithme ϵ-glouton explore aléatoirement à l'infini, y compris lorsque le bras optimal a déjà été identifié, et donc que la probabilité $P(A(t) = i)$ pour les mauvais bras vaut toujours au moins ϵ/K.

Preuve

$$\begin{aligned}P(A(t) = i) &= P\Big(\textit{algo glouton},\ \textit{glouton choisit } i \bigcup \textit{algo aleatoire},\ \textit{aleatoire choisit } i\Big) \\ &= P(\textit{algo glouton},\ \textit{glouton choisit } i) + P(\textit{algo aleatoire},\ \textit{aleatoire choisit } i)\end{aligned}$$

avec $P(\textit{algo glouton},\ \textit{glouton choisit } i) \geq 0$. Donc

$$\begin{aligned} P(A(t) = i) &\geq P(\textit{algo aleatoire},\ \textit{aleatoire choisit } i) \\ &= P(\textit{algo aleatoire})P(\textit{aleatoire choisit } i) \\ &= \epsilon \cdot \frac{1}{K}. \end{aligned}$$

Pour éviter cela, il suffirait de toujours appliquer l'algorithme glouton à partir de l'instant où le bras optimal est identifié, mais il est impossible de prédire à quel instant cela correspond.

Remarque : en faisant décroître ϵ avec t, on peut obtenir un regret logarithmique, mais cela requiert la connaissance des Δ_i sensés être inconnus.

3.4 Algorithme UCB

Imaginons que l'on dispose non seulement d'une estimation $\hat{\mu}(i)$ de $\mu(i)$ mais aussi d'un semi-intervalle de confiance $C(i)$ tel que :

$$\mu(i) \leq \hat{\mu}(i) + C(i) = B(i)$$

Alors, on pourrait choisir le bras qui maximise $B(i)$ pour

- favoriser les bras que l'on croit bons (grand $\hat{\mu}(i)$) ;
- sans oublier les bras qui *pourraient* être meilleurs (grand $C(i)$).

Cette idée repose sur **l'optimisme face à l'incertitude** : on « fera confiance » aux bras en considérant qu'ils peuvent donner le meilleur d'eux-mêmes plutôt que de se concentrer uniquement sur leur performance moyenne observée.

L'algorithme UCB (*Upper Confidence Bound*) applique cette idée ainsi (pour des gains $X_i \in [0, 1]$) :

Tirer chaque bras une fois pour $t \leq K$, puis le bras i qui maximise

$$B_t(i) = \hat{\mu}_{N_t(i)}(i) + \sqrt{\frac{3 \ln t}{2 N_t(i)}}$$

où

- $\hat{\mu}_n(i) = \frac{1}{n} \sum_{j=1}^{n} x_{i,j}$ est l'estimation de $\mu(i)$ à partir de n observations $x_{i,j}$ des gains du bras i ;
- $N_t(i)$ est le nombre de tirages du bras i sur les t premiers coups.

3.4.1 Interprétation du choix basé sur $B_t(i)$

Les décisions de l'algorithme UCB visent à équilibrer le **compromis Exploitation/Exploration** :

- prendre $\hat{\mu}_{N_t(i)}(i)$ en compte signifie *exploiter* l'information gagnée sur les précédents tirages ;
- alors que le terme $\sqrt{\frac{3 \ln t}{2N_t(i)}}$ favorise l'**exploration** :
  - il augmente pour les bras qui ne sont pas assez souvent sélectionnés : $N_t(i)$ est constant et t augmente à chaque tirage dans lequel i n'a pas été choisi ;
  - il décroît pour le bras sélectionné car $N_t(i)$ augmente plus rapidement que $\ln t$

Donc si un bras paraît bon sur la base de $\hat{\mu}_{N_t(i)}(i)$, l'algorithme le sélectionnera mais il finira aussi par sélectionner un autre bras après de nombreuses sélections concentrées sur le même bras.

3.4.2 Explication de la formule de $B_t(i)$

La formule de $B_t(i)$ provient de la quantification de l'intervalle de confiance dans l'estimation de la moyenne. Celle-ci est obtenue par l'inégalité de Hoeffding qui garantie que pour une moyenne $\mu = \mathbb{E}X$ estimée à partir de n observations X_j par $\hat{\mu} = \frac{1}{n}\sum_{j=1}^{n} X_j$,

$$P\left\{\mu > \hat{\mu} + \epsilon\right\} \leq e^{-2n\epsilon^2}$$

dans le cas où $X \in [0, 1]$.

Pour l'algorithme UCB, on souhaite être de plus en plus sûr de l'intervalle de confiance ϵ utilisé au cours du temps, ce qui implique une probabilité de plus en plus faible que μ sorte de l'intervalle de confiance est soit supérieur à $\hat{\mu} + \epsilon$.

Soit δ_t une borne que l'on se fixe sur cette probabilité à l'instant t, on pose

$$e^{-2n\epsilon_t^2} = \delta_t$$

et on impose

$$\delta_t = t^{-3}$$

pour s'assurer que cette borne diminue au cours du temps.

On en déduit le (semi)-intervalle de confiance

$$\epsilon_t = \sqrt{\frac{3 \ln t}{2n}}$$

qui correspond au second terme de $B_t(i)$.

Avec ce choix, plus t est grand, plus on est sûr que $\mu \leq \hat{\mu} + \epsilon_t$. Cependant, pour pouvoir assurer cela, l'intervalle de confiance ϵ_t augmente, mais de façon raisonnable (logarithmique).

3.4.3 Regret de l'algorithme UCB

Le regret de la politique UCB sur T tirages est **logarithmique en** T :

$$Regret(T) \leq \left[\sum_{\Delta_i \neq 0} \frac{6}{\Delta_i}\right] \ln T + K\left(1 + \frac{\pi^2}{3}\right) = O(\ln T)$$

Preuve

Étant donné l'Équation 3.1, il suffit de montrer que

$$\mathbb{E}\left[N_T(i)\right] \leq \frac{6 \ln T}{\Delta_i^2} + 1 + \frac{\pi^2}{3}$$

pour tout $\Delta_i \neq 0$ pour obtenir le résultat souhaité.
Calculons le nombre de tirages du bras $i \neq i^*$ à l'aide de l'indicatrice: $N_T(i) = \sum_{t=1}^{T} \mathbf{1}(A(t) = i)$. Mais ajoutons une petite astuce : nous ne compterons que les tirages *après les n premiers tirages du bras*. C'est-à-dire que si l'on tire moins de n fois le bras i, nous nous contenterons de noter que $N_T(i) \leq n$. Cela donne :

$$\begin{aligned}
N_T(i) &\leq n + \sum_{t=n+1}^{T} \mathbf{1}(A(t) = i,\ N_{t-1}(i) \geq n) \\
&\leq n + \sum_{t=n+1}^{T} \mathbf{1}(\underbrace{\hat{\mu}_{t-1}(i^*) + C_{t-1,N_{t-1}(i^*)} \leq \hat{\mu}_{t-1}(i) + C_{t-1,N_{t-1}(i)}}_{\text{nécessaire pour avoir } A(t)=i},\ N_{t-1}(i) \geq n) \\
&\leq n + \sum_{t=n}^{T-1} \mathbf{1}(\hat{\mu}_t(i^*) + C_{t,N_t(i^*)} \leq \hat{\mu}_t(i) + C_{t,N_t(i)},\ N_t(i) \geq n)
\end{aligned}$$

Avec $C_{t,N} = \sqrt{\frac{3 \ln t}{2N}}$ l'intervalle de confiance utilisé par UCB.
La question est de savoir quand l'algorithme peut choisir le bras i au lieu du bras i^*. Cela arrive lorsque

$$B_t(i^*) = \hat{\mu}_{N_t(i^*)}(i^*) + C_{t,N_t(i^*)} \leq \hat{\mu}_{N_t(i)}(i) + C_{t,N_t(i)} = B_t(i)$$

qui nécessite soit $\mu(i^*)$ très sous-estimé, soit $\mu(i)$ très sur-estimé, soit une petite différence Δ_i :

$$\hat{\mu}_{N_t(i^*)}(i^*) \leq \mu(i^*) - C_{t,N_t(i^*)} \tag{3.2}$$

$$\hat{\mu}_{N_t(i)}(i) \geq \mu(i) + C_{t,N_t(i)} \tag{3.3}$$

$$\Delta_i = \mu(i^*) - \mu(i) < 2C_{t,N_t(i)} \tag{3.4}$$

On peut vérifier qu'au moins une de ces trois condition doit être remplie, par exemple, si on n'a ni Équation 3.2 ni Équation 3.4, $\hat{\mu}_{N_t(i^*)}(i^*) + C_{t,N_t(i^*)} > \mu(i^*) \geq \mu(i) + 2C_{t,N_t(i)}$ et il faut $\hat{\mu}_{N_t(i)}(i) + C_{t,N_t(i)} \geq \mu(i) + 2C_{t,N_t(i)} \Leftrightarrow$ Équation 3.3, et ainsi de suite...

Regardons l'Équation 3.4 : elle implique $N_t(i) < 6\ln t/\Delta_i^2 \leq 6\ln T/\Delta_i^2$, donc avec $N_t(i) \geq n = 6\ln T/\Delta_i^2 + 1$, il faut Équation 3.2 ou Équation 3.3 pour avoir $A(t+1) = i$ et tirer sur le bras i.
C'est pour cela que l'on ne compte pas les tirages avant n : pour simplifier les conditions et ne considérer les 2 premières.
Avec la linéarité de l'espérance et la propriété de l'indicatrice $\mathbb{E}[\mathbf{1}(E)] = P(E)$, on a

$$\begin{aligned}\mathbb{E}[N_T(i)] &\leq \mathbb{E}\left[n + \sum_{t=n}^{T-1} \mathbf{1}\left((3.2)\ ou\ (3.3)\right)\right] \\ &\leq n + \sum_{t=n}^{T-1} \mathbb{E}\left[\mathbf{1}\left((3.2)\ ou\ (3.3)\right)\right] \\ &\leq n + \sum_{t=n}^{T-1} P\left((3.2)\ ou\ (3.3)\right)\end{aligned}$$

Par la borne de l'union (Équation B.1) :

$$P\left((3.2)\ ou\ (3.3)\right) \leq P(3.2) + P(3.3)$$

La probabilité de Équation 3.2 est la probabilité de sortie d'un intervalle de confiance, qui peut être bornée par l'inégalité de Hoeffding. Cependant, celle-ci requiert que le nombre d'observations N soit fixé à l'avance et ne dépende pas des tirages aléatoires, comme c'est le cas avec $N_t(i^*)$. Pour contourner ce problème, on utilise le fait que l'Équation 3.2 n'intervient si elle est valide pour au moins une valeur de N entre 1 et t (l'instant courant) :

$$P(3.2) \leq P\left(\bigcup_{N=1}^{t} \{(3.2), N_t(i^*) = N\}\right) = \sum_{N=1}^{t} P\left((3.2), N_t(i^*) = N\right)$$

L'inégalité de Hoeffding donne alors

$$P\left((3.2), N_t(i^*) = N\right) \leq e^{-2NC_{t,N}^2}$$

et

$$\Rightarrow \quad P(3.2) \leq \sum_{N=1}^{t} e^{-2NC_{t,N}^2} = \sum_{N=1}^{t} e^{-3\ln t} = \sum_{N=1}^{t} t^{-3} = t^{-2}$$

De même, la probabilité de l'Équation 3.3 est $\leq t^{-2}$, donc

$$\mathbb{E}[N_T(i)] \leq n + \sum_{t=n}^{T-1} P((3.2)\ or\ (3.3)) \leq \frac{6\ln T}{\Delta_i^2} + 1 + \sum_{t=n}^{T-1} 2t^{-2}$$

La somme peut être bornée par une constante (grâce au problème de Bâle) :

$$\sum_{t=n}^{T-1} 2t^{-2} \leq 2\sum_{t=1}^{\infty} \frac{1}{t^2} = \frac{\pi^2}{3}$$

Donc

$$\mathbb{E}[N_T(i)] \leq \frac{6 \ln T}{\Delta_i^2} + 1 + \frac{\pi^2}{3} = O\left(\frac{6 \ln T}{\Delta_i^2}\right) = O\left(\ln T\right)$$

4 Réseaux bayésiens

Les réseaux bayésiens permettent de modéliser des phénomènes aléatoires complexes impliquant de nombreuses variables aléatoires par une approche graphique mettant en valeur les dépendances entre variables et limitant la taille de la représentation.

4.1 Représetation directe d'une loi jointe

Un **vecteur aléatoire** $\boldsymbol{X} \in \mathbb{R}^n$ est la concaténation de n variables aléatoires. Leur **loi jointe** se définie comme pour un couple de variables aléatoires. En particulier, pour un vecteur aléatoire *discret* :

$$P(\boldsymbol{X} = \boldsymbol{x}) = P(X_1 = x_1, X_2 = x_2, \ldots, X_n = x_n)$$

la loi jointe donne la probabilité d'observer les valeurs $x_1, ..., x_n$ pour les toutes les composantes $X_1, ..., X_n$ de $\boldsymbol{X}$.

Si l'on se limite à des vecteurs binaires $\boldsymbol{X} \in \{0, 1\}^n$, alors il est possible de représenter cette loi de probabilité par un tableau

Table 4.1: Représentation de la loi joint par un tableau de probabilités

x_1	x_2	$\ldots$	x_n	$P(\boldsymbol{X} = (x_1, x_2, \ldots, x_n))$
0	0		0	0.1
0	0		1	0.3
	$\vdots$			$\vdots$

Mais la taille de ce tableau peut vite rendre cette approche directe irréalisable. En considérant que les valeurs des x_i encodent l'indice de la ligne en binaire, il ne reste que la dernière colonne à stocker dans la mémoire, mais cela représente 2^n cases de nombres réels. Si l'on considère des `double` encodés sur 8 octets, cela donne les tailles de tableau suivantes :

n	taille mémoire du tableau
10	8 Ko
20	8 Mo
30	8 Go
40	8 To

Très vite, ce tableau ne tient plus en mémoire, ni même sur un disque dur.

En réalité, il suffit de ne stocker que $2^n - 1$ valeurs car la somme de toutes les probabilités doit être égale à 1 et il est donc possible de calculer la valeur non stockée comme 1 moins la somme des autres. Cependant, cela ne change pas vraiment le problème (et est probablement une mauvaise idée à implémenter en pratique).

> **!** Important
>
> La Table 4.1 a un autre gros défaut : la plupart des valeurs de la loi jointe sont inconnues ou difficiles à estimer car elles impliquent des cas de figures que l'expert du système modélisé n'a (presque) jamais rencontré. Et ces cas sont en si grand nombre qu'il est difficile de tous les considérer.

4.2 Représentation graphique d'une loi jointe

Un **réseau bayésien** représente une loi jointe de n variables avec un **graphe orienté sans cycles** tel que :

- chaque nœud correspond à une variable aléatoire (v.a.) ;
- les arcs orientés déterminent les (in)dépendances conditionnelles entre variables de sorte que *chaque v.a. est conditionnellement indépendante de ses prédécesseurs sachant ses parents :*

$$P(X_i = x_i | \boldsymbol{X}_{1:i-1} = \boldsymbol{x}_{1:i-1}) = P(X_i = x_i | \boldsymbol{X}_{parents(X_i)} = \boldsymbol{x}_{parents(X_i)})$$

  où $\boldsymbol{X}_{1:i} = (X_1, X_2, \ldots, X_i)$, $parents(X_i)$ est l'ensemble des parents de X_i et les v.a. sont ordonnées de telles que $j \in parents(X_i) \Rightarrow j < i$,

4.2.1 Factorisation de la loi jointe

Les règles de construction du graphe permettent d'obtenir la factorisation de la jointe

$$P(\boldsymbol{X} = \boldsymbol{x}) = \prod_{i=1}^{n} P(X_i = x_i \mid \boldsymbol{X}_{parents(X_i)} = \boldsymbol{x}_{parents(X_i)}) \tag{4.1}$$

qui constitue la définition de base d'un réseau bayésien.

Preuve

En utilisant la factorisation de la loi d'un couple récursivement, nous avons

$$\begin{aligned} P(\boldsymbol{X} = \boldsymbol{x}) &= P(X_n = x_n | \boldsymbol{X}_{1:n-1} = \boldsymbol{x}_{1:n-1}) P(\boldsymbol{X}_{1:n-1} = \boldsymbol{x}_{1:n-1}) \\ &= P(X_n = x_n | \boldsymbol{X}_{1:n-1} = \boldsymbol{x}_{1:n-1}) P(X_{n-1} = x_{n-1} | \boldsymbol{X}_{1:n-2} = \boldsymbol{x}_{1:n-2}) P(\boldsymbol{X}_{1:n-2} = \boldsymbol{x}_{1:n-2}) \\ &= \dots \\ &= \prod_{i=1}^{n} P(X_i = x_i | \boldsymbol{X}_{1:i-1} = \boldsymbol{x}_{1:i-1}) = \prod_{i=1}^{n} P(X_i = x_i | \boldsymbol{X}_{parents(X_i)} = \boldsymbol{x}_{parents(X_i)}) \end{aligned}$$

avec $1:0 = \emptyset$ et $parents(X_1) = \emptyset$.

4.2.2 Tables de probabilités conditionnelles

D'après la factorisation Équation 4.1, il suffit donc de connaître les valeurs des probabilités conditionnelles

$$P(X_i = x_i \mid \boldsymbol{X}_{parents(X_i)} = \boldsymbol{x}_{parents(X_i)})$$

pour pouvoir calculer n'importe quelle probabilité $P(\boldsymbol{X} = \boldsymbol{x})$ sans avoir besoin de stocker toute la Table 4.1.

Pour des variables discrètes, cela revient à stocker les tables de probabilités conditionnelles associées à chaque variable X_i.

En plus du gain mémoire, les valeurs de ces probabilités conditionnelles correspondent à des quantitées beaucoup plus simples à connaître ou estimer, ce qui permet en général de résoudre le problème identifié ici.

Exemple d'un réseau en forme de chaîne

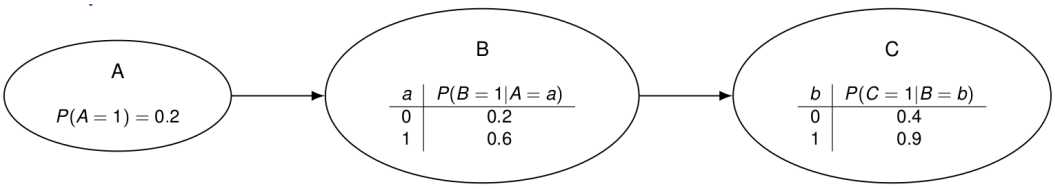
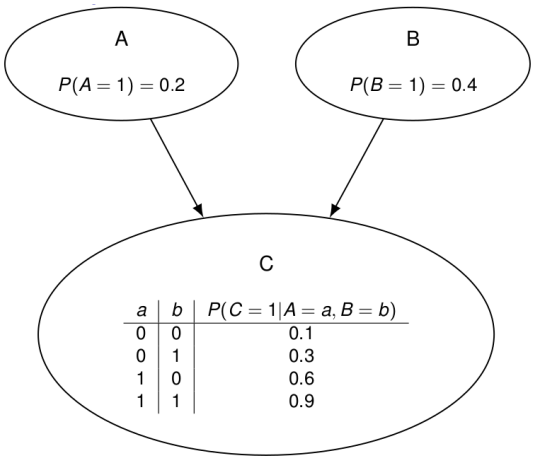


Figure 4.1: Réseau en forme de chaine

Dans ce réseau, 5 nombres suffisent pour encoder la loi jointe de 3 variables binaires au lieu de $2^3 - 1 = 7$. On peut calculer par exemple

$$\begin{aligned} P(A = 0, B = 1, C = 1) &= P(C = 1|B = 1) P(B = 1|A = 0) P(A = 0) \\ &= 0.9 \times 0.2 \times 0.8 = 0.144 \end{aligned}$$

Exemple d'un réseau en forme de V



a	b	$P(C = 1 \mid A = a, B = b)$
0	0	0.1
0	1	0.3
1	0	0.6
1	1	0.9

Ici, 6 nombres suffisent ici pour encoder la loi jointe de 3 variables binaires. Le calcul devient

$$\begin{aligned} P(A = 0, B = 1, C = 1) &= P(C = 1 \mid A = 0, B = 1) P(A = 0) P(B = 1) \\ &= 0.3 \times 0.8 \times 0.4 = 0.288 \end{aligned}$$

Exemple d'un réseau en étoile

On modélise un système d'alarme avec les variables suivantes :

- V : il y a un vol
- T : il y a un tremblement de terre
- A : l'alarme sonne
- J : mon voisin m'appelle
- M : une autre voisine m'appelle

Les voisins sont pris en compte car ils ont tendance à m'appeler plus souvent lorsque l'alarme sonne.

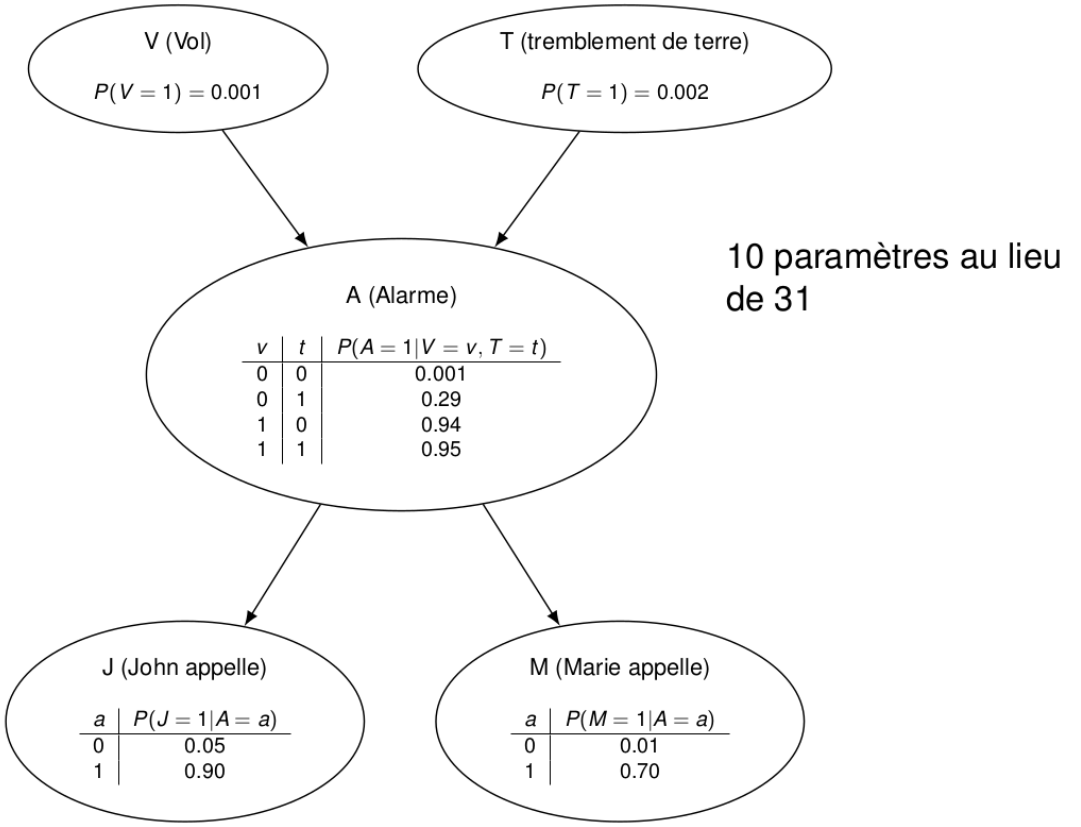


Figure 4.2: Réseau d'alarme

On peut aisément calculer la probabilité $P(V = 1, T = 0, A = 1, J = 1, M = 0)$ avec l'Équation 4.1 :

$$\begin{aligned}
&P(V = 1)P(T = 0)P(A = 1|V = 1, T = 0)P(J = 1|A = 1) \\
&\times P(M = 0|A = 1)P(A = 0, B = 1, C = 1) \\
&= P(C = 1|A = 0, B = 1)P(A = 0)P(B = 1) \\
&= 0.3 \times 0.8 \times 0.4 = 0.288
\end{aligned}$$

4.3 Construction d'un réseau bayésien

Pour une loi jointe donnée, il existe plusieurs topologies de réseau possibles, certaines étant plus complexes et moins intuitives que d'autres. Le but est donc en général de construire le réseau le plus simple possible pour modéliser un problème donné.

Dans un réseau simple, les **arcs modélisent les liens de cause à effet directs** et les « racines » du graphe (qui n'ont pas de parents) correspondent aux causes premières (les événements qui semblent indépendants du reste).

La procédure pour obtenir une telle représentation est la suivante.

- Ordonner les variables X_1, X_2, ..., X_n dans l'ordre des causes à effets.
- Pour $i = 1, \ldots, n$:
    1. ajouter un nœud pour la variable X_i ;
    2. ajouter des arcs en provenance des $parents(X_i)$ *déjà présents dans le réseau* en respectant les hypothèses d'indépendances conditionnelles : $P(X_i = x_i | \boldsymbol{X}_{parents(X_i)} = \boldsymbol{x}_{parents(x_i)}) = P(X_i = x_i | \boldsymbol{X}_{1:i-1} = \boldsymbol{x}_{1:i-1})$;
    3. remplir la table des probabilités conditionnelles de X_i (les valeurs de $P(X_i = x_i | \boldsymbol{X}_{parents(X_i)} = \boldsymbol{x}_{parents(x_i)})$ pour tout $\boldsymbol{x}_{parents(x_i)}$)

Reprenons l'exemple du réseau d'alarme, dans lequel la loi jointe sur 5 variables avait pu être encodée avec uniquement 10 paramètres. Ce réseau avait été obtenu par la procédure ci-dessus. Mais si nous inversons l'ordre des variables et l'appliquons, nous obtenons plus d'arcs et donc plus de paramètres.

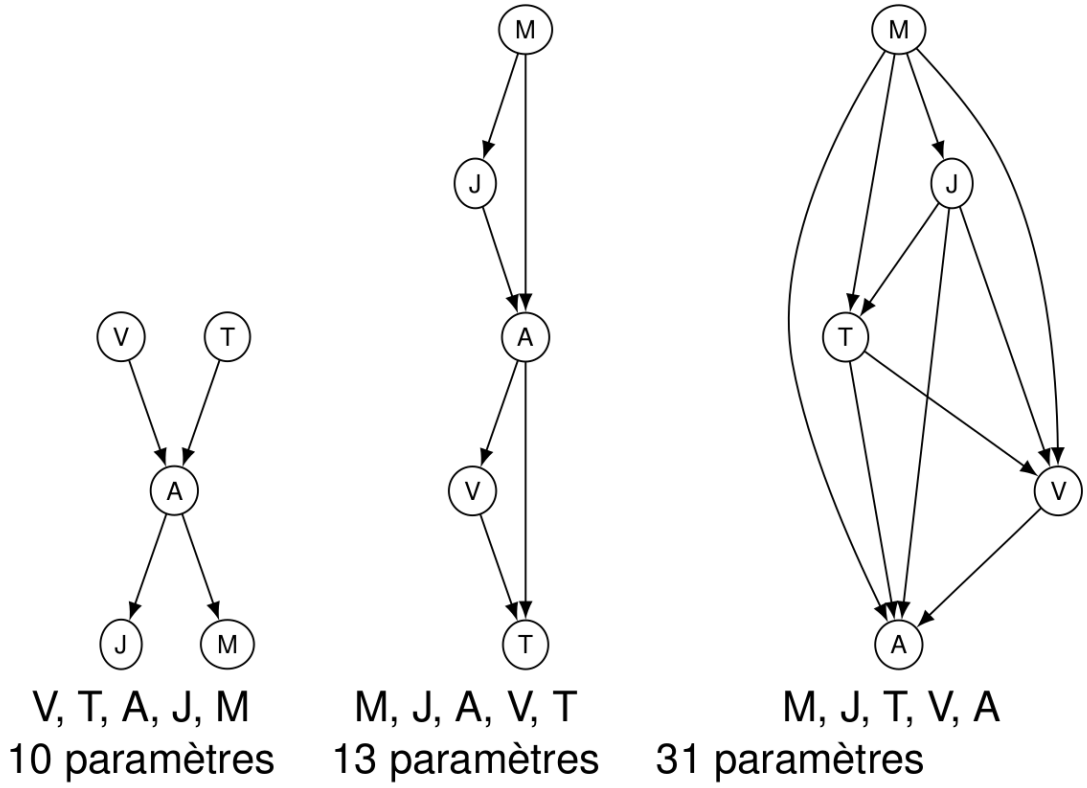


Figure 4.3: Influence de l'ordre des variables sur la taille du réseau

Cela illustre aussi le problème des valeurs à entrer dans les tables : alors que celles-ci peuvent facilement être déterminées dans le réseau de gauche, les autres correspondent à des questions du type *quelle est la probabilité que l'alarme sonne sachant que le voisin m'appelle et que la voisine ne m'appelle pas ?*, dont la réponse est loin d'être évidente...

4.4 Analyse des liens d'indépendances conditionnelles

Etant donné un réseau bayésien, nous pouvons retrouver les liens de dépendance et d'indépendance entre les variables, grâce à la notion de *d-séparation* (séparation dans un graphe orienté).

Ici nous introduisons les notations suivantes :

- $A \perp B$ pour « A et B sont indépendantes » ;
- $A \perp B \mid C$ pour « A et B sont conditionnellement indépendantes sachant C ».

Ici la variable C (la partie à droite du sachant que) est appelée « évidence » : c'est l'ensemble des variables dont la valeur est connue / observée.

Pour déterminer si $A \perp B \mid \boldsymbol{E}$, il faut savoir si A et B sont d-séparées par $\boldsymbol{E}$. Nous allons étudier cela sur quelques exemples de triplets élémentaires, avant de généraliser.

4.4.1 Triplet en forme de chaîne de cause à effet

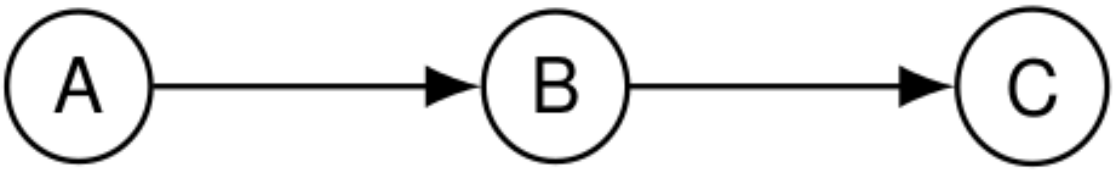


Figure 4.4: Chaine de cause à effet

Ici,

- $A \not\perp B$: A n'est *pas indépendant* de B car « A influence directement B » ;
- $A \not\perp C$: A n'est *pas indépendant* de B car « A influence directement B qui influence C » ;
- $A \perp C \mid B$: A est *indépendant* de C sachant B car toute l'influence de A sur C passe par B dont la valeur est fixée par l'observation.

Preuve d'indépendance

Pour prouver une indépendance, il suffit de montrer que la loi jointe est égale au produit des lois marginales :

$$\begin{aligned} P(A=a, C=c|B=b) &= \frac{P(A=a, B=b, C=c)}{P(B=b)} \\ &= \frac{P(A=a)P(B=b|A=a)P(C=c|B=b)}{P(B=b)} \\ &= P(A=a|B=b)P(C=c|B=b) \end{aligned}$$

4.4.2 Triplet en forme de V inversé : cause commune

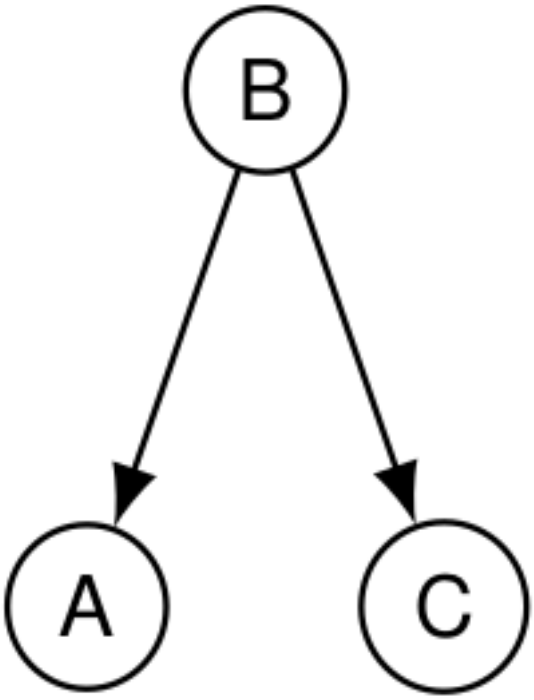


Figure 4.5: Cause commune

Ici,

- $A \not\perp C$: A n'est *pas indépendant* de C car « A et C sont tous deux directement influencés par B » ;
- $A \perp C \,|B$: A est *indépendant* de C sachant B car si la valeur de B est fixée, les probabilités de A sont entièrement déterminées et ne peuvent pas être modifiées par la valeur de C.

Preuve d'indépendance

Pour prouver une indépendance, il suffit de montrer que la loi jointe est égale au produit des lois marginales :

$$\begin{aligned} P(A=a, C=c|B=b) &= \frac{P(A=a, B=b, C=c)}{P(B=b)} \\ &= \frac{P(A=a|B=b)P(B=b)P(C=c|B=b)}{P(B=b)} \\ &= P(A=a|B=b)P(C=c|B=b) \end{aligned}$$

4.4.3 Triplet en forme de V : effet commun

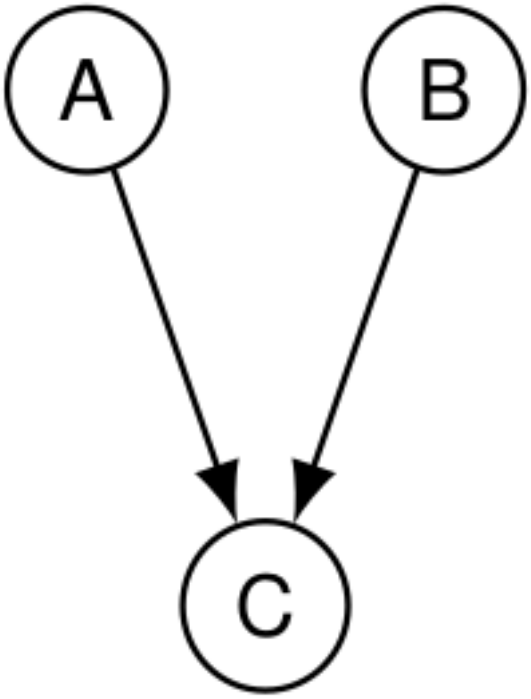


Figure 4.6: Effet commun

Ici,

- $A \perp B$: A est *indépendant* de B.

Preuve d'indépendance

Pour prouver une indépendance, il suffit de montrer que la loi jointe est égale au produit des lois marginales :

$$\begin{aligned} P(A=a, B=b) &= \sum_c P(A=a, B=b, C=c) \\ &= P(A=a)P(B=b)\sum_c P(C=c|A=a, B=b) \\ &= P(A=a)P(B=b) \end{aligned}$$

- $A \not\perp B \,|C$: A n'est *pas indépendant* de B sachant C car connaître la valeur de B modifie l'influence de A sur C, et inversement connaître la valeur de C crée un lien entre A et B.

Par exemple, si on constante une panne C qui peut avoir deux causes possibles A ou B, alors éliminer la cause B va augmenter la probabilité de la cause A.

Exemple illustratif avec A = j'ai gagné au loto, B = il fait beau, C = « je suis content » : si vous me voyez content, vous allez vous dire soit qu'il fait beau soit que j'ai gagné au loto. Mais si vous constatez ensuite qu'il pleut, cela va augmenter la probabilité avec laquelle j'ai gagné au loto : savoir que je suis content crée un lien de dépendance entre la météo et le résultat du loto.

4.4.4 Effect commun observé à distance

Reprenons le schéma précédent avec une variable supplémentaire :

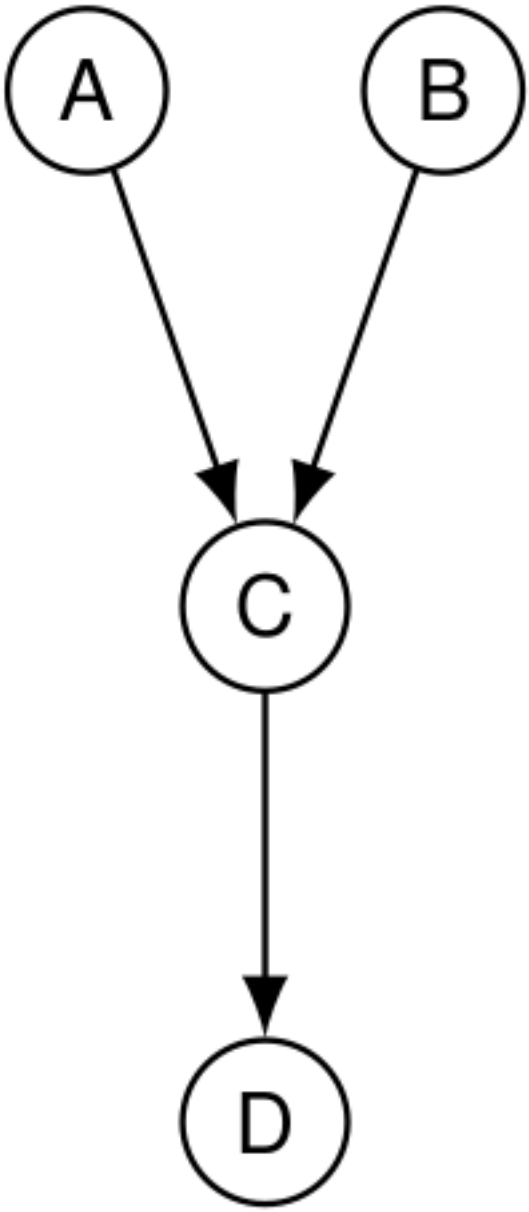


Figure 4.7: Effet commun observé à distance

Ici,

- $A \perp B$: A est *indépendant* de B ;
- $A \not\perp B \,|C$: A n'est *pas indépendant* de B sachant C ;
- em $A \not\perp B|D$: A n'est *pas indépendant* de B sachant D car connaître la valeur de D revient à observer C à distance et nous donne de l'information sur sa valeur, et donc crée un lien entre A et B.

Pour l'*exemple illustratif* ci-dessus, imaginez $D =$ « vous m'entendez chanter au loin ». Alors m'entendre chanter peut être suiffisant pour imaginer que je suis content et donc créer un lien statistique entre la météo et le loto dans votre tête.

4.4.5 Cas général

La méthode pour analyser les liens de dépendances dans le cas général repose sur l'étude des triplets élémentaires ci-dessus et résumée dans ce tableau :

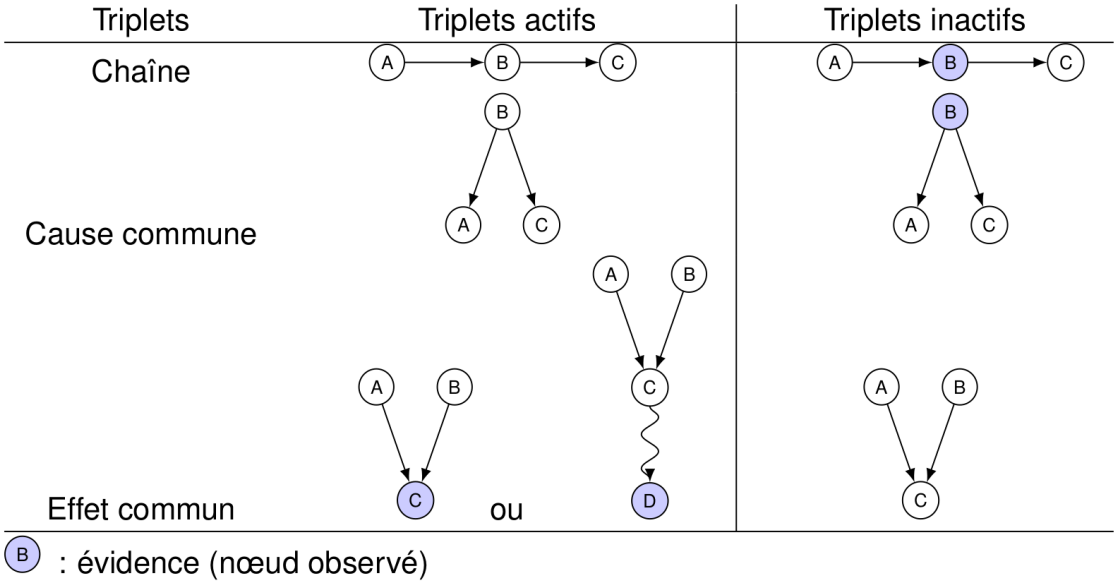


Figure 4.8: Triplets élémentaires

La règle est la suivante :

- $X \perp Y \,|Z$ *si et seulement si il n'existe pas de chemin (indépendamment de l'orientation des arcs) actif entre* X *et* Y *;*
- *un chemin est actif si tous les triplets qui le composent sont actifs.*

4.5 Inférence

Un réseau bayésien encode une loi jointe et permet donc de calculer n'importe quelle probabilité impliquant n'importe quel sous-ensemble des variables. Cela s'appelle l'inférence.

Exemple d'inférence dans un réseau en forme de chaîne

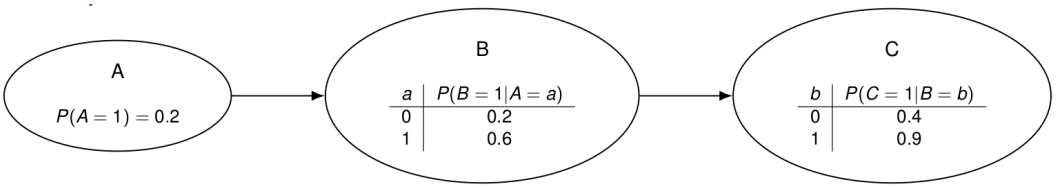


Figure 4.9: Réseau en forme de chaine

Différents cas de figure :

- $P(B = 1|A = 1) = 0.6 \rightarrow$ lecture directe depuis la table
- $P(C = 1|A = 1)$? Ici, il faut utiliser les règles de calcul sur les probabilités pour la reformuler en fonction des paramètres du réseau (c'est-à-dire les probabilités lisibles dans les tables) :

  $$\begin{aligned} P(C = 1|A = 1) &= P(C = 1, B = 0|A = 1) + P(C = 1, B = 1|A = 1) \\ &= P(C = 1|A = 1, B = 0)P(B = 0|A = 1) + P(C = 1|A = 1, B = 1)P(B = 1|A = 1) \\ &= P(C = 1|B = 0)P(B = 0|A = 1) + P(C = 1|B = 1)P(B = 1|A = 1) \end{aligned}$$

  où nous avons utilisé la loi des probabilités totales, puis la factorisation de la loi jointe pour un couple.
- $P(A = 1|B = 1)$? Ici, les variables sont inversées par rapport à l'ordre naturel du réseau. Pour inverser le conditionnement, nous utilisons la règle de Bayes, suivie de la loi des probabilités totales :

  $$\begin{aligned} P(A = 1|B = 1) &= \frac{P(B = 1|A = 1)P(A = 1)}{P(B = 1} \\ &= \frac{P(B = 1|A = 1)P(A = 1)}{P(B = 1|A = 1)P(A = 1) + P(B = 1|A = 0)P(A = 0)} \end{aligned}$$

Dans le cas général, nous pouvons utiliser les règles suivantes :

- calcul d'une probabilité marginale avec la loi des probabilités totales :

  $$P(X = 1) = \sum_{\boldsymbol{v}} P(X = 1, \boldsymbol{V} = \boldsymbol{v})$$

- calcul d'une probabilité conditionnelle $P(X = 1|\boldsymbol{E} = \boldsymbol{e})$:
  - si $parents(X) \subseteq \boldsymbol{E} \subseteq predecesseurs(X)$, alors lire la table des probabilités conditionnelles de X
  - sinon, il y a d'autres variables $\boldsymbol{V}$ et on passe par la loi jointe

$$
\begin{aligned}
P(X=1|\boldsymbol{E}=\boldsymbol{e}) = \frac{P(X=1,\boldsymbol{E}=\boldsymbol{e})}{P(\boldsymbol{E}=\boldsymbol{e})} &= \sum_{\boldsymbol{v}} \frac{P(X=1,\boldsymbol{V}=\boldsymbol{v},\boldsymbol{E}=\boldsymbol{e})}{P(\boldsymbol{E}=\boldsymbol{e})} \\
&= \sum_{\boldsymbol{v}} P(X=1,\boldsymbol{V}=\boldsymbol{v}|\boldsymbol{E}=\boldsymbol{e}) \\
&= \sum_{\boldsymbol{v}} P(X=1|\boldsymbol{V}=\boldsymbol{v},\boldsymbol{E}=\boldsymbol{e})P(\boldsymbol{V}=\boldsymbol{v}|\boldsymbol{E}=\boldsymbol{e})
\end{aligned}
$$

- calcul d'une probabilité conditionnelle en inversant la condition par la règle de Bayes :

$$
P(X=x|(\boldsymbol{E}_1,\boldsymbol{E}_2)=(\boldsymbol{e}_1,\boldsymbol{e}_2)) = \frac{P(\boldsymbol{E}_1=\boldsymbol{e}_1|X=x,\boldsymbol{E}_2=\boldsymbol{e}_2)P(X=x|\boldsymbol{E}_2=\boldsymbol{e}_2)}{P(\boldsymbol{E}_1=\boldsymbol{e}_1|\boldsymbol{E}_2=\boldsymbol{e}_2)}
$$

Exemple avec la réseau d'alarme

Reprenons l'exemple du réseau d'alarme pour calculer la probabilité que l'alarme sonne sachant que la voisine appelle et qu'il y a un vol :

$$
\begin{aligned}
P(A=1|M=1,V=1) &= \frac{P(M=1|A=1,V=1)P(A=1|V=1)}{P(M=1|V=1)} \\
&= \frac{P(M=1|A=1)P(A=1|V=1)}{P(M=1|V=1)}
\end{aligned}
$$

où nous avons appliqué la règle de Bayes et l'indépendance conditionnelle $M \perp V|A$ pour simplifier le numérateur.
Le dénominateur s'écrit

$$
\begin{aligned}
P(M=1|V=1) &= P(M=1|V=1,A=1)P(A=1|V=1) \\
&\quad + P(M=1|V=1,A=0)P(A=0|V=1) \\
&= P(M=1|A=1)P(A=1|V=1) + P(M=1|A=0)P(A=0|V=1)
\end{aligned}
$$

et

$$
\begin{aligned}
P(A=1|V=1) &= P(A=1|V=1,T=0)P(T=0) + P(A=1|V=1,T=1)P(T=1) \\
P(A=0|V=1) &= 1 - P(A=1|V=1)
\end{aligned}
$$

ce qui permet de calculer le résultat à partir des paramètres du réseau.

4.6 Inférence approximative

L'inférence exacte discutée ci-dessus est en fait bien souvent trop complexe, dans le sens où le nombre d'additions et de multiplications peut croître rapidement, en particulier lorsque plusieurs chemins existent entre les variables.

Cependant, la plupart du temps un résultat exact n'est pas nécessaire. En effet, estimer une probabilité valant 0.2098456 à 0.21 peut se révéler suffisant pour beaucoup d'applications.

L'inférence approximativement cherche donc à estimer les probabiltés par simulation, c'est-à-dire à partir de tirages aléatoires des variables selon la loi jointe encodée par le réseau. En effet, une probabilité peut toujours s'exprimer comme l'espérance d'une indicatrice, elle-même pouvant s'estimer par une moyenne sur un certain nombre de tirages :

$$P(\boldsymbol{X} = \boldsymbol{x}) = \mathbb{E}[\mathbf{1}(\boldsymbol{X} = \boldsymbol{x})] \approx \frac{\text{nb de tirages avec } \boldsymbol{X} = \boldsymbol{x}}{\text{nb total de tirages}}$$

Pour pouvoir simuler des tirages selon la bonne loi de probabilité, il faut

- commencer par tirer les variables dans les racines du graphe (les nœuds sans parents) dont les probabilités sont données indépendamment des autres variables ;
- suivre les arcs pour propager les valeurs tirées et connaître le conditionnement des variables suivantes.

Exemple de simulation avec un réseau bayésien

Considérons 4 variables modélisant la météo et l'humidité de l'herbe :

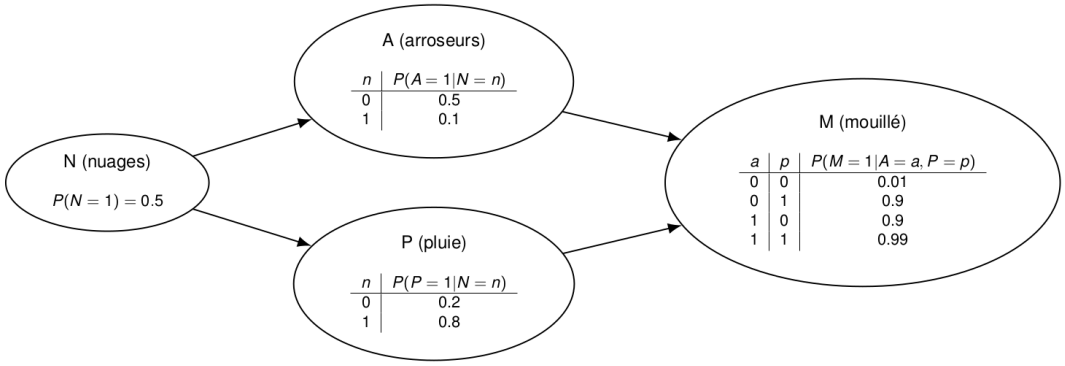


Figure 4.10: Exemple de réseau

Pour estimer $P(N = 1, A = 0, P = 1, M = 1)$, nous générons des tirages en commençant par la valeur binaire n de N avec $P(N = 1) = 0.5$. Cette valeur n est ensuite utilisée pour fixer les lois de probabilité de A et P à utiliser pour tirer ces variables : $P(A = 1|N = n)$ et $P(P = 1|N = n)$. Enfin, la valeur de M est tirée avec une probabilité d'avoir 1 fixée par les valeurs de a et p.

4.6.1 Estimation des probabilités conditionnelles

Pour une probabilité conditionnelle, nous utilisons la définition :

$$P(X = x|\boldsymbol{E} = \boldsymbol{e}) = \frac{P(X = x, \boldsymbol{E} = \boldsymbol{e})}{P(\boldsymbol{E} = \boldsymbol{e})} \approx \frac{\frac{\text{nb de tirages avec} X=x \text{ et } \boldsymbol{E}=\boldsymbol{e}}{\text{nb tirages total}}}{\frac{\text{nb de tirages avec } \boldsymbol{E}=\boldsymbol{e}}{\text{nb tirages total}}}$$
$$\approx \frac{\text{nb de tirages avec} X = x \text{ et } \boldsymbol{E} = \boldsymbol{e}}{\text{nb de tirages avec } \boldsymbol{E} = \boldsymbol{e}}$$

4.6.2 Pondération par la vraisemblance

L'estimation ci-dessus pose un souci lorsque la condition $\boldsymbol{E} = \boldsymbol{e}$ fait référence à un événement rare. Dans ce cas, la plupart des tirages seront jetés à la poubelle avant de calculer la probabilité qui sera au final estimée réellement qu'à partir de très peu de données.

Pour éviter ce souci, la pondération par la vraisemblance applique les modifications suivantes :

- à chaque tirage i, les valeurs $\boldsymbol{E} = \boldsymbol{e}$ sont forcées et non aléatoires (ce qui implique que tous les tirages sont retenus dans le calcul) ;
- la vraisemblance de chaque tirage v_i est mémorisée : elle est donnée par le produit des probabilités conditionnelles des valeurs forcées et correspond à la probabilité que ce tirage des variables forcées soit réellement issu de la loi jointe encodée par le réseau ;
- l'estimation est corrigée ainsi :

$$P(X = x|\boldsymbol{E} = \boldsymbol{e}) \approx \frac{\sum_i v_i \mathbf{1}(X = x)}{\sum_i v_i}$$

où chaque tirage est pondéré par sa vraisemblance.

partie II

Apprentissage supervisé

L'apprentissage machine cherche à résoudre (évenutellement de manière approximative) des problèmes complexes qui n'ont pas de solution explicite (formule mathématique connue, algorithme...) ou dont la solution explicite est trop coûteuse pour être utilisée.

L'apprentissage supervisé s'intéresse plus particulièrement aux problèmes pour lesquels il est possible de recueillir des exemples de comportement souhaité, c'est-à-dire, des questions associées à leur réponse attendue.

À partir de ces exemples, un algorithme d'apprentissage va construire un modèle capable de prédire la bonne réponse pour d'autres questions dont les réponses sont inconnues.

5 Bases de l'apprentissage supervisé

L'apprentissage supervisé se place dans le contexte où les données disponibles sont étiquetées, c'est-à-dire que pour chaque donnée d'entrée, la sortie attendue (la bonne réponse) est connue.

À partir de telles données, il s'agit d'apprendre un **modèle de prédiction** capable de **généraliser**, c'est-à-dire de répondre correctement à des questions qu'il n'a pas rencontrer pendant son apprentissage, et dont il n'a pas pu retenir la réponse par cœur.

5.1 Données disponibles

Les entrées d'un système d'apprentissage correspondent à une représentation numérique des objets sur lesquels on se pose des questions, que ce soit des images, des sons, des textes, ou plus simplement des tableaux de nombres. De manière générale, toutes ces données peuvent être représentées par des vecteurs d'une certaine dimension d :

$$\boldsymbol{x} \in \mathcal{X} \subset \mathbb{R}^d$$

Le choix des composantes de $\boldsymbol{x}$ (les descripteurs) est très lié à l'application visée. Il est aussi possible d'utilier des techniques de *sélection de variables* ou de *réduction de dimension* pour construire des représentations $\boldsymbol{x}$ plus efficaces.

Les étiquettes $y \in \mathcal{Y}$ correspondent aux réponses attendues pour chaque entrée $\boldsymbol{x}$. Ainsi un **exemple** pour l'apprentissage supervisé est un couple entrée-sortie $(\boldsymbol{x}, y)$. Les étiquettes peuvent prendre différentes formes qui définissent le type de problème considéré (et donc la famille d'algorithmes à utiliser).

- Si les étiquettes sont discrètes (par ex. $\mathcal{Y} = \{1, 2, 3\}$), il s'agit d'un problème de classification ;
- Si les étiquettes sont continues, $\mathcal{Y} \subset \mathbb{R}$, alors il s'agit d'un problème de régression.

En apprentissage supervisé, la **base d'apprentissage** contient des données étiquetées, c'est-à-dire des exemples d'entrées $\boldsymbol{x} \in \mathcal{X}$ et de sorties $y \in \mathcal{Y}$ associées :

$$S = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_m, y_m)\}$$

Ici, m représente la taille de la base d'apprentissage, ou encore, nombre d'exemples disponibles. En apprentissage, la base d'apprentissage S contient (souvent) les seules informations disponibles pour résoudre le problème.

5.2 Modèle de prédiction

À partir de ces données, un algorithme d'apprentissage construit un **modèle de prédiction**

$$f : \mathcal{X} \to \mathcal{Y}$$

qui est une fonction capable de prédire une valeur de y pour tout $x \in \mathcal{X}$. En réalité, l'algorithme se contente de choisir un modèle f parmi un ensemble de modèles possibles $\mathcal{F}$, avec comme objectif principal de choisir un modèle capable de généraliser, c'est-à-dire tel que

$$\forall\, \boldsymbol{x} \in \mathcal{X}, \quad f(\boldsymbol{x}) \stackrel{?}{=} y.$$

Pour vérifier cette capacité, il est nécessaire de tester le modèle sur des données indépendantes de celles utilisées pour l'apprentissage.

5.3 Modélisation probabiliste de l'apprentissage

Pour développer des algorithmes et comprendre les phénomènes en jeu dans l'apprentissage, nous avons besoin d'une modélisation. Celle-ci sera probabiliste, car un objet central de l'étude est la généralisation des modèles, c'est-à-dire leur capacité à prédire correctement les étiquettes inconnues et il nous est donc impossible d'affirmer quoi que ce soit avec certitude. Cependant, les probabilités nous permettrons d'être assez confiant dans les résultats.

Pour commencer, nous considérons que tous les exemples $(\boldsymbol{x}, y)$, aussi bien ceux de la base d'apprentissage que ceux que nous ne rencontrerons que dans le futur, sont issus de tirages indépendants d'un couple de variables aléatoires $(\boldsymbol{X}, Y) \in \mathcal{X} \times \mathcal{Y}$.

Cette hypothèse simple implique deux choses :

- les données sont **indépendantes**, et donc la probabilité de voir un nouvel exemple d'une certaine forme ne dépend pas des exemples déjà observés (cette hypothèse permet de garantir que chaque nouvel exemple apporte un maximum d'information) ;
- les données sont toutes issues de la même loi de probabilité, et donc ce qui définit intrinséquement le problème et les liens entre $\boldsymbol{x}$ et y ne change pas au cours du temps ou d'un exemple à l'autre.

La seconde hypothèse permet de garantir que les exemples de la base d'apprentissage contiennent de l'information utile pour fournir des prédictions correctes sur les exemples futurs. Cela signifie aussi que l'apprentissage ne permettra pas de prédire l'imprévisible, ce qui est plutôt rassurant.

Enfin, il faut noter qu'en apprentisage, la loi de probabilité du couple $(\boldsymbol{X}, Y)$ est supposée **inconnue**. En effet, la connaissance de cette loi implique la connaissance des liens qui existent entre $\boldsymbol{X}$ et Y il n'y aurait plus rien à apprendre.

5.4 Fonction de perte et risque

La capacité du modèle à généraliser est mesurée par son **risque**, aussi appelé **erreur de généralisation** et défini par

$$R(f) = \mathbb{E}\ell(f, \boldsymbol{X}, Y)$$

où ℓ est la **fonction de perte** qui mesure l'erreur de prédiction commise par f sur un exemple $(\boldsymbol{X}, Y)$.

Ce risque, défini par une espérance, représente l'erreur moyenne sur l'ensemble de toutes les données possibles, pondérées par leur probabilité d'occurrence. Ainsi, un modèle commettant beaucoup d'erreurs sur des données très rares n'aura pas nécessairement un risque important.

La fonction de perte choisie dépend du problème considéré mais doit respecter les propriétés suivantes :

- être positive : $\ell(f, \boldsymbol{x}, y) \geq 0$ pour tout $f, \boldsymbol{x}, y$;
- retourner une erreur nulle en cas de prédiction parfaite : $f(\boldsymbol{x}) = y \Rightarrow \ell(f, \boldsymbol{x}, y) = 0$.

5.5 Objectif de l'apprentissage et risque empirique

L'objectif principal de l'apprentissage est donc de trouver une fonction $\hat{f} \in \mathcal{F}$ qui minimise le risque $R(f)$. Cependant, le risque ne peut pas être calculé sans la connaissance de la loi de probabilité du couple $(\boldsymbol{X}, Y)$ supposée inconnue.

La plupart des algorithmes d'apprentissage se contentent donc de minimiser une estimation du risque au lieu de $R(f)$. À partir des données observées $S = ((\boldsymbol{x}_i, y_i))_{1 \leq i \leq m}$, nous pouvons définir le **risque empirique** (aussi appelé **erreur d'apprentissage**),

$$R_{emp}(f) = \frac{1}{m} \sum_{i=1}^{m} \ell(f, \boldsymbol{x}_i, y_i),$$

et l'algorithme ERM (pour Minimisation du Risque Empirique en anglais) :

$$\hat{f} = \arg\min_{f \in \mathcal{F}} R_{emp}(f)$$

qui retient le modèle $\hat{f}$ qui, parmi tous les modèles possibles de $\mathcal{F}$, donne la plus petite erreur moyenne sur la base d'apprentissage.

5.6 Paramètres et hyperparamètres

Les familles de modèles $\mathcal{F}$ sont souvent (mais pas toujours) paramétriques, c'est-à-dire que chaque modèle $f \in \mathcal{F}$ est défini par un nombre prédéfini de paramètres. Par exemple, l'ensemble des modèles linéaires ou affines de $x \in \mathbb{R}$ est défini avec 2 paramètres libres :

$$\mathcal{F} = \{f : \ f(x) = wx + b, \ w \in \mathbb{R}, \ b \in \mathbb{R}\}$$

Choisir $(w, b) \in \mathbb{R}^2$ revient à choisir une fonction $f \in \mathcal{F}$ et inversement. Les valeurs de ces paramètres sont donc déterminées par l'algorithme d'apprentissage.

À l'inverse, les **hyperparamètres** sont des paramètres d'un plus haut niveau qui influencent le modèle retenu au final mais qui ne peuvent pas être déterminés par l'algorithme lui-même, comme par exemple :

- les paramètres de fonctionnement interne de l'algorithme (comme le nombre maximum d'itérations autorisées) ;
- les paramètres de la classe de fonctions $\mathcal{F}$ elle-même, comme par exemple le degré D pour l'ensemble des polynômes

$$\mathcal{F}_D = \left\{ f : f(x) = \sum_{k=0}^{D} w_k x^k \right\}$$

6 Surapprentissage et Régularisation

Le surapprentissage est un phénomène courant en apprentissage supervisé auquel il faut faire particulièrement attention lorsque l'on apprend des modèles complexes avec insuffisamment de données.

La régularisation est une technique classique pour éviter le surapprentissage.

6.1 Surapprentissage

Le surapprentissage désigne l'apprentissage d'un modèle de prédiction qui « colle » trop aux données et qui se rapproche un peu trop d'un apprentissage par cœur ne permettant pas de généraliser.

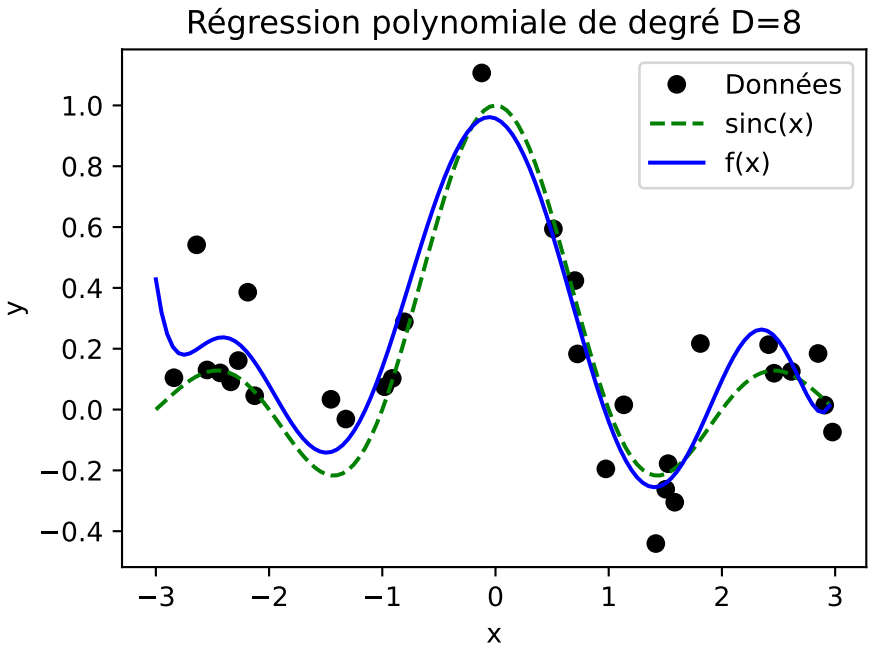
En effet, apprendre par cœur les données d'apprentissage est à la fois trop simple et pas efficace, car cela ne permet pas de prédire la bonne étiquette pour de nouvelles données inconnues.

Le fait d'apprendre un modèle trop proche des données peut être relié à la complexité d'un modèle : pour pouvoir coller aux données un modèle devra se montrer très flexible et donc capable d'implémenter beaucoup de fonctions différentes.

Illustration du surapprentissage en régression polynomiale

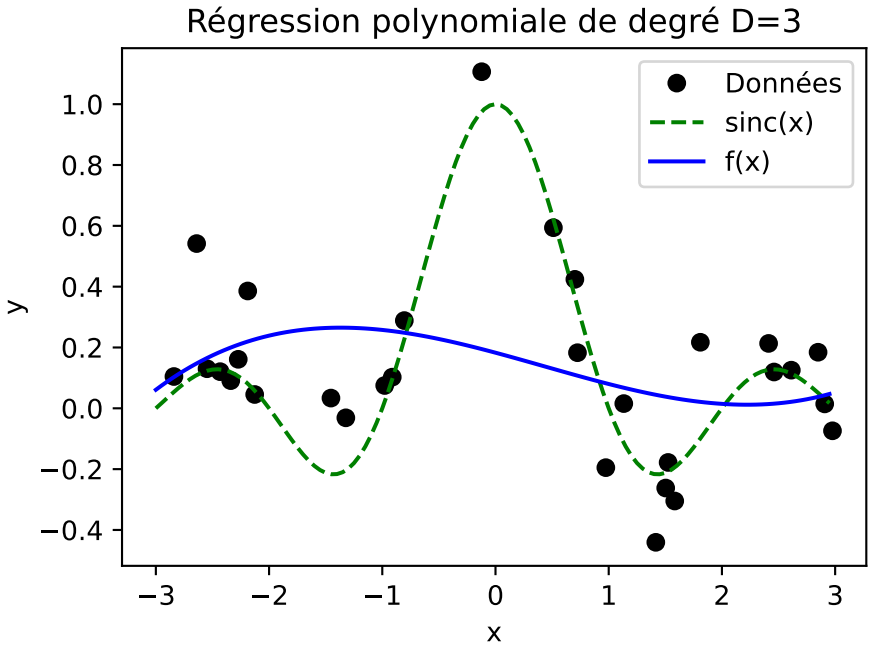
La régression polynomiale crée un modèle $f(x)$ polynomial pour prédire la valeur $y \in \mathbb{R}$. L'hyperparamètre majeur de cette méthode est le degré du polynôme qui doit être fixé à l'avance.
Si le degré D est correctement choisi, le modèle obtenu est à la fois satisfaisant sur les données d'apprentissage et en généralisation, comme ici pour $D = 8$ et des données correspondant à des mesures bruitées de la fonction $\text{sinc}(x) = \sin(\pi x)/(\pi x)$, si $x \neq 0$ et 1 si $x = 0$.

```python
def polyreg(X,y,D):
    PHI = np.ones((len(y),D+1))
    for k in range(1,D+1):
        PHI[:,k] = X * PHI[:,k-1]
    return np.linalg.lstsq(PHI,y)[0]

def polypred(X,w):
    PHI = np.ones((len(X),len(w)))
    for k in range(1,len(w)):
        PHI[:,k] = X * PHI[:,k-1]
    return PHI @ w

x = 6*np.random.rand(30)-3
y = np.sinc(x) + np.random.randn(30)*0.15
w = polyreg(x,y,8)
xt = np.arange(-3,3,0.05)
plt.plot(x,y, "ok")
plt.plot(xt,np.sinc(xt),"--g")
plt.plot(xt, polypred(xt,w), "-b")
plt.legend(["Données", "sinc(x)", "f(x)"])
plt.xlabel("x")
plt.ylabel("y")
t=plt.title("Régression polynomiale de degré D=8")
```



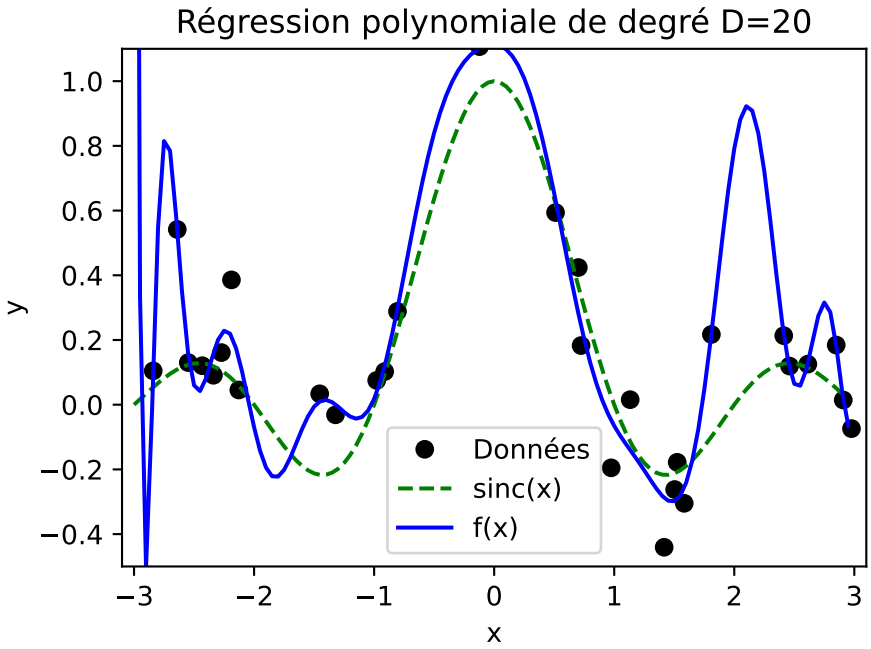
Par contre, si le degré est trop faible, par exemple $D = 3$, le modèle n'a pas la capacité suffisante pour approcher ni les données ni le modèle optimal.

```
w = polyreg(x,y,3)

plt.plot(x,y, "ok")
plt.plot(xt,np.sinc(xt),"--g")
plt.plot(xt, polypred(xt,w), "-b")
plt.legend(["Données", "sinc(x)", "f(x)"])
plt.xlabel("x")
plt.ylabel("y")
t=plt.title("Régression polynomiale de degré D=3")
```



À l'inverse, si le degré est trop élevé, par exemple avec $D = 20$, alors le modèle a la capacité de coller presque parfaitement aux données, mais conduit à des prédictions très éloignées de la réalité pour la plupart des x qui n'ont pas été vus pendant l'apprentissage.

```
w = polyreg(x,y,20)

plt.plot(x,y, "ok")
plt.plot(xt,np.sinc(xt),"--g")
plt.plot(xt, polypred(xt,w), "-b")
plt.legend(["Données", "sinc(x)", "f(x)"])
plt.xlabel("x")
plt.ylabel("y")
plt.axis([-3.1,3.1,-0.5,1.1])
t=plt.title("Régression polynomiale de degré D=20")
```



Cela est dû au surapprentissage : le modèle très flexible a fini par apprendre le bruit présent dans les données qui n'est pas généralisable pour d'autres points x.

Apprendre correctement pour pouvoir généraliser revient donc à trouver le bon compromis entre l'erreur d'apprentissage faible et la simplicité du modèle.

6.2 Erreur vs complexité

En règle général, les courbes du risque (en rouge) et du risque empirique (en bleu) en fonction de la complexité du modèle sont de cette forme :

- Lorsque la complexité est trop faible (vers la gauche), le modèle ne peut pas apprendre correctement les données et est aussi mauvais en généralisation.
- Plus la complexité du modèle augmente (vers la droite), plus sa flexibilité lui permet de « coller » aux données d'apprentissage et le risque empirique (en bleu) diminue.
- À partir d'une certaine complexité, le surapprentissage apparaît et l'erreur de généralisation augmente.

6.3 Décomposition de l'erreur

Notons le meilleur modèle disponible dans la classe de modèles prédéfinie comme $f^* \in \mathcal{F}$, c'est-à-dire celui qui minimise le risque en respectant les contraintes imposées par la méthode

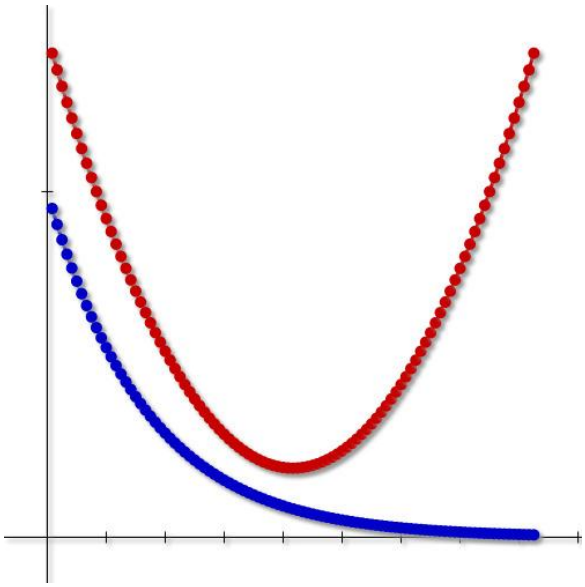


Figure 6.1: Erreur vs complexité

d'apprentissage choisie :

$$f^* = \arg\min_{f \in \mathcal{F}} R(f)$$

Le risque du meilleur modèle dans l'absolu (hors de toute contrainte) est

$$R(f_{Bayes/reg}) = \inf_f R(f)$$

où f_{Bayes} est le classifieur de Bayes pour la classification et f_{reg} est la fonction de régression, avec potentiellement $f_{Bayes/reg} \notin \mathcal{F}$.

Le risque d'un modèle $\hat{f}$ sélectionné par un algorithme se décompose en

$$R(\hat{f}) - R(f_{Bayes/reg}) = \underbrace{\left[R(\hat{f}) - R(f^*)\right]}_{\text{erreur d'estimation}} + \underbrace{\left[R(f^*) - R(f_{Bayes/reg})\right]}_{\text{erreur d'approximation}}$$

où

- l'**erreur d'approximation** est l'erreur due au choix de $\mathcal{F}$, elle vaut zéro si $f_{Bayes/reg} \in \mathcal{F}$;
- l'**erreur d'estimation** est l'erreur qui dépend des données et de la capacité de l'algorithme à choisir la bonne fonction dans $\mathcal{F}$.

Le dilemme de l'apprentissage est le suivant : plus la classe de fonctions $\mathcal{F}$ est grande, plus l'erreur d'approximation peut être faible, mais plus l'erreur d'estimation aura tendance à augmenter, car il est plus difficile pour un algorithme de trouver un modèle proche de f^* dans un ensemble plus vaste de modèles.

Le but de l'apprentissage est donc de trouver le bon compromis entre ces deux termes d'erreur. Cela se fait en pratique en règlant les hyperparamètres, comme le degré du polynôme dans l'exemple ci-dessus.

6.4 Réglage des hyperparamètres et validation

En pratique, le réglage des hyperparamètres ne peut se faire sans information supplémentaire, typiquement apportée par un jeu de données supplémentaires : la **base de validation**.

En effet, le risque n'étant pas accessible, seul le risque empirique est disponible, mais celui-ci ne permet pas de détecter le surapprentissage : la courbe bleue sur la Figure 6.1 ne permet pas de localiser le minimum de la courbe rouge.

La base de validation va nous permettre d'estimer le risque en testant le modèle sur des données *indépendantes* qu'il pas vues pendant l'apprentissage. Ainsi, nous pourrons sélection la valeur de l'hyperparamètre qui conduit à la plus petite erreur de validation sur cette base sans risquer le surapprentissage.

6.5 Régularisation

La régularisation vise à contrôler la complexité du modèle *pendant* l'apprentissage afin d'éviter le surapprentissage. Pour cela, l'apprentissage est formulé comme la minimisation d'un compromis entre

- un terme d'erreur (ou d'attache aux données) $\sum_{i=1}^{m} \ell(f, x_i, y_i)$ (ou une relaxation convexe de cette erreur)
- et un terme de régularisation, $\Omega(f)$, pénalisant les fonctions complexes :

$$\min_{f \in \mathcal{F}} \sum_{i=1}^{m} \ell(f, x_i, y_i) + \lambda \, \Omega(f)$$

Cette formulation introduit un hyperparamètre $\lambda > 0$ pondérant la régularisation et permettant de régler le compromis :

- pour λ grand, l'apprentissage se concentrera sur la minimisation de la complexité du modèle mesurée par $\Omega(f)$ et conduira à des modèles plus simples et plus éloignés des données ;
- pour λ petit, l'apprentissage se concentrera au contraire sur la minimisation des erreurs et conduira typiquement à des modèles plus complexes.

La régression ridge, le LASSO ou les SVM sont des exemples de méthodes d'apprentissage régularisé.

7 Estimation du risque

En apprentissage, le risque (ou erreur de généralisation) d'un modèle de prédiction est défini comme l'espérance de la fonction de perte :

$$R(f) = \mathbb{E}\ell(f, \boldsymbol{X}, Y)$$

et n'est pas calculable en pratique sans la connaissance de la loi de probabilité du couple $(\boldsymbol{X}, Y)$ ni l'accès à une infinité d'exemples (tirages de $(\boldsymbol{X}, Y)$).

À partir des données de la base d'apprentissage $S = ((\boldsymbol{x}_i, y_i))_{1 \leq i \leq m}$, nous pouvons calculer le risque empirique

$$R_{emp}(f) = \frac{1}{m} \sum_{i=1}^{m} \ell(f, \boldsymbol{x}_i, y_i)$$

mais celui-ci ne permet pas d'estimer $R(f)$ correctement lorsque le modèle f dépend des données (x_i, y_i) elles-mêmes (notamment à cause du surapprentissage). Il est donc nécessaire de disposer d'autres techniques pour estimer les performances des modèles de prédiction.

7.1 Erreur de test

Une approche assez simple consiste à conserver de côté une partie des données pour créer une **base de test** avec des données *indépendantes* de la base d'apprentissage :

$$S_{test} = ((\boldsymbol{x}'_i, y'_i))_{1 \leq i \leq m_{test}}$$

sur laquelle nous pouvons calculer l'**erreur de test**

$$R_{test}(f) = \frac{1}{m_{test}} \sum_{i=1}^{m_{test}} \ell(f, \boldsymbol{x}'_i, y'_i)$$

> **!** Important
>
> Pour rester valide, l'erreur de test doit bien être calculée sur des données **indépendantes** des données d'apprentissage, mais aussi du modèle f et donc de toute la procédure de sélection de ce modèle.
> La base de test ne peut donc pas être utilisée pour régler les hyperparamètres de l'algorithme.

7.1.1 Quelles garanties sur les performances en généralisation ?

Il est possible de garantir les performances en généralisation à partir de l'erreur de test.

Pour tout classifieur f et tout $\delta > 0$, avec une probabilité égale ou supérieure à $1 - \delta$ sur le tirage aléatoire de la base de test,

$$R(f) \leq R_{test}(f) + \sqrt{\frac{\ln \frac{1}{\delta}}{2m_{test}}} \tag{7.1}$$

Cette borne nous dit que le véritable risque $R(f)$ est estimé avec une précision de $\epsilon = \sqrt{\frac{\ln \frac{1}{\delta}}{2m_{test}}}$ par l'erreur de test $R_{test}(f)$.

Attention : cette borne n'est valide qu'avec une forte probabilité définie par l'indice de confiance δ. Le risque reste une quantité inaccessible avec une certitude de 100%. Cependant, il est tout à fait possible d'être assez sûr de ce que l'on avance à partir de l'erreur de test.

Pour obtenir cette garantie, nous utilisons l'inégalité de Hoeffding :

> Soit n variables aléatoires, $Z_1, \ldots, Z_n$, indépendantes et identiquement distribuées selon la loi de Z, et une fonction bornée h telle que $h(Z) \in [a, b]$. Alors, pour tout $\epsilon > 0$,
>
> $$P\left\{\mathbb{E}[h(Z)] - \frac{1}{n}\sum_{i=1}^{n} h(Z_i) \geq \epsilon\right\} \leq \exp\left(\frac{-2n\epsilon^2}{(b-a)^2}\right)$$

qui est instanciée avec

- un couple aléatoire exemple/étiquette $Z = (X, Y)$;
- un échantillon $(Z_i)_{1 \leq i \leq n} = (X'_i, Y'_i)_{1 \leq i \leq m_{test}}$ de $n = m_{test}$ copies indépendantes de Z
- une fonction de perte de classification $h(Z) = \mathbf{1}(f(X) \neq Y)$ bornée avec $a = 0$ et $b = 1$

Cela nous donne

$$P\left\{R(f) - R_{test}(f) \geq \epsilon\right\} \leq \exp\left(-2m_{test}\epsilon^2\right)$$

qui peut être utilisé ainsi :

$$\begin{aligned} P\left\{R(f) \leq R_{test}(f) + \epsilon\right\} &= 1 - P\left\{R(f) > R_{test}(f) + \epsilon\right\} \\ &\leq 1 - P\left\{R(f) \geq R_{test}(f) + \epsilon\right\} \\ &\leq 1 - \exp\left(-2m_{test}\epsilon^2\right) \end{aligned}$$

En posant

$$\delta = \exp\left(-2m_{test}\epsilon^2\right)$$

cela donne

$$\epsilon = \sqrt{\frac{\ln \frac{1}{\delta}}{2m_{test}}}$$

et la garantie d'avoir un écart pus faible que ce ϵ entre le risque et l'erreur de test avec une probabilité d'au moints $1 - \delta$.

7.1.2 Combien d'exemples faut-il dans la base de test pour garantir une bonne estimation de $R(f)$?

Pour une précision ϵ souhaitée et un indice de confiance δ fixé, la borne

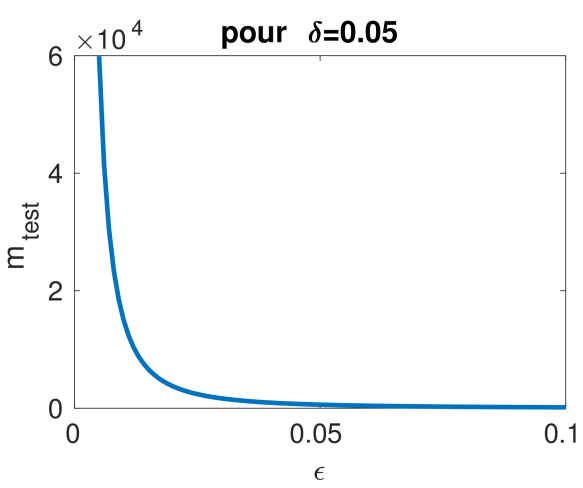
$$R(f) \leq R_{test}(f) + \epsilon$$

est satisfaite avec une probabilité d'au moins $1 - \delta$ si

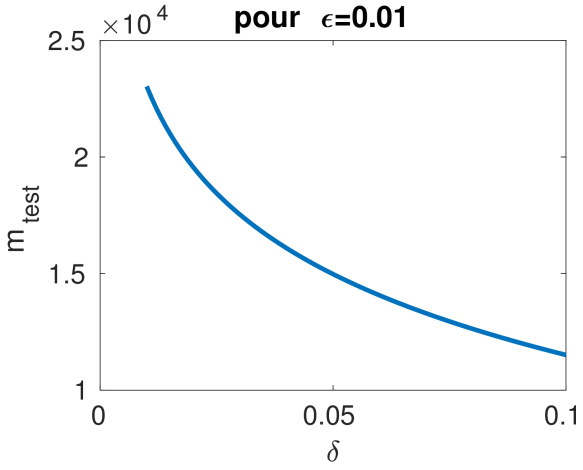
$$m_{test} \geq \frac{\ln \frac{1}{\delta}}{2\epsilon^2}$$

Pour obtenir ce résultat, il suffit de résoudre $\delta = \exp\left(-2m_{test}\epsilon^2\right)$ par rapport à m au lieu de ϵ.

Les courbes ci-dessous permettent de visualiser l'influence de ϵ et δ sur le nombre d'exemples de test nécessaire.



(a) Nombre d'exemples de test vs précision



(b) Nombre d'exemples de test vs indice de confiance

Par exemple, pour garantir une précision de $\epsilon = 1\%$ avec une confiance de 95% ($\delta = 0.05$), on doit avoir $m_{test} \geq 14980$.

7.1.3 Que gagne-t-on à avoir une plus grande base de test ?

La courbe ci-dessous montre la précision de l'estimation à partir de l'erreur de test en fonction de la taille de la base de test.

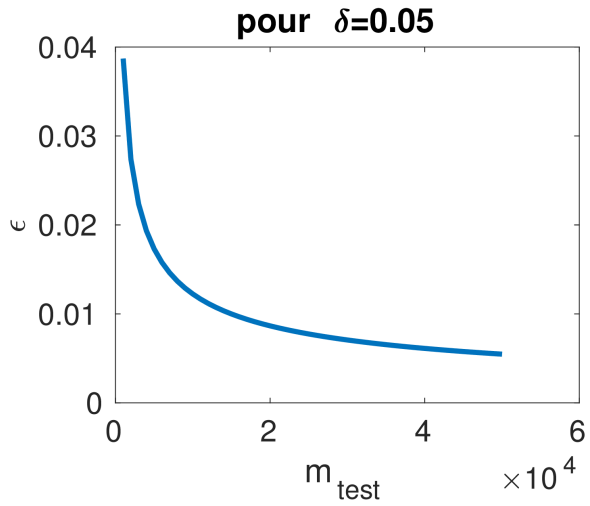


Figure 7.2: Précision vs le nombre d'exemples de test

On peut voir notamment qu'il n'est pas très intéressant d'avoir de très grandes bases de test car le gain en termes de précision n'est pas très important après $m > 50\ 000$.

7.1.4 Applicabilité de ces courbes et de la borne Équation 7.1

Les résultats ci-dessus sont valables

- quel que soit le problème / la distribution des données,
- quel que soit le classifieur / l'algorithme choisi,

sous l'hypothèse d'une base de test indépendante contenant des exemples indépendants et identiquement distribués.

Une autre limite concerne le choix de la fonction de perte utilisée pour calculer le risque. Les résultats sont limités à la classification avec l'erreur 0-1.

Cependant, en régression, il est possible d'obtenir le même type de résultats. La seule différence concerne l'amplitude de l'erreur donnée par a et b dans l'inégalité de Hoeffding. Pour

$$\ell(f, \boldsymbol{X}, Y) = (Y - f(\boldsymbol{X}))^2,$$

si les sorties sont bornées, $Y \in [-M, M]$, et que les modèles le sont aussi, $f(\boldsymbol{X}) \in [-M, M]$, alors $\ell(f, \boldsymbol{X}, Y) \in [0, 4M^2]$ et la garantie devient :

Pour tout $\delta > 0$, avec une probabilité égale ou supérieure à $1 - \delta$ sur le tirage aléatoire de la base de test,

$$R(f) \leq R_{test}(f) + 4M^2 \sqrt{\frac{\ln \frac{1}{\delta}}{2m_{test}}} \tag{7.2}$$

c'est-à-dire que ϵ est simplement multiplié par l'amplitude maximale de l'erreur (le ϵ précédent peut donc être vu comme un pourcentage de cette amplitude).

7.2 Validation croisée

L'erreur de test est un bon estimateur du risque, mais il a un défaut notoire : il nécessite de mettre des données de côté qui ne sont plus disponibles pour l'apprentissage. Lorsque que les données sont trop peu nombreuses, cela n'est pas toujours raisonable car l'algorithme n'a plus assez d'exemples pour apprendre correctement.

La validation croisée permet de palier à ce problème avec une procédure itérative qui considère un jeu de test différent à chaque itération.

7.2.1 K-fold

La procédure K-fold est basée sur le découpage de l'ensemble des données en K paquets. Puis, à chaque itération un paquet différent est sélectionné pour le test et les K-1 paquets restants servent à l'apprentissage.

Par exemple, pour $K = 5$, cela donne les 5 itérations suivantes :

Itération	Paquet 1	Paquet 2	Paquet 3	Paquet 4	Paquet 5	Erreur
1	**Test**	App	ren	tis	sage	R_1
2	App	**Test**	ren	tis	sage	R_2
3	App	ren	**Test**	tis	sage	R_3
4	App	ren	tis	**Test**	sage	R_4
5	App	ren	tis	sage	**Test**	R_5

À la fin de la procédure, le risque du modèle f **appris sur toutes les données** est estimé par

$$R_{Kfold}(f) = \frac{1}{K}\sum_{k=1}^{K} R_k$$

où R_k est l'erreur moyenne calculée sur le paquet de **Test** du modèle appris à la même itération sur les $K-1$ autres paquets.

7.2.2 *Leave-one-out* (LOO)

L'estimateur du risque LOO est donné par

$$R_{loo}(\hat{f}_S) = \frac{1}{m}\sum_{i=1}^{m} \ell(\hat{f}_{S\setminus(\boldsymbol{x}_i,y_i)}, \boldsymbol{x}_i, y_i)$$

où $\hat{f}_S$ est le modèle appris sur la base S.

Le LOO correspond simplement au cas limite du K-fold avec $K = m$: chaque exemple forme un « paquet » et il y autant d'itérations que de données disponibles.

Cette procédure est bien plus coûteuse en temps, car il faut réaliser m apprentissage, mais elle est parfois nécessaire lorsque m est trop faible.

Le **LOO est un estimateur « presque non biaisé » du risque** :

$$\mathbb{E}_S R_{loo}(\hat{f}_S) = \mathbb{E}_{S'} R(\hat{f}_{S'})$$

pour des jeux de données aléatoires S de m exemples et S' de $m-1$ exemples.

Preuve

$$\begin{aligned}
\mathbb{E}_S R_{loo}(\hat{f}_S) &= \frac{1}{m}\sum_{i=1}^{m} \mathbb{E}_S \ell(\hat{f}_{S\setminus(X_i,Y_i)}, X_i, Y_i) && (\mathbb{E} \text{ est linéaire}) \\
&= \frac{1}{m}\sum_{i=1}^{m} \mathbb{E}_S \ell(\hat{f}_{S\setminus(X_1,Y_1)}, X_1, Y_1) && (car\ S\ est\ iid) \\
&= \mathbb{E}_S \ell(\hat{f}_{S\setminus(X_1,Y_1)}, X_1, Y_1) \\
&= \mathbb{E}_{S',(X_1,Y_1)} \ell(\hat{f}_{S'}, X_1, Y_1) \\
&= \mathbb{E}_{S'} \mathbb{E}_{(X_1,Y_1)} \ell(\hat{f}_{S'}, X_1, Y_1) && (car\ (X_1,Y_1) \text{ est indép. de } S') \\
&= \mathbb{E}_{S'} R(\hat{f}_{S'})
\end{aligned}$$

8 Classification

En classification, l'étiquette $y \in \mathcal{Y}$ détermine la catégorie de $\boldsymbol{x} \in \mathcal{X}$ parmi un nombre de catégories C fini. Le modèle de prédiction est ainsi amené à classer les données $\boldsymbol{x}$ en différentes catégories et est couramment appelé un **classifieur**.

L'erreur d'un classifieur sur un exemple est mesurée par la fonction de perte 0-1 :

$$\ell(f, \boldsymbol{x}, y) = \mathbf{1}(f(\boldsymbol{x}) \neq y) = \begin{cases} 1, & \text{si } f(x) \neq y \\ 0, & \text{sinon} \end{cases}$$

D'après les propriétés de la fonction indicatrice, le risque de classification est donc la probabilité de mal classé un exemple :

$$R(f) = \mathbb{E}\mathbf{1}(f(\boldsymbol{X}) \neq Y) = P(f(\boldsymbol{X}) \neq Y)$$

On distingue souvent entre

- la **classification binaire** où seules deux catégories sont représentées, et généralement encodées par les étiqettes $\mathcal{Y} = \{-1, +1\}$;
- et la classification à catégories multiples ou multi-classe où $|\mathcal{Y}| > 2$.

8.1 Classifieur optimal : le classifieur de Bayes

Le classifieur de Bayes prédit la catégorie la plus probable pour tout $\boldsymbol{x}$:

$$f_{Bayes}(\boldsymbol{x}) = \arg\max_{y \in \mathcal{Y}} P(Y = y | \boldsymbol{X} = \boldsymbol{x})$$

Dans le cas binaire où $\mathcal{Y} = \{-1, +1\}$, il s'écrit aussi comme :

$$f_{Bayes}(\boldsymbol{x}) = \begin{cases} +1, & \text{si } P(Y = 1 | \boldsymbol{X} = \boldsymbol{x}) \geq P(Y = -1 | \boldsymbol{X} = \boldsymbol{x}) \\ -1, & \text{si } P(Y = 1 | \boldsymbol{X} = \boldsymbol{x}) < P(Y = -1 | \boldsymbol{X} = \boldsymbol{x}) \end{cases}$$

Ce classifieur est le classifieur optimal, c'est-à-dire le meilleur possible parmi toutes les fonctions de $\mathcal{X}$ dans $\mathcal{Y}$:

$$R(f_{Bayes}) = \min_{f : \mathcal{X} \to \mathcal{Y}} R(f)$$

et donc celui qui possède la plus petite probabilité de se tromper.

Preuve

Pour tout $f : \mathcal{X} \to \mathcal{Y}$, nous montrons les étapes suivantes :

1. $P(f(\boldsymbol{X}) \neq Y | \boldsymbol{X} = \boldsymbol{x}) = [1 - 2P(Y = 1 | \boldsymbol{X} = \boldsymbol{x})]\mathbf{1}[f(\boldsymbol{x}) = 1] + P(Y = 1 | \boldsymbol{X} = \boldsymbol{x})$.
2. Pour tout $\boldsymbol{x} \in \mathcal{X}$,

$$\Delta = P(f(\boldsymbol{X}) \neq Y | \boldsymbol{X} = \boldsymbol{x}) - P(f_{Bayes}(\boldsymbol{X}) \neq Y | \boldsymbol{X} = \boldsymbol{x}) \geq 0$$

3. $R(f) \geq R(f_{Bayes})$.

8.2 Classification linéaire

En classification binaire, avec $\mathcal{Y} = \{-1, +1\}$, un classifieur linéaire de $\mathcal{X} \subset \mathbb{R}^d$ est un classifieur de la forme

$$f(\boldsymbol{x}) = \text{signe}\left(\boldsymbol{w}^T \boldsymbol{x} + b\right) \tag{8.1}$$

avec des paramètres $\boldsymbol{w} \in \mathbb{R}^d$ et $b \in \mathbb{R}$.

La **frontière de décision** d'un classifieur est la frontière entre les régions de l'espace $\mathcal{X}$ affectées à différentes catégories. Pour un classifieur linéaire, cette frontière est un **hyperplan** :

$$H = \{\boldsymbol{x} : \boldsymbol{w}^T \boldsymbol{x} + b = 0\}$$

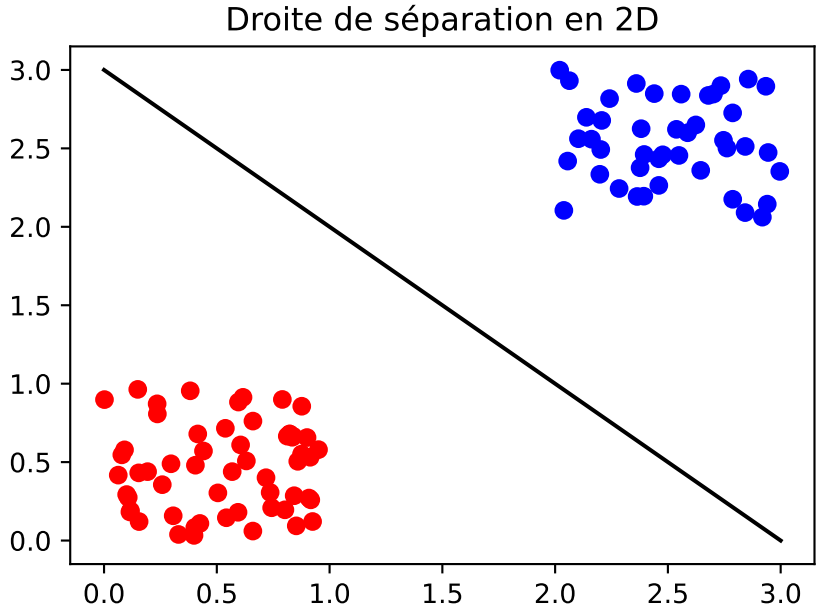
où $\boldsymbol{w}$ est un vecteur orthogonal à l'hyperplan indiquant son orientation et b nous informe sur sa distance à l'origine qui vaut $|b|/\|\boldsymbol{w}\|$.

En dimension $d = 3$, cet hyperplan est en fait un plan, en dimension $d = 2$ il devient une simple droite et se réduit à un point $x = -b/w$ en dimension $d = 1$.

Exemple en 2D

Nous générons 100 points $\boldsymbol{x}_i$ en 2D répartis dans 2 catégories (bleus et rouges). Un classifieur linéaire correspond à une droite qui coupe le plan en deux parties, chacune affectée à une des catégories : $f(\boldsymbol{x}) =$ bleu pour tous les points $\boldsymbol{x}$ au-dessus de la droite et rouge en-dessous.

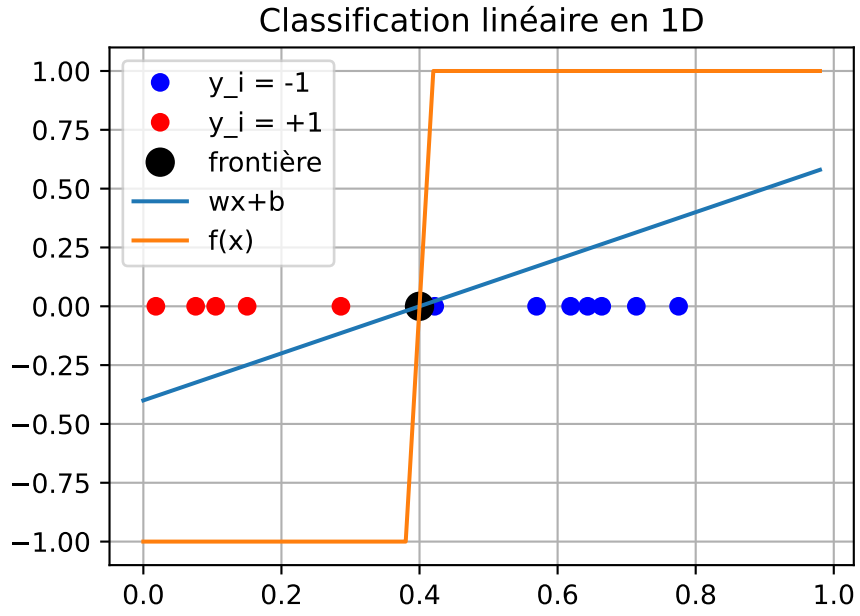
```
X = np.random.rand(100,2)
y = np.random.rand(100) > 0.5
X[y,:] += np.array([2,2])
plt.plot(X[y,0], X[y,1], "ob")
plt.plot(X[~y,0], X[~y,1], "or")
w = np.array([1, 1])
b = -3
x1 = 0
x2 = 3
plt.plot([x1,x2], [-(x1*w[0] + b)/w[1], -(x2*w[0] + b)/w[1]], "-k")
t=plt.title("Droite de séparation en 2D")
```



Exemple en 1D

Un classifieur linéaire de $x \in \mathbb{R}$ est une fonction constante par morceaux qui commute entre $f(x) = -1$ et $f(x) = 1$ lorsque $wx + b = 0$, c'est-à-dire au point $x = -b/w$.

```
w = 1
b = -0.4
X = np.random.rand(12)
y = w*X + b >= 0
plt.plot(X[y],np.zeros(np.sum(y)), "ob")
plt.plot(X[~y], np.zeros(np.sum(~y)), "or")
plt.plot(-b/w,0, "ok", markersize=10)
x = np.arange(0,1,0.02)
plt.plot(x,w*x+b)
plt.plot(x, np.sign(w*x+b))
plt.grid()
plt.legend(["y_i = -1", "y_i = +1", "frontière", "wx+b", "f(x)"])
t=plt.title("Classification linéaire en 1D")
```



Un jeu de données $((\boldsymbol{x}_i, y_i))_{1\leq i\leq m}$ est dit **linéairement séparable** s'il existe un classifieur linéaire capable de classer correctement toutes ces données, autrement dit, s'il existe $(\boldsymbol{w}, b)$ tels que

$$\text{signe}\left(\boldsymbol{w}^T\boldsymbol{x}_i + b\right) = y_i, \quad i = 1, \ldots, m$$

ce qui peut être réécrit comme un système d'inégalités linéaires :

$$y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) \geq 1, \quad i = 1, \ldots, m.$$

Preuve

En classification binaire, $y_i \in \{-1, +1\}$, et donc $y_i = \text{signe}(y_i)$. Ainsi, tester la séparabilité linéaire et trouver $\boldsymbol{w}$ et b tels que signe $\left(\boldsymbol{w}^T \boldsymbol{x}_i + b\right) = y_i$ revient à trouver $\boldsymbol{w}$ et b tels que $\boldsymbol{w}^T \boldsymbol{x}_i + b$ et y_i aient le même signe, ce qui est garanti si

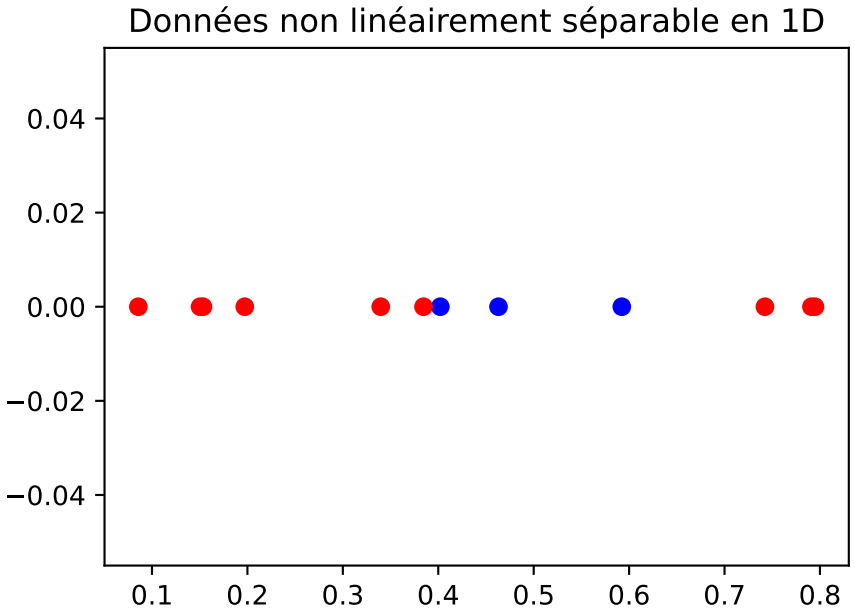
$$y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b) > 0$$

ce qui est bien le cas si $y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b) \geq 1$. Dans le cas où il n'existe pas de couple $(\boldsymbol{w}, b)$ tel que $y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b) \geq 1$ alors il ne peut pas en exister pour satisfaire $y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b) > 0$ (car si $y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b) \geq \epsilon > 0$, alors $(\boldsymbol{w}/\epsilon, b/\epsilon)$ serait solution de $y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b) \geq 1$) et donc le jeu de données est non linéairement séparable.

Exemples de données non linéairement séparables

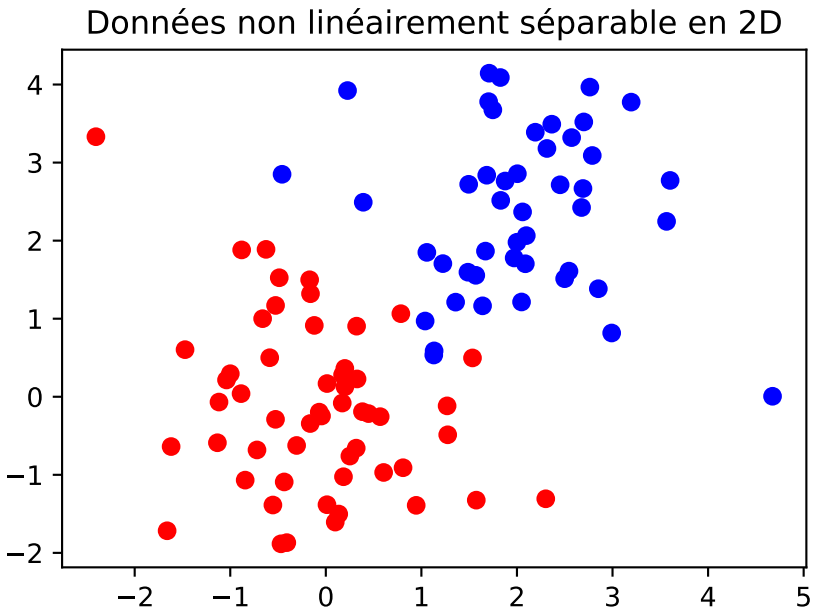
En dimension $d = 1$, un jeu de données est non linéairement séparable si n'existe aucun point sur l'axe des réels tel que tous les points de la base d'apprentissage de chaque côté soient de la même catégorie (couleur).

```
X = np.random.rand(12)
y = (X >= 0.4) & (X < 0.7)
plt.plot(X[y],np.zeros(np.sum(y)), "ob")
plt.plot(X[~y], np.zeros(np.sum(~y)), "or")
t=plt.title("Données non linéairement séparable en 1D")
```



En dimension $d = 2$, un jeu de données est non séaparable si aucune droite ne peut correctement classer tous les exemples.

```
X = np.random.randn(100,2)
y = np.random.rand(100) > 0.5
X[y,:] += np.array([2,2])
plt.plot(X[y,0], X[y,1], "ob")
plt.plot(X[~y,0], X[~y,1], "or")
t=plt.title("Données non linéairement séparable en 2D")
```



8.3 Classification multi-classe

Lorsque le nombre de catégories C est supérieur à 2, le problème de classification devient multi-classe et certaines méthodes ne sont plus directement applicables. Par exemple, un classifieur linéaire de la forme donnée en Équation 8.1 ne peut que diviser l'espace en deux catégories.

Pour pouvoir appliquer un algorithme de classification binaire à un problème multi-classe, il est nécessaire de décomposer le problème en un ensemble de sous-problèmes binaires. Il existe deux grandes approches décrites ci-dessous pour y parvenir.

8.3.1 Décomposition un-contre-tous

La décomposition un-contre-tous crée C classifieurs binaires pour traiter un problème à C catégories, où chaque classifieur binaire $f_k(\boldsymbol{x}) = \text{signe}\,(g_k(\boldsymbol{x}))$ est chargé de séparer une classe de l'ensemble des autres.

La classification multi-classe est obtenue par

$$f(\boldsymbol{x}) = \arg\max_{k \in \mathcal{Y}} g_k(\boldsymbol{x})$$

où la sortie réelle du $k$ème classifieur, $g_k(\boldsymbol{x})$, peut être interprétée comme un score pour la catégorie k, et $\boldsymbol{x}$ est affeté à la catégorie avec le plus grand score.

Cette technique permet de lever l'ambiguïté lorsque plusieurs classifieurs binaires prédisent que $\boldsymbol{x}$ appartient à leur catégorie, ou lorsqu'ils le rejettent tous.

8.3.2 Décomposition un-contre-un

La décomposition un-contre-un crée $C(C-1)/2$ classifieurs binaires, c'est-à-dire un pour chaque paire de catégories. Ainsi, chaque classifieur binaire $f_{j/k}$ est chargé de séparer un couple de catégories et produit une sortie binaire dans $\{j, k\}$.

La classification multi-classe est ensuite obtenue par un vote majoritaire :

$$f(\boldsymbol{x}) = \arg\max_{y \in \mathcal{Y}} \sum_{1 \leq j < k \leq C} \mathbf{1}(f_{j/k}(\boldsymbol{x}) = y)$$

9 Arbres de décision, de régression et forêts aléatoires

Les arbres de décision et de régression sont des outils simples pour l'apprentissage supervisé qui combinent

- l'*interprétabilité* avec des décisions qui peuvent être lues le long de l'arbre et qui n'impliquent que des opérations facilement compréhensibles par les humains ;
- l'efficacité en mémoire avec une représentation en général compacte d'une partition complexe de l'espace ;
- l'efficacité en temps avec des calculs simples et peu nombreux.

9.1 Arbres de décision (pour la classification)

Un arbre de décision implémente un modèle de prédiction où chaque nœud de l'arbre « coupe » les données en 2 paquets (ou plus) en testant une composante x_k de $\boldsymbol{x} \in \mathbb{R}^d$ face à un seuil s :

$$x_k \leq s \ ?$$

puis,

- une branche est créée pour chaque résultat possible du test
- chaque feuille est associée à une étiquette de catégorie y
- pour calculer la prédiction $f(\boldsymbol{x})$, on propage $\boldsymbol{x}$ à travers l'arbre jusqu'à atteindre une feuille et l'étiquette correspondante

9.2 Apprentissage

L'apprentissage d'un arbre de décision vise à construire un arbre en choisissant les coupes de chaque nœud et les étiquettes à associer aux feuilles de telle sorte que le risque empirique (par ex., l'erreur de classification sur les données de la base d'apprentissage) $R_{emp}(f)$ soit minimisé.

Cependant, il s'agit aussi de minimiser la taille de l'arbre, sans quoi une solution simple (et plutôt mauvaise) consisterait à créer une feuille pour chaque exemple $(\boldsymbol{x}_i, y_i)$ pour assurer $R_{emp}(f) = 0$.

L'algorithme d'apprentissage fonctionne au niveau de chaque nœud, qui a accès à un sous-ensemble des données d'apprentissage (les $\boldsymbol{x}_i$ qui atteignent ce nœud en suivant les coupes depuis la racine).

1. Créer la racine en lui associant toutes les données d'apprentissage
2. Pour chaque nœud non traité :
    - Calculer l'étiquette y majoritaire sur les données du nœud et la stocker
    - Calculer le nombre d'erreurs de classification du nœud sur ses données. Si il est tolérable, marquer le nœud comme une feuille, sinon
        1. Déterminer la composante x_k et le seuil s pour couper les données
        2. Créer les nœuds fils avec leurs données associées en fonction de la coupe choisie

Dans cet algorithme, un paramètre important est la **tolérance sur l'erreur** dans les feuilles, qui limite le nombre de nœuds et permet d'éviter le surapprentissage.

Dans chaque nœud, x_k et s sont choisis pour minimiser l'**impureté** des fils :

$$\min_{k,s} n_1 \times Impur(fils_1) + n_2 \times Impur(fils_2)$$

avec l'impureté $Impur(fils_j)$ du fils j associé à n_j données qui peut être définie comme

- le taux d'erreur de classification : $1 - p_y$
- l'index de Gini : $\sum_{c=1}^{C} p_c(1 - p_c)$
- l'entropie : $-\sum_{c=1}^{C} p_c \log p_c$

où p_c est la proportion d'exemples de la catégorie $c \in \{1, \ldots, C\}$ dans le nœud et y est la catégorie majoritaire.

9.2.1 Recherche de la coupe optimale

Pour les composantes continues ($x_k \in \mathbb{R}$), il existe une infinité de seuils s possibles pour une coupe. Mais en réalité, il suffit de tester les $m - 1$ valeurs de seuil :

$$s_i = \frac{x_{i,k} + x_{i+1,k}}{2}$$

où les $k$èmes composantes des données, $x_{i,k}$, sont triées par ordre croissant.

En effet, les autres valeurs de seuil ne donnent pas naissance à d'autres classifications des données différentes de celles obtenues avec les s_i, et donc conduiront aux même valeurs d'impureté.

À chaque itération (à chaque nœud), le nombre de couples (k, s) à tester et pour lesquelles il faut calculer l'impureté résultante est donc $d(m - 1)$.

9.2.2 Variables qualitatives

Si $\boldsymbol{x}$ contient des composantes qualitatives (à valeurs dans un ensemble discret, par ex. $x_k \in \{rouge, vert, jaune\}$), l'algorithme s'adapte simplement en découpant les branches selon les modalités :

- soit en créant une branche par valeur possible,
- soit en coupant en deux paquets de valeurs (ou une seule vs les autres).

9.3 Partition de l'espace de représentation

Un arbre de décision (ou de régression) équivaut à une partition de $\mathcal{X}$ en régions $\mathcal{X}_j$ rectangulaires associées aux feuilles :

$$f(\boldsymbol{x}) = y \quad \Leftrightarrow \quad \exists j, \quad \boldsymbol{x} \in \mathcal{X}_j \text{ et } y(j) = y$$

avec $\mathcal{X}_j = [a_1, b_2] \times [a_2, b_2] \times \cdots \times [a_d, b_d]$, les a_k, b_k déterminés par les coupes dans chaque nœud, et $y(j)$ l'étiquette de la feuille j.

9.4 Arbres de régression

En régression, les étiquettes sont réelles : $y_i \in \mathbb{R}$. Dans ce cas, l'algorithme ci-dessus fonctionne avec les modifications suivantes :

- l'étiquette $y(noeud)$ d'un nœud est la moyenne des y_i des données affectées au nœud (d'indices $\mathcal{I}(noeud)$) ;
- le modèle implémente une fonction $f(\boldsymbol{x})$ en escalier (constante par morceaux) ;
- le critère à minimiser pour choisir une coupe est l'erreur quadratique moyenne :

$$Impur(noeud) = \frac{1}{|\mathcal{I}(noeud)|} \sum_{i \in \mathcal{I}(noeud)} (y_i - y(noeud))^2$$

$$\Rightarrow \qquad \min_{k,s} \sum_{i \in \mathcal{I}(fils_1)} (y_i - y(fils_1))^2 + \sum_{i \in \mathcal{I}(fils_2)} (y_i - y(fils_2))^2$$

On peut aussi créer des arbres de régression avec des modèles linéaires locaux associés aux feuilles au lieu de simples constantes (le modèle devient alors une fonction linéaire par morceaux au lieu d'un escalier).

9.5 Forêts aléatoires

Les arbres de décision et de régression ont de nombreux avantages, mais sont à l'inverse très sensisbles aux données. Ainsi, on observe une grande variance pour l'apprentissage d'un arbre : quelques modifications minimales des données peuvent conduirent à des arbres très différents.

Pour limiter cet effet, les techniques de *bagging* et de *forêts aléatoires* construisent un ensemble d'arbres dont les sorties sont moyennées afin d'en réduire la variance. En effet, la variance d'une moyenne μ de B variables aléatoires indépendantes et identiquement distribuées (i.i.d.) Z_i est divisée par B :

$$\mu = \frac{1}{B}\sum_{i=1}^{B} Z_i \quad \Rightarrow \quad Var(\mu) = \frac{Var(Z_1)}{B} \tag{9.1}$$

Preuve

Les propriétés de la variance permettent d'obtenir

$$Var(\mu) = \frac{1}{B^2} Var\left(\sum_{i=1}^{B} Z_i\right)$$

et, pour des variables i.i.d., puisque $Var(Z_1 + Z_2) = Var(Z_1) + Var(Z_2)$,

$$Var\left(\sum_{i=1}^{B} Z_i\right) = \sum_{i=1}^{B} Var\left(Z_i\right)$$

et

$$Var(\mu) = \frac{1}{B^2}\sum_{i=1}^{B} Var\left(Z_1\right) = \frac{BVar(Z_1)}{B^2} = \frac{Var(Z_1)}{B}$$

Ainsi, il suffirait d'apprendre B arbres f_j différents pour calculer des prédictions

$$f(\boldsymbol{x}) = \frac{1}{B}\sum_{j=1}^{B} f_j(\boldsymbol{x})$$

ou, pour la classification,

$$f(\boldsymbol{x}) = \text{catégorie majoritaire parmi les } f_j(\boldsymbol{x}),$$

qui seraient moins sensibles aux petites modifications des données et donc potentiellement meilleures en généralisation. Cependant, pour apprendre des arbres différents, il faut des jeux de données différents.

9.5.1 Bootstrap

Le bootstrap est une techniue classique en statistique pour générer plusieurs jeux de données à partir d'un seul.

Avec n données de départ, un nouveau jeu de données peut être créé en tirant aléatoirement n' données parmi n avec remise (les mêmes données peuvant donc être choisies plusieurs fois).

Cette opération peut être répétée autant de fois que nécessaire pour générer un ensemble de variations du jeu de données initial.

9.5.2 Bagging

Le *bagging* (pour *bootstrap aggregation*) revient à apprendre B arbres sur B jeux de données générés par bootstrap avec $n' = n$.

Cette approche est assez simple et directe à mettre en place, et peut s'utiliser en fait avec d'autres modèles que les arbres. En règle générale, on parle de « modèles faibles » dont les performances sont améliorées par bagging.

Un inconvénient de cette méthode simple est que les différents modèles faibles (ou les arbres) ne sont pas indépendants les uns des autres. En effet, les B jeux de données étant tous générés à partir des mêmes données initiales, ils sont dépendants les uns des autres.

Cela remet en question l'intérêt de l'approche suggérée par la formule de réduction de la variance Équation 9.1, qui n'est valide que pour des variables indépendantes. Dans le cas général, la covariance s'en mêle et la formule devient :

$$Var(\mu) = \frac{Var(Z_1)}{n} + \left(1 - \frac{1}{n}\right) C, \qquad C = Cov(Z_i, Z_j)$$

ce qui peut être bien moins intéressant si la covariance est grande.

9.5.3 Forêts aléatoires (*Random forests*)

Pour limiter l'impact de la covariance sur la réduction de la variance par moyennage, l'idée est de réduire la dépendance des B arbres composant le modèle global.

L'algorithme des forêts aléatoires y parvient en injectant arbitrairement de l'aléatoire dans l'apprentissage de chaque arbre. Plus précisément, à chaque création de nœud, la recherche d'une coupe optimale se limite à un sous-ensemble de p variables choisies aléatoirement parmi les d composantes de $\boldsymbol{x}$.

À noter cependant que ces techniques (bagging ou forêts aléatoires) ne permettent plus d'interpréter le modèle aussi facilement, puisque celui-ci est maintenant une moyenne d'un (potentiellement grand) ensemble de modèles créés aléatoirement.

10 Analyse discriminante Linéaire (ADL)

L'analyse disciminante linéaire peut s'intérpréter comme une méthode générative, c'est-à-dire basée sur un modèle génératif de densité conditionnelle $p(\boldsymbol{x}|y)$ et la règle de décision optimale

$$f(\boldsymbol{x}) = \arg\max_{y \in \mathcal{Y}} P(Y = y | \boldsymbol{X} = x)$$

qui peut se réécrire grâce à la règle de Bayes comme

$$f(\boldsymbol{x}) = \arg\max_{y \in \mathcal{Y}} \frac{p(\boldsymbol{x}|y)P(Y = y)}{p(\boldsymbol{x})} = \arg\max_{y} p(\boldsymbol{x}|y)P(Y = y)$$

où le dénominateur est constant par rapport à y et peut être ignorer pour la classification.

Cependant, les probabilités et densités ci-dessus ne peuvent en pratique qu'être estimées et, bien que s'inspirant du classifieur de Bayes, le classifieur basé sur les estimations ne sera pas nécessairement optimal.

10.1 Hypothèses sur le modèle génératif

La méthode ADL repose sur l'hypothèse qu'à l'intérieur de chaque catégorie, les données sont distribuées selon une loi gaussienne, soit pour une seule variable $x \in \mathbb{R}$:

$$p(x \mid Y = y) = \frac{1}{\sigma_y \sqrt{2\pi}} \exp\left(-\frac{(x - \mu_y)^2}{2\sigma_y^2}\right) \qquad \begin{cases} \mu_y & : \textit{moyenne} \\ \sigma_y^2 & : \textit{variance} \end{cases}$$

Avec $\boldsymbol{x} \in \mathbb{R}^d$ en dimension d, cela devient

$$p(\boldsymbol{x} \mid Y = y) = \frac{1}{(2\pi)^{d/2} |\boldsymbol{C}_y|^{1/2}} \exp\left(-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu}_y)^T \boldsymbol{C}_y{}^{-1} (\boldsymbol{x} - \boldsymbol{\mu}_y)\right)$$

De plus une hypothèse supplémentaire est faite : la variance ou covariance ($\boldsymbol{C}_y$) est constante sur toutes les catégories y :

$$\sigma_y^2 = \sigma^2 \qquad \boldsymbol{C}_y = \boldsymbol{C}$$

10.2 ADL et classification linéaire dans le cas binaire

Dans le cas binaire, nous pouvons reformuler le modèle de l'ADL comme un classifieur linéaire :

$$\begin{aligned} f(\boldsymbol{x}) &= \arg\max_{y\in\mathcal{Y}} p(\boldsymbol{x}|Y=y)P(Y=y) \\ &= \begin{cases} +1, & \text{si } p(\boldsymbol{x}|Y=1)P(Y=1) \geq p(\boldsymbol{x}|Y=-1)P(Y=-1) \\ -1, & \text{sinon} \end{cases} \\ &= \text{signe}\left(\boldsymbol{w}^T\boldsymbol{x}+b\right) \end{aligned}$$

avec

$$\boldsymbol{w} = \boldsymbol{C}^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_{-1}), \qquad b = \frac{1}{2}(\boldsymbol{\mu}_{-1}+\boldsymbol{\mu}_1)^T\boldsymbol{C}^{-1}(\boldsymbol{\mu}_{-1}-\boldsymbol{\mu}_1) + \log\frac{m_1}{m_{-1}}$$

Preuve

$$\begin{aligned} f(\boldsymbol{x}) &= \arg\max_{y\in\mathcal{Y}} p(\boldsymbol{x}|Y=y)P(Y=y) \\ &= \begin{cases} +1, & \text{si } p(\boldsymbol{x}|Y=1)P(Y=1) \geq p(\boldsymbol{x}|Y=-1)P(Y=-1) \\ -1, & \text{sinon} \end{cases} \\ &= \begin{cases} +1, & \text{si } \frac{p(\boldsymbol{x}|Y=1)P(Y=1)}{p(\boldsymbol{x}|Y=-1)P(Y=-1)} \geq 1 \\ -1, & \text{sinon} \end{cases} = \text{signe}\left(\underbrace{\log\frac{p(\boldsymbol{x}|Y=1)P(Y=1)}{p(\boldsymbol{x}|Y=-1)P(Y=-1)}}_{g(\boldsymbol{x})}\right) \end{aligned}$$

$$g(\boldsymbol{x}) = \underbrace{\log\frac{p(\boldsymbol{x}|Y=1)}{p(\boldsymbol{x}|Y=-1)}}_{L(\boldsymbol{x})} + \underbrace{\log\frac{P(Y=1)}{P(Y=-1)}}_{c=\log\frac{m_1/m}{m_{-1}/m}=\log\frac{m_1}{m_{-1}}}$$

$$
\begin{aligned}
L(\boldsymbol{x}) &= \log p(\boldsymbol{x}|Y=1) - \log p(\boldsymbol{x}|Y=-1) \\
&= \log \frac{1}{(2\pi)^{d/2}|\boldsymbol{C}|^{1/2}} + \log \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_1)^T\boldsymbol{C}^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_1)\right) \\
&\quad - \log \frac{1}{(2\pi)^{d/2}|\boldsymbol{C}|^{1/2}} - \log \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_{-1})^T\boldsymbol{C}^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_{-1})\right) \\
&= \cancel{\log \frac{1}{(2\pi)^{d/2}|\boldsymbol{C}|^{1/2}}} + \log \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_1)^T\boldsymbol{C}^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_1)\right) \\
&\quad \cancel{- \log \frac{1}{(2\pi)^{d/2}|\boldsymbol{C}|^{1/2}}} - \log \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_{-1})^T\boldsymbol{C}^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_{-1})\right) \\
&= -\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_1)^T\boldsymbol{C}^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_1) + \frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_{-1})^T\boldsymbol{C}^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_{-1}) \\
&= \frac{1}{2}\boldsymbol{x}^T\boldsymbol{C}^{-1}\boldsymbol{x} - \boldsymbol{\mu}_{-1}^T\boldsymbol{C}^{-1}\boldsymbol{x} + \frac{1}{2}\boldsymbol{\mu}_{-1}^T\boldsymbol{C}^{-1}\boldsymbol{\mu}_{-1} - \frac{1}{2}\boldsymbol{x}^T\boldsymbol{C}^{-1}\boldsymbol{x} + \boldsymbol{\mu}_1^T\boldsymbol{C}^{-1}\boldsymbol{x} - \frac{1}{2}\boldsymbol{\mu}_1^T\boldsymbol{C}^{-1}\boldsymbol{\mu}_1 \\
&= \underbrace{(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_{-1})^T\boldsymbol{C}^{-1}}_{\boldsymbol{w}^T}\boldsymbol{x} + \frac{1}{2}(\boldsymbol{\mu}_{-1} + \boldsymbol{\mu}_1)^T\boldsymbol{C}^{-1}(\boldsymbol{\mu}_{-1} - \boldsymbol{\mu}_1)
\end{aligned}
$$

Donc,

$$f(\boldsymbol{x}) = \text{signe}(g(\boldsymbol{x}))$$

avec

$$g(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} + b \qquad \boldsymbol{w} = \boldsymbol{C}^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_{-1}), \qquad b = \frac{1}{2}(\boldsymbol{\mu}_{-1} + \boldsymbol{\mu}_1)^T\boldsymbol{C}^{-1}(\boldsymbol{\mu}_{-1} - \boldsymbol{\mu}_1) + \log\frac{m_1}{m_{-1}}$$

10.3 Apprentissage

L'apprentissage de l'ADL correspond à l'estimation des paramètres des lois gaussiennes d'une part et des probabilités a priori d'autre part.

Pour les probabilités a priori, elles peuvent être simplement estimées par les pourcentages d'exemples de chaque catégorie :

$$P(Y=y) = \frac{m_y}{m}$$

avec m_y le nombre d'exemples de la classe y dans la base d'apprentissage et m le nombre total d'exemples.

Pour les lois gaussiennes, nous avons deux types de paramètres : les moyennes $\boldsymbol{\mu}_y$ et la matrice de covariance $\boldsymbol{C}$.

Les moyennes sont simplement estimées par les moyennes empiriques :

$$\boldsymbol{\mu}_y = \frac{1}{m_y}\sum_{y_i=y}\boldsymbol{x}_i$$

La matrice de variance-covariance au sein d'une catégorie peut être estimée par :

$$\boldsymbol{C}_y = \frac{1}{m_y - 1} \sum_{y_i = y} (\boldsymbol{x}_i - \boldsymbol{\mu}_y)(\boldsymbol{x}_i - \boldsymbol{\mu}_y)^T$$

Puis, la matrice $\boldsymbol{C}$ est obtenue en calculant une moyenne de ces matrices pondérées par les nombres d'exemples :

$$\boldsymbol{C} = \frac{\sum_{y \in \mathcal{Y}} (m_y - 1) \boldsymbol{C}_y}{\sum_{y \in \mathcal{Y}} (m_y - 1)}$$

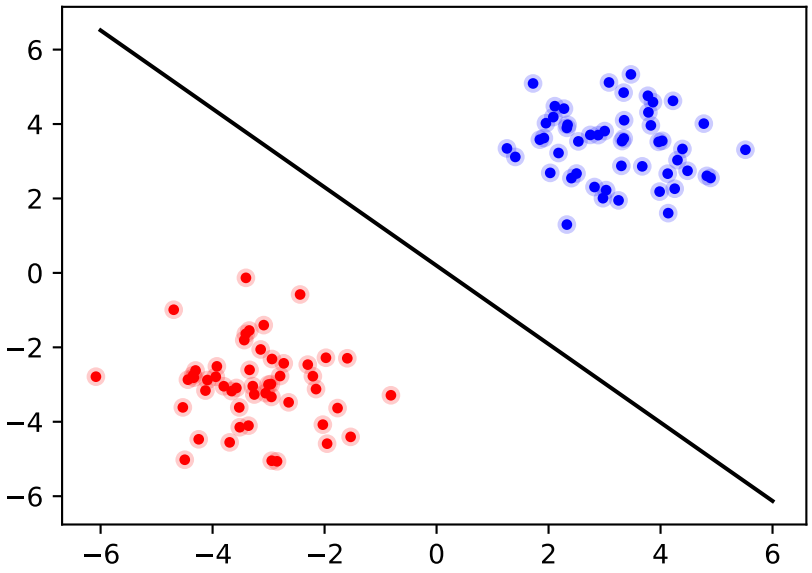
Enfin, il peut être plus efficace dans le cas binaire de calculer $f(\boldsymbol{x})$ à partir de la forme linéaire en précalculant $\boldsymbol{w} = \boldsymbol{C}^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_{-1})$ par résolution du système linéaire $\boldsymbol{C}\boldsymbol{w} = \boldsymbol{\mu}_1 - \boldsymbol{\mu}_{-1}$ plutôt qu'inversion directe de la matrice $\boldsymbol{C}$. Ensuite, b est donné par $\log \frac{m_1}{m_{-1}} - \frac{1}{2}(\boldsymbol{\mu}_{-1} + \boldsymbol{\mu}_1)^T \boldsymbol{w}$.

Exemple en python

```
# Création des données (en respectant les hypothèses)
X1 = np.random.randn(50, 2) + np.array([3,3])
X2 = np.random.randn(50, 2) + np.array([-3,-3])
X = np.vstack([X1,X2])
plt.plot(X1[:,0], X1[:,1], '.b')
plt.plot(X2[:,0], X2[:,1], '.r')

# Estimation des paramètres
m1 = len(X1)
m2 = len(X2)
mu1 = np.mean(X1, axis=0)
mu2 = np.mean(X2, axis=0)
print("Catégorie 1 : " + str(m1) + " exemples de moyenne", mu1)
print("Catégorie 2 : " + str(m2) + " exemples de moyenne", mu2)

C = ( (m1-1) * np.cov(X1.T) + (m2-1) * np.cov(X2.T) ) / (m1+m2-2)
print("Matrice de variance-covariance unique :\n", C)

w = np.linalg.solve(C, mu1 - mu2)
b = np.log(m1/m2) - 0.5 * (mu1+mu2) @ w

# Classification des données
ypred = np.sign( X@w + b )

plt.plot(X[ypred==1,0], X[ypred==1,1], 'ob', alpha=0.2)
plt.plot(X[ypred==-1,0], X[ypred==-1,1], 'or', alpha=0.2)
plt.plot([-6,6], [(w[0]*6-b)/w[1], (-w[0]*6-b)/w[1]], "-k")
plt.show()
```

```
Catégorie 1 : 50 exemples de moyenne [3.19879259 3.42629177]
```

```
Catégorie 2 : 50 exemples de moyenne [-3.24049098 -2.99361089]
Matrice de variance-covariance unique :
 [[ 0.98201701 -0.0983634 ]
 [-0.0983634   1.03729117]]
```



11 Classifieur naïf de Bayes

Le classification naïf de Bayes repose, comme l'analyse discrimnante linéaire, sur un modèle génératif et la règle de décision optimale

$$f(\boldsymbol{x}) = \arg\max_{y \in \mathcal{Y}} P(Y = y | \boldsymbol{X} = \boldsymbol{x})$$

qui peut se réécrire grâce à la règle de Bayes comme

$$f(\boldsymbol{x}) = \arg\max_{y \in \mathcal{Y}} \frac{p(\boldsymbol{x}|y)P(Y = y)}{p(\boldsymbol{x})} = \arg\max_{y} p(\boldsymbol{x}|y)P(Y = y)$$

où le dénominateur est constant par rapport à y et peut être ignorer pour la classification.

11.1 Hypothèses sur le modèle génératif

Le classifieur naïf de Bayes ne présume pas d'un modèle génératif particulier : il peut s'appliquer à différents modèles. En revanche, une hypothèse forte est ajoutée : l'**indépendance conditionnelle des composantes** x_k de $\boldsymbol{x} = [x_1, \ldots, x_d]^T \in \mathbb{R}^d$, c'est-à-dire

$$p(\boldsymbol{x}|Y = y) = \prod_{k=1}^{d} p(x_k|Y = y)$$

Par exemple, pour un modèle gaussien

$$p(\boldsymbol{x} \mid Y = y) = \frac{1}{(2\pi)^{d/2}|\boldsymbol{C}_y|^{1/2}} \exp\left(-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu}_y)^T \boldsymbol{C}_y{}^{-1}(\boldsymbol{x} - \boldsymbol{\mu}_y)\right),$$

les paramètres à estimer sont au nombre de d (pour la moyenne $\boldsymbol{\mu}_y$) + d^2 (pour $\boldsymbol{C}_y$) pour chaque catégorie. Mais sous l'hypothèse d'indépendance conditionnelle, cela se ramène à l'estimation de d gaussiennes unidimensionnelles

$$p(x_k \mid Y = y) = \frac{1}{\sigma_{y,k}\sqrt{2\pi}} \exp\left(-\frac{(x - \mu_{y,k})^2}{2\sigma_{y,k}^2}\right)$$

et donc à d estimations indépendantes de 2 paramètres.

11.2 Application à des données binaires

Pour $\boldsymbol{X} \in \{0,1\}^d$, sous l'hypothèse d'indépendance des composantes, nous pouvons écrire

$$\begin{aligned} P(\boldsymbol{X} = \boldsymbol{x}|Y = y) &= \prod_{k=1}^{d} P(X_k = x_k|Y = y) \\ &= \prod_{k=1}^{d} (b_k^y)^{x_k}(1 - b_k^y)^{1-x_k} \end{aligned}$$

car chaque composante X_k peut être modélisée par une loi de Bernouilli :

$$P(X_k = 1|Y = y) = b_k^y, \quad P(X_k = 0|Y = y) = 1 - b_k^y$$

$$\Rightarrow P(X_k = x_k|Y = y) = (b_k^y)^{x_k}(1 - b_k^y)^{1-x_k}$$

Les paramètres b_k^y correspondant à de simples probabilités d'occurrence de $x_k = 1$, ils peuvent être simplement estimés par les pourcentages d'exemples pour lesquel $x_k = 1$:

$$b_k^y = \frac{\#exemples\ de\ la\ cat.\ y\ avec\ x_k = 1}{m_y}$$

Cependant, pour éviter $b_k^y = 0$ ou $b_k^y = 1$ qui risquerait de compromettre le calcul du produit $P(\boldsymbol{X} = \boldsymbol{x}|Y = y)$, il est nécessaire de régulariser, avec pour $\epsilon > 0$ (typiquement $\epsilon = 1$) :

$$b_k^y = \frac{(\#exemples\ de\ la\ cat.\ y\ avec\ x_k = 1) + \epsilon}{m_y + 2\epsilon}$$

Exemple d'application : filtre anti-spam

Le classifieur naïf de Bayes est très souvent utilisé dans la filtres anti-spam pour classer les mails en deux catégories : SPAM ou HAM (mail légitime). Dans ce contexte, il est courant de représenter un mail par un sac de mots : un vecteur $\boldsymbol{x}$ binaire de la taille d d'un dictionnaire prédéfini où chaque composante x_k indique la présence du $k$ème mot du dictionnaire.
Dans ce cas, les paramètres b_k^{SPAM} et b_k^{HAM} correspondent aux probabilité de voir le $k$ème mot du dictionnaire dans un SPAM ou un HAM. Les probabilités $P(Y = SPAM)$ et $P(Y = HAM)$ sont estimées par

$$P(Y = SPAM) = \frac{\#spams}{m} \qquad P(Y = HAM) = 1 - P(Y = SPAM)$$

et correspondent à la probabilité qu'un mail quelconque soit un SPAM ou HAM.
Dans ce type d'applications, la dimension d correspond à la taille du dicitonnaire et peut être très grande et le produit $P(\boldsymbol{X} = \boldsymbol{x}|Y = y) = \prod_{k=1}^{d}(b_k^y)^{x_k}(1 - b_k^y)^{1-x_k}$ de d

nombres entre 0 et 1 peut être trop proche de zéro pour permettre de calculer une classification correcte sur une machine à précision limitée. Il est donc courant de passer par les log car

$$f(\boldsymbol{x}) = \arg\max_{y\in\mathcal{Y}} P(\boldsymbol{X}=\boldsymbol{x}|Y=y)P(Y=y) = \arg\max_{y\in\mathcal{Y}} \log P(\boldsymbol{X}=\boldsymbol{x}|Y=y)+\log P(Y=y)$$

et

$$\log P(\boldsymbol{X}=\boldsymbol{x}|Y=y) = \sum_{k=1}^{d} \log(b_k^y)^{x_k}(1-b_k^y)^{1-x_k} = \sum_{x_k=1} \log b_k^y + \sum_{x_k=0} \log(1-b_k^y)$$

est une somme qui aura tendance à rester dans des plages de valeurs beaucoup plus raisonnables.

```
from math import log

# imaginons que tous les termes du produit valent
b = 0.9

print("valeur de P(X=x|Y=y) pour d=1 000, 5 000 et 10 000 : ")
print(b**1000)
print(b**5000)
print(b**10000)

print("valeur de log P(X=x|Y=y) pour d=1 000, 5 000 et 10 000 : ")
print(1000 * log(b))
print(5000 * log(b))
print(10000 * log(b))
```

```
valeur de P(X=x|Y=y) pour d=1 000, 5 000 et 10 000 :
1.7478712517226947e-46
1.631350185342827e-229
0.0
valeur de log P(X=x|Y=y) pour d=1 000, 5 000 et 10 000 :
-105.36051565782628
-526.8025782891314
-1053.6051565782627
```

Pour $d = 10000$, sans passer par les log, la classification sera donc donnée par le résultat de la comparaison de 0 avec 0, ce qui n'est pas raisonnable, alors que la comparaison des log restera correcte.

12 Perceptron

Le perceptron est un modèle de classification linéaire qui s'écrit

$$f(\boldsymbol{x}) = \begin{cases} +1, & \text{si } \boldsymbol{\theta}^T \boldsymbol{x} \geq \theta_0 \\ -1, & \text{sinon} \end{cases}$$

où les paramètres $\boldsymbol{\theta} \in \mathbb{R}^d$ et θ_0 sont respectivement les poids et le seuil de déclenchement du modèle et correspondent aux paramètres $\boldsymbol{w}$ et $-b$ d'un classifieur linéaire tel que décrit ici.

12.1 Apprentissage des poids (à seuil fixe)

L'algorithme d'apprentissage du Perceptron pour un seuil $\theta_0 = 0$ constant se définit ainsi :

1. Initialisation (aléatoire) des poids $\boldsymbol{\theta}$
2. Répéter jusqu'à convergence :
   - Pour i de 1 à m

$$\boldsymbol{\theta} \leftarrow \begin{cases} \boldsymbol{\theta}, & \text{si } f(\boldsymbol{x}_i) = y_i \\ \boldsymbol{\theta} + y_i \boldsymbol{x}_i, & \text{sinon} \end{cases}$$

     (chaque itération sur la base d'apprentissage est appelée une *epoch*)

Cet algorithme peut s'interpréter ainsi :

- tous les exemples de la base d'apprentissage sont classés tour à tour ;
- pour chaque erreur, une correction positive est appliquée aux poids si f ne prend pas assez en compte l'exemple $\boldsymbol{x}_i$ étiqueté positivement, et inversement dans le cas d'un exemlpe négatif.

Justification de l'algorithme

Si $y_i = 1$, alors la correction n'est appliquée que si $f(\boldsymbol{x}_i) = -1$ et donc $\boldsymbol{\theta}^T \boldsymbol{x}_i < 0$ et la correction devrait permettre d'augmenter la valeur de $\boldsymbol{\theta}^T \boldsymbol{x}_i$ pour la faire éventuellement passer au-dessus de zéro.
Après correction, cette quantité vaut

$$(\boldsymbol{\theta} + y_i \boldsymbol{x}_i)^T \boldsymbol{x}_i = \underbrace{\boldsymbol{\theta}^T \boldsymbol{x}_i}_{\text{ancienne valeur}} + y_i \boldsymbol{x}_i^T \boldsymbol{x}_i$$

où $\boldsymbol{x}_i^T \boldsymbol{x}_i = \|\boldsymbol{x}_i\|^2 \geq 0$. Donc la valeur de $\boldsymbol{\theta}^T \boldsymbol{x}_i$ sera augmentée si $y_i = +1$ et diminuée

si $y_i = -1$, ce qui rapprochera le classifieur d'une bonne classification sur cet exemple.

12.2 Convergence de l'algorithme

- Si la base d'apprentissage n'est **pas** linéairement séparable, l'algorithme du perceptron ne peut pas converger, car il y aura toujours des erreurs et donc des corrections appliquées aux poids. C'est pour cela qu'en pratique on applique toujours cet algorithme avec un **nombre maximum d'epochs**.
- Si la base d'apprentissage est linéairement séparable, l'algorithme du perceptron converge en un **nombre fini d'itérations**, qui dépend de la marge avec laquelle un classifieur linéaire peut classer les données, et donc de la plus petite distance entre deux points de catégories différentes.

🔥 Preuve de convergence du Perceptron

Dans le cas linéairement séparable, il existe une solution $\boldsymbol{\theta}^*$ de norme unité ($\|\boldsymbol{\theta}^*\| = 1$), qui classe correctement tous les exemples avec une marge d'au moins γ, c'est-à-dire que

$$y_i \boldsymbol{x}_i^T \boldsymbol{\theta}^* \geq \gamma, \quad i = 1, \ldots, m.$$

Soit $\boldsymbol{\theta}_t$ le vecteur des poids après t corrections. La convergence vers $\boldsymbol{\theta}^*$ à partir d'une initialiation à $\boldsymbol{\theta}_0 = \mathbf{0}$ sur des données bornées ($\|\boldsymbol{x}_i\| \leq R$) se démontre en combinant deux idées :

1. le produit scalaire $\boldsymbol{\theta}_t^T \boldsymbol{\theta}^*$ augmente à chaque itération ;
2. la norme de $\boldsymbol{\theta}_t$ n'augmente pas trop à chaque itération ;

et donc $\boldsymbol{\theta}_t$ est de plus en plus aligné avec $\boldsymbol{\theta}^*$, ce qui revient à dire qu'ils produisent la même classification.
Montrons le point 1. :

$$\boldsymbol{\theta}_{t+1}^T \boldsymbol{\theta}^* = (\boldsymbol{\theta}_t + y_i \boldsymbol{x}_i)^T \boldsymbol{\theta}^* = \boldsymbol{\theta}_t^T \boldsymbol{\theta}^* + y_i \boldsymbol{x}_i^T \boldsymbol{\theta}^* \geq \gamma + \boldsymbol{\theta}_t^T \boldsymbol{\theta}^*.$$

Après t itérations, nous avons donc (avec $\boldsymbol{\theta}_0 = \mathbf{0}$)

$$\boldsymbol{\theta}_t^T \boldsymbol{\theta}^* \geq t\gamma + \boldsymbol{\theta}_0^T \boldsymbol{\theta}^* = t\gamma.$$

Pour le point 2., nous avons (en utilisant $|y_i| = 1$):

$$\|\boldsymbol{\theta}_{t+1}\|^2 = \|\boldsymbol{\theta}_t + y_i \boldsymbol{x}_i\|^2 \leq \|\boldsymbol{\theta}_t\|^2 + \|y_i \boldsymbol{x}_i\|^2 = \|\boldsymbol{\theta}_t\|^2 + R^2$$

Après t itérations, nous avons donc

$$\|\boldsymbol{\theta}_t\|^2 \leq tR^2$$

Combinant ces deux résultats et l'inégalité de Cauchy-Schwarz, nous obtenons

$$t\gamma \leq \boldsymbol{\theta}_t^T \boldsymbol{\theta}^* \leq \|\theta_t\| \|\boldsymbol{\theta}^*\| \leq \sqrt{t} R$$

et donc

$$t \leq \frac{R^2}{\gamma^2}$$

12.3 Apprentissage des poids et du seuil

Pour permettre l'apprentissage du seuil, il suffit d'appliquer l'algorithme précédent à des données étendues avec une composante supplémentaire égale à -1 :

$$\boldsymbol{\theta}^T\boldsymbol{x} \geq \theta_0 \;\Leftrightarrow\; \boldsymbol{\theta}^T\boldsymbol{x} - \theta_0 \geq 0 \;\Leftrightarrow \begin{bmatrix}\boldsymbol{\theta}^T & \theta_0\end{bmatrix}\begin{bmatrix}\boldsymbol{x}\\ -1\end{bmatrix} \geq 0 \;\Leftrightarrow\; \tilde{\boldsymbol{\theta}}^T\tilde{\boldsymbol{x}} \geq 0$$

Ainsi, en appliquant l'algorithme de base à $\tilde{\boldsymbol{\theta}}$, nous obtenons

$$\boldsymbol{\theta} \leftarrow \begin{cases}\boldsymbol{\theta}, & \text{si } f(\boldsymbol{x}_i) = y_i\\ \boldsymbol{\theta} + y_i\boldsymbol{x}_i, & \text{sinon}\end{cases} \quad \text{et} \quad \theta_0 \leftarrow \begin{cases}\theta_0, & \text{si } f(\boldsymbol{x}_i) = y_i\\ \theta_0 - y_i, & \text{sinon}\end{cases}$$

En pratique, un **taux d'apprentissage** $\mu \in (0, 1]$ est inséré dans les mises à jour pour accélérer la convergence

$$\boldsymbol{\theta} \leftarrow \begin{cases}\boldsymbol{\theta}, & \text{si } f(\boldsymbol{x}_i) = y_i\\ \boldsymbol{\theta} + \mu y_i\boldsymbol{x}_i, & \text{sinon}\end{cases} \quad \text{et} \quad \theta_0 \leftarrow \begin{cases}\theta_0, & \text{si } f(\boldsymbol{x}_i) = y_i\\ \theta_0 - \mu y_i, & \text{sinon}\end{cases}$$

Le taux d'apprentissage joue le même rôle que le paramètre μ de la descente de gradient.

13 K-plus proches voisins

L'algorithme des K-plus proches voisins (Kppv) est une méthode de classification et de régression assez simple et intuitive, basée sur la notion de voisinage.

Ici, le voisinage $\mathcal{V}_K(\boldsymbol{x})$ d'un point $\boldsymbol{x}$ est l'ensemble des K points $\boldsymbol{x}_i$ de la base d'apprentissage les plus proches de $\boldsymbol{x}$ au sens d'une certaine distance, typiquement la distance euclidienne $\|\boldsymbol{x}_i - \boldsymbol{x}\|$.

13.1 K-plus proches voisins pour la classification

En classification, l'algorithme des K-plus proches voisins classe l'exemple $\boldsymbol{x}$ dans la catégorie majoritaire au sein de son voisinage. Pour un problème à C catégories, cela donne :

$$f(\boldsymbol{x}) = \arg \max_{k=1,\dots,C} \sum_{\boldsymbol{x}_i \in \mathcal{V}_K(\boldsymbol{x})} \mathbf{1}(y_i = k)$$

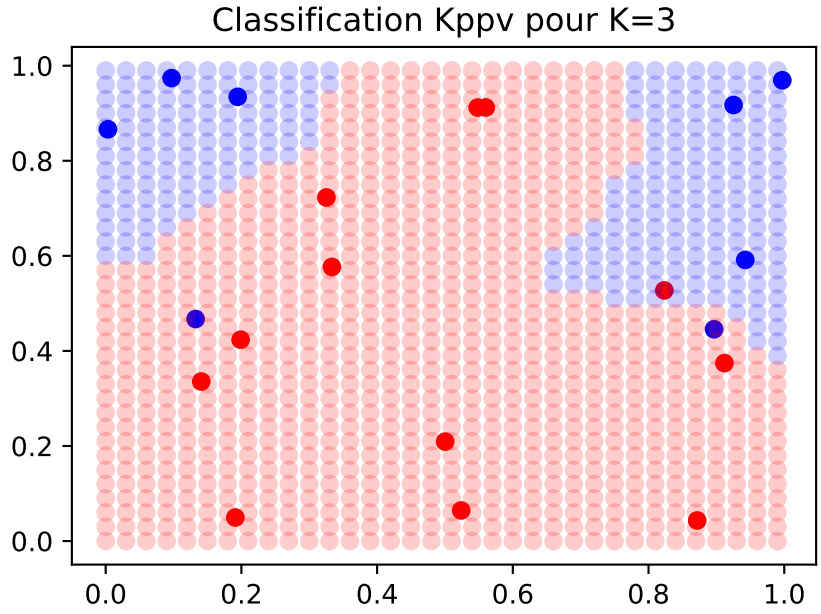
On remarque que cet algorithme ne nécessite pas de phase d'apprentissage à proprement parlé en dehors de la mémorisation de tous les exemples de la base d'apprentissage. L'ensemble des calculs s'effectuent simplement au moment de la prédiction.

Un autre avantage de cette approche est qu'elle fonctionne exactement de la même manière pour des problèmes de classification binaire ou multi-classe.

Concernant la forme de la frontière implémentée par cette méthode, elle n'est pas donnée de manière explicite, car elle dépend directement des données d'apprentissage. Il n'y a donc pas de limite intrinsèque sur la complexité du modèle, la méthode s'adaptant automatiquement aux données.

```
def kppv(Xt,X,y,K):
    yp = np.zeros(len(Xt))
    for i in range(len(Xt)):
        dist = np.linalg.norm(X-Xt[i], axis=1)
        voisinage = []
        for k in range(K):
            idx = np.argmin(dist)
            voisinage.append(idx)
            dist[idx] = np.inf
        votes = np.zeros(np.max(y)+1)
        for k in range(K):
```

```
            votes[y[voisinage[k]]] += 1
        yp[i] = np.argmax(votes)
    return yp

m=20
X = np.random.rand(m, 2)
y = 1 * (np.sin(3*X[:,0]) > X[:,1])
r = np.arange(0,1,0.03)
i=0
Xt = np.zeros((len(r)**2, 2))
for x1 in r:
    for x2 in r:
        Xt[i,:] = [x1,x2]
        i += 1
plt.plot(X[y==0,0],X[y==0,1],"ob")
plt.plot(X[y==1,0],X[y==1,1],"or")
yp = kppv(Xt,X,y,3)
plt.plot(Xt[yp==0,0],Xt[yp==0,1],"ob",alpha=0.2)
plt.plot(Xt[yp==1,0],Xt[yp==1,1],"or",alpha=0.2)
t=plt.title("Classification Kppv pour K=3")
```



13.2 K-plus proches voisins pour la régression

L'algorithme des K plus proches voisins s'adapte aisément à la régression. La seule modification par rapport au cas de la classification est que la valeur prédite n'est pas l'étiquette y_i

majoritaire mais l'étiquette **moyenne** sur le voisinage :

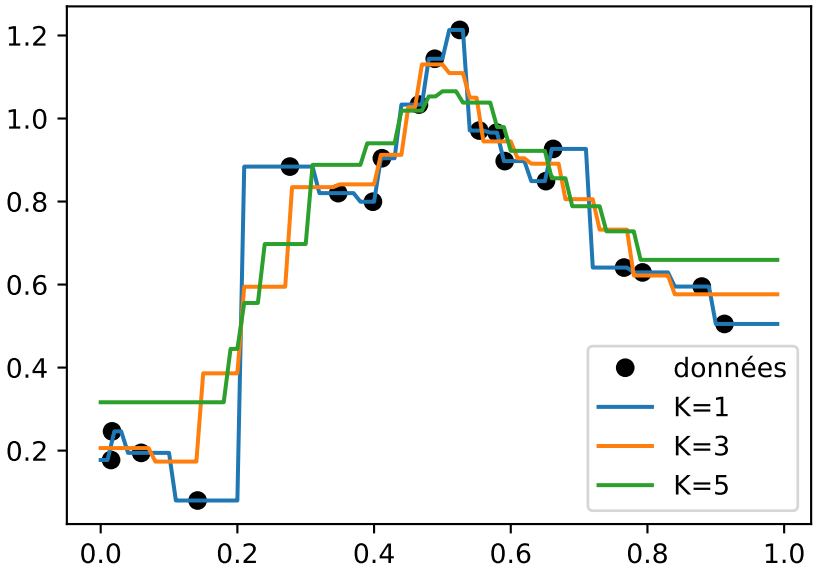
$$f(\boldsymbol{x}) = \frac{1}{K} \sum_{\boldsymbol{x}_i \in \mathcal{V}_K(\boldsymbol{x})} y_i$$

Cette méthode peut être vue comme un moyen d'approcher la fonction de régression.

Les mêmes remarques que pour la classification s'appliquent ici : pas de phase d'apprentissage, adaptation automatique de la forme du modèle à la complexité du problème.

Cependant, la forme du modèle reste particulière : la fonction $f(\boldsymbol{x})$ est une fonction « en escalier ».

```
def kppvreg(Xt,X,y,K):
    yp = np.zeros(len(Xt))
    for i in range(len(Xt)):
        dist = np.abs(X-Xt[i])
        voisinage = []
        for k in range(K):
            idx = np.argmin(dist)
            voisinage.append(idx)
            dist[idx] = np.inf
        yp[i] = np.mean(y[voisinage])
    return yp

m=20
X = np.random.rand(m)
y = np.sin(3*X) + 0.2*np.random.randn(m)
Xt = np.arange(0,1,0.01)
plt.plot(X,y,"ok")
for K in [1,3,5]:
    yp = kppvreg(Xt,X,y,K)
    plt.plot(Xt,yp)
plt.legend(["données"] + ["K=" + str(k) for k in [1,3,5] ])
```



On remarque aussi que l'hyperparamètre K contrôle directement la complexité du modèle : plus K est grand plus la fonction est lissée car les prédictions sont données par des moyennes sur des plus grands voisinages.

13.3 Optimisation des calculs et KD-tree

Le temps de calcul des Kppv peut vite devenir prohibitif lorsque la base d'apprentissage est très grande et que les données $\boldsymbol{x}_i \in \mathbb{R}^d$ sont en grande dimension d. En effet, à chaque prédiction $f(\boldsymbol{x})$, il faut calculer toutes les distances $\|\boldsymbol{x}_i - \boldsymbol{x}\|$ sur l'ensemble de la base avant de pouvoir identifier les plus proches voisins.

Pour optimiser la recherche des voisins, il est possible d'organiser les données en mémoire dans un arbre binaire appelé *KD-tree* :

- Chaque nœud intermédiaire coupe l'espace en 2 (comme un arbre de décision).
- La composante à couper est simplement d'indice $k = profondeur \mod d$.
- Le seuil de coupe est $s = x_{ik}$ pour le point $\boldsymbol{x}_i$ médian selon l'axe k (c'est-à-dire qu'il y a autant de points $\boldsymbol{x}_j$ avec $x_{jk} < x_{ik}$ qu'avec $x_{jk} > x_{ik}$).
- Chaque nœud stocke un point (le point $\boldsymbol{x}_i$ médian choisi pour la coupe) et chaque feuille stocke un petit groupe de points.

La recherche des plus proches voisins de $\boldsymbol{x}$ dans un tel arbre s'effectue ensuite récursivement, en partant de la racine :

- Si le nœud est une feuille : calculer les distances avec tous les points affectés à la feuille et mettre à jour la liste des voisins
- Sinon

- Tester si le point $\boldsymbol{x}$ est à gauche ou à droite de la coupe
- Appeler la recherche sur le nœud fils correspondant
- Tester si l'autre fils peut contenir des voisins, si oui, lancer la recherche (si la distance au voisin le plus loin est supérieure à la distance entre $\boldsymbol{x}$ et le plan de coupe)
- Tester si la recherche peut être stoppée (si la distance au voisin le plus loin est inférieure à toutes les distances entre $\boldsymbol{x}$ et les bords de la région affectée au nœud)

Cette procédure permet en pratique d'éviter le calcul de toutes les m distances et de se concentrer rapidement sur les points potentiellement les plus proches de $\boldsymbol{x}$, et cela avec simplement quelques tests simples sur les composantes.

14 Machines à Vecteurs Supports (SVM)

Les SVM sont une méthode de classification et de régression à la fois versatile, numériquement efficace et relativement simple d'utilisation. Cette méthode est basée sur la maximisation de la marge en classification.

14.1 Hyperplan de marge maximale

En classification linéaire, l'hyperplan de marge maximale, aussi appelé hyperplan de séparation optimale, est l'hyperplan qui classe correctement tous les exemples de la base d'apprentissage avec la marge la plus grande possible.

La **marge** est ici définie comme la plus petite distance entre un point et la frontière de décision du classifieur. Pour un classifier linéaire $f(\boldsymbol{x}) = \text{signe}(\boldsymbol{w}^T\boldsymbol{x} + b)$, cette frontière est un hyperplan $\mathcal{H} = \{\boldsymbol{x} : \boldsymbol{w}^T\boldsymbol{x} + b = 0\}$ et la marge est donnée par

$$\Delta = \min_{\boldsymbol{x}_i \in S} dist(\boldsymbol{x}_i, H) = \min_{\boldsymbol{x}_i \in S} \frac{|\boldsymbol{w}^T\boldsymbol{x}_i + b|}{\|\boldsymbol{w}\|}$$

Trouver l'hyperplan de marge maximale revient à trouver les paramètres $\boldsymbol{w}$ et b du classifieur linéaire correspondant. Cependant, ces paramètres ne sont pas définis de manière unique : le classifieur avec tous ses paramètres multipliés par 2 implémente en fait le même hyperplan de séparation (car $2\boldsymbol{w}^T\boldsymbol{x} + 2b = 0$ est équivalent à $\boldsymbol{w}^T\boldsymbol{x} + b = 0$).

Pour pouvoir résoudre mathématiquement le problème d'apprentissage de l'hyperplan de marge maximale il est donc nécessaire de fixer une contrainte supplémentaire pour retrouver une solution unique.

Cela peut être fait simplement en considérant l'hyperplan canonique pour lequel le point le plus proche satisfait $|\boldsymbol{w}^T\boldsymbol{x}_i + b| = 1$, et donc

$$\min_{\boldsymbol{x}_i \in S} |\boldsymbol{w}^T\boldsymbol{x}_i + b| = 1 \tag{14.1}$$

Dans ce cas, il est évident que multiplier les paramètres n'est plus possible et la solution est bien unique.

Sous la contrainte Équation 14.1, le problème devient donc

- maximiser $\Delta = \frac{1}{\|\boldsymbol{w}\|}$, ce qui revient à minimiser $\|\boldsymbol{w}\|$;
- en classant correctement tous les exemples : $\text{signe}(\boldsymbol{w}^T\boldsymbol{x}_i + b) = y_i$, $i = 1, \ldots, m$;

- et en s'assurant qu'aucun point n'est dans la marge, ce qui revient à s'assurer que $|\boldsymbol{w}^T\boldsymbol{x}_i + b| \geq 1$ (car l'Équation 14.1 s'applique au point le plus proche).

Cela peut être reformulé comme un problème de programmation quadratique convexe pour lequel la solution peut être calculée efficacement :

$$\min_{\boldsymbol{w}\in\mathbb{R}^d,\ b\in\mathbb{R}} \frac{1}{2}\|\boldsymbol{w}\|^2 \quad s.c. \begin{cases} y_1(\boldsymbol{w}^T\boldsymbol{x}_1 + b) \geq 1 \\ \vdots \\ y_m(\boldsymbol{w}^T\boldsymbol{x}_m + b) \geq 1 \end{cases} \tag{14.2}$$

Cet algorithme d'apprentissage est aussi appelé **SVM à marge dure**.

Détails

Pour chaque exemple $(\boldsymbol{x}_i, y_i)$, il faut satisfaire à la fois la contrainte de bon classement et la contrainte de marge. Ces deux contraintes peuvent être résumées en une seule ainsi.
Si $y_i = +1$, il faut s'assurer que

$$\begin{cases} \boldsymbol{w}^T\boldsymbol{x}_i + b \geq 0 \\ |\boldsymbol{w}^T\boldsymbol{x}_i + b| \geq 1 \end{cases} \quad \Leftrightarrow \ \boldsymbol{w}^T\boldsymbol{x}_i + b \geq 1 \ \Leftrightarrow \ y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) \geq 1$$

Si $y_i = -1$, il faut s'assurer que

$$\begin{cases} \boldsymbol{w}^T\boldsymbol{x}_i + b < 0 \\ |\boldsymbol{w}^T\boldsymbol{x}_i + b| \geq 1 \end{cases} \quad \Leftrightarrow \ \boldsymbol{w}^T\boldsymbol{x}_i + b < -11 \ \Leftrightarrow \ y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) \geq 1$$

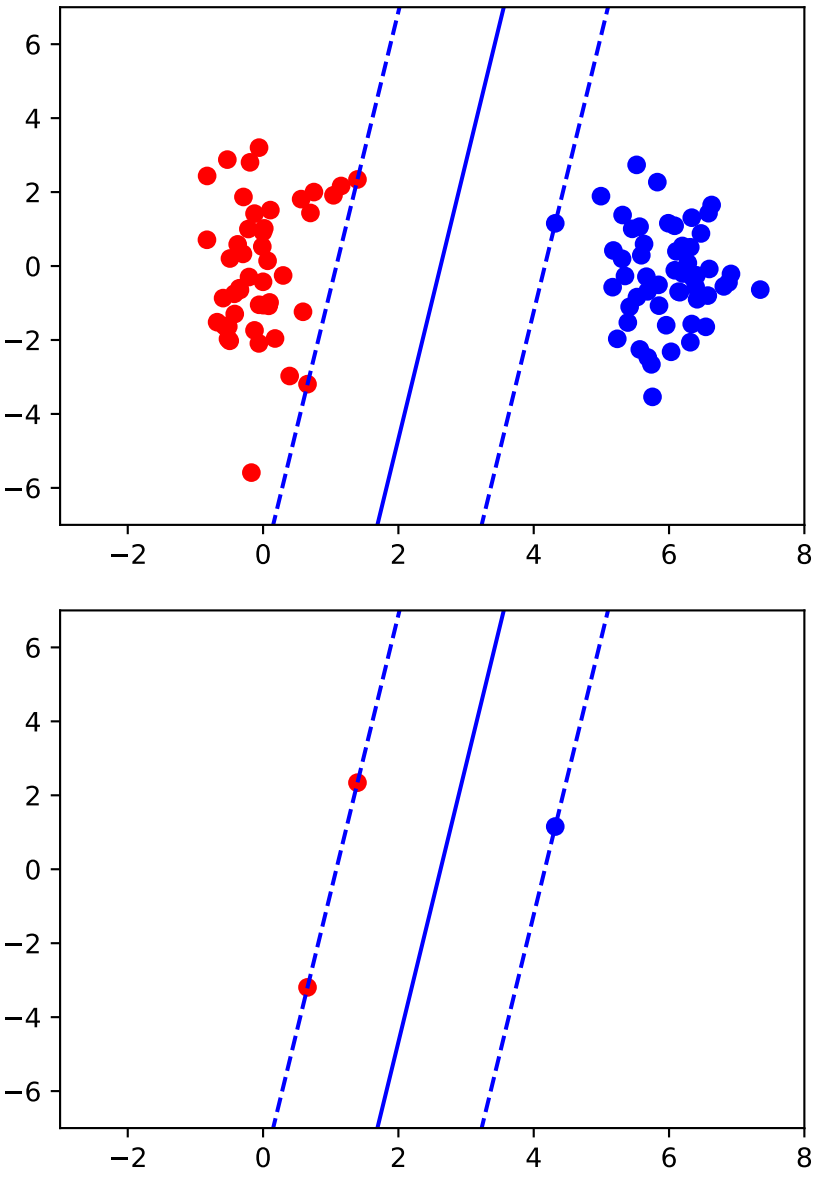
Pour l'objectif minimisé, il est en fait équivalent, car on ne s'intéresse pas ici à la valeur du minimum, mais uniquement à la valeur des variables qui permettent de l'obtenir. Ainsi, le vecteur $\boldsymbol{w}$ qui a la plus petite norme est bien celui qui a la plus petite norme au carré divisée par deux.

L'hyperpan de marge maximale possède quelques propriétés intéressantes :

- Les exemples $\boldsymbol{x}_i$ sur les bords de la marge (avec $|\boldsymbol{w}^T\boldsymbol{x}_i + b| = 1$) sont appelés **vecteurs support**, car ils « supportent » la solution et contiennent toute l'information nécessaire à la résolution du problème.
- Déplacer ou supprimer un exemple $\boldsymbol{x}_i$ en dehors de la marge (qui n'est pas vecteur support) ne change pas la solution.

Exemple

```
from sklearn import svm

m=100
X = 0.5*np.random.randn(m,2)
X[:,1] *= 3
y = np.random.randint(0,2,size=m)
X[y==0,0] += 6

plt.plot(X[y==0,0],X[y==0,1],"ob")
plt.plot(X[y==1,0],X[y==1,1],"or")

# Apprentissage SVM
modele = svm.LinearSVC(C=1e15)
modele.fit(X,y)
w = modele.coef_[0]
b = modele.intercept_
x1 = -2
x2 = 8
plt.plot([x1,x2], [-(x1*w[0] + b)/w[1], -(x2*w[0] + b)/w[1]], "-b")
plt.plot([x1,x2], [-(1+x1*w[0] + b)/w[1], -(1 + x2*w[0] + b)/w[1]], "--b")
plt.plot([x1,x2], [-(-1+x1*w[0] + b)/w[1], -(-1 + x2*w[0] + b)/w[1]], "--b")
plt.axis([-3, 8, -7, 7])

SVs = np.abs(X@w + b) <= 1.001

modele.fit(X[SVs,:],y[SVs])
w = modele.coef_[0]
b = modele.intercept_
plt.figure()
plt.plot(X[(y==0) & SVs,0],X[(y==0) & SVs,1],"ob")
plt.plot(X[(y==1) & SVs,0],X[(y==1) & SVs,1],"or")
plt.plot([x1,x2], [-(x1*w[0] + b)/w[1], -(x2*w[0] + b)/w[1]], "-b")
plt.plot([x1,x2], [-(1+x1*w[0] + b)/w[1], -(1 + x2*w[0] + b)/w[1]], "--b")
plt.plot([x1,x2], [-(-1+x1*w[0] + b)/w[1], -(-1 + x2*w[0] + b)/w[1]], "--b")
plt.axis([-3, 8, -7, 7])
```



L'hyperplan de marge maximale est exactement le même si l'on retire tous les points de la base d'apprentissage sauf ceux exactement sur le bord de la marge.

14.2 Algorithme d'apprentissage SVM linéaire

Dans le cas de données non linéairement séparables, l'hyperplan de marge maximale n'est pas défini car aucun hyperplan ne classe correctement les données (le problème Équation 14.2 n'a pas de solution réalisable).

Pour palier à ce défaut, nous allons relâcher les contraintes, de manière à imposer une **marge**

douce, c'est-à-dire une marge autorisant certaines erreurs. Cela se fait simplement par l'ajout de variables libres $\xi_i \geq 0$ dans les contraintes qui mesurent le niveau de violation de ces contraintes : $y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b) \geq 1$ devient alors

$$y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b) \geq 1 - \xi_i$$

Il est ensuite nécessaire de limiter la quantité d'erreur $\sum_{i=1}^m \xi_i$ autorisée. En fait, cela revient à trouver un compromis entre la maximisation de la marge et la minimisation des erreurs :

$$\min_{\boldsymbol{w} \in \mathbb{R}^d,\ b \in \mathbb{R}, \xi_i} \underbrace{\frac{1}{2}\|\boldsymbol{w}\|^2}_{\text{maximisation de la marge}} + c \underbrace{\sum_{i=1}^m \xi_i}_{\text{minimisation des erreurs}} \tag{14.3}$$

$$s.c. \begin{cases} y_1(\boldsymbol{w}^T \boldsymbol{x}_1 + b) \geq 1 \\ \vdots \\ y_m(\boldsymbol{w}^T \boldsymbol{x}_m + b) \geq 1 \\ \xi_i \geq 0,\ i = 1, \ldots, m \end{cases}$$

où la constante $c > 0$ est un hyperparamètre qui permet de régler ce compromis.

- Pour c grand : les erreurs auront plus de poids dans l'optimisation et le modèle retenu commettra donc moins d'erreurs, mais avec une marge plus faible.
- Pour c petit : l'optimisation se concentrera sur la maximisation de la marge et le modèle retenu commettra plus d'erreurs.

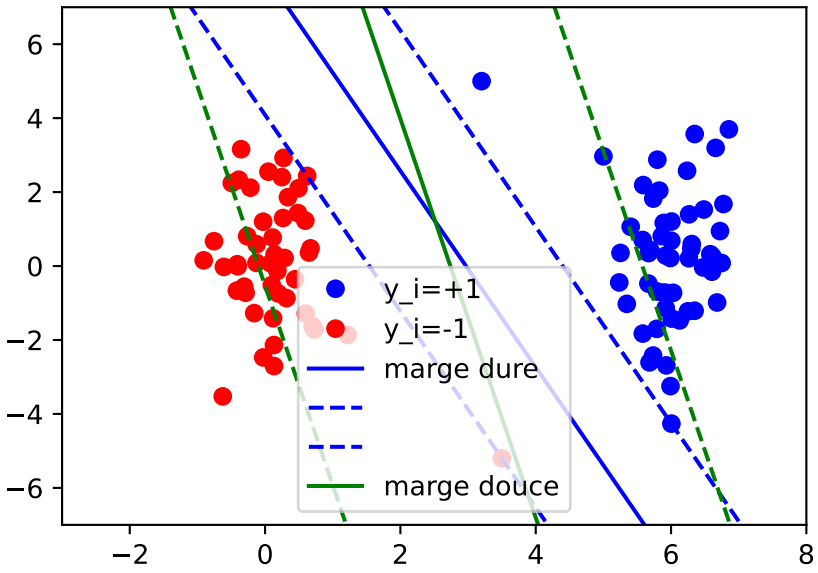
Notons que les erreurs sur les données de la base d'apprentissage ne sont pas catastrophiques en soi. Il est même parfois avantageux de commettre plus d'erreurs lors de l'apprentissage pour éviter le surapprentissage.

L'algorithme SVM ci-dessus possède des propriétés similaires à celles de l'hyperplan de marge maximale :

- Les exemples $\boldsymbol{x}_i$ mal classés, ou dans la marge ou sur les bords de la marge (avec $y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b) \leq 1$) sont appelés **vecteurs support**, car ils « supportent » la solution et contiennent toute l'information nécessaire à la résolution du problème.
- Déplacer ou supprimer un exemple $\boldsymbol{x}_i$ bien classé en dehors de la marge (qui n'est pas vecteur support) ne change pas la solution.
- L'apprentissage se formule toujours comme un problème de programmation quadratique convexe

Exemple

```
from sklearn import svm

m=100
X = 0.5*np.random.randn(m,2)
X[:,1] *= 3
y = np.random.randint(0,2,size=m)
X[y==0,0] += 6
X[0,:] = np.array([3.2,5])
X[1,:] = np.array([3.5,-5.2])
y[0] = 0
y[1] = 1

plt.plot(X[y==0,0],X[y==0,1],"ob")
plt.plot(X[y==1,0],X[y==1,1],"or")

# Apprentissage SVM
modele = svm.LinearSVC(C=1e15)
modele.fit(X,y)
w = modele.coef_[0]
b = modele.intercept_
x1 = -2
x2 = 8
plt.plot([x1,x2], [-(x1*w[0] + b)/w[1], -(x2*w[0] + b)/w[1]], "-b")
plt.plot([x1,x2], [-(1+x1*w[0] + b)/w[1], -(1 + x2*w[0] + b)/w[1]], "--b")
plt.plot([x1,x2], [-(-1+x1*w[0] + b)/w[1], -(-1 + x2*w[0] + b)/w[1]], "--b")

modele = svm.LinearSVC(C=.1)
modele.fit(X,y)
w = modele.coef_[0]
b = modele.intercept_
x1 = -2
x2 = 8
plt.plot([x1,x2], [-(x1*w[0] + b)/w[1], -(x2*w[0] + b)/w[1]], "-g")
plt.plot([x1,x2], [-(1+x1*w[0] + b)/w[1], -(1 + x2*w[0] + b)/w[1]], "--g")
plt.plot([x1,x2], [-(-1+x1*w[0] + b)/w[1], -(-1 + x2*w[0] + b)/w[1]], "--g")

plt.legend(["y_i=+1", "y_i=-1", "marge dure", None, None, "marge douce"])
plt.axis([-3, 8, -7, 7])
```



Le classifieur à marge dure (en bleu) ne fait aucune erreur et n'autorise aucun point dans la marge, mais donne un hyperplan qui ne correspond pas à la distribution générale des données. À l'inverse, le classifieur à marge souple (vert) autorise un certain nombre de points dans le marge, ce qui lui permet d'obtenir une plus grande marge et un hyerplan plus proche de la solution optimale pour ce problème.

14.3 SVM et régularisation

Les SVM peuvent être vu comme une méthode de régularisation et leur apprentissage, Équation 14.3, peut être reformulé comme

$$\min_{\boldsymbol{w}\in\mathbb{R}^d,\ b\in\mathbb{R}} \underbrace{\sum_{i=1}^{m} \max\{0, 1 - y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b)\}}_{\text{terme d'erreur}} \quad + \quad \lambda \quad \underbrace{\|\boldsymbol{w}\|_2^2}_{\text{terme de régularisation}}$$

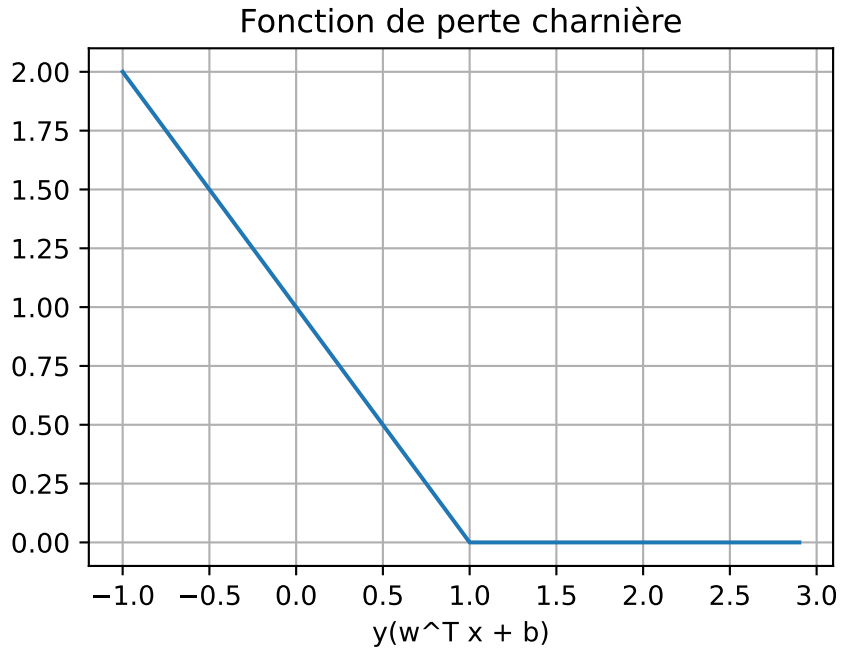
où

- λ est l'hyperparamètre qui joue un rôle similaire à $1/2c$ dans la formulation SVM d'origine ;
- l'erreur est mesurée par la fonction de perte « charnière » (*hinge loss*)

  $$\ell_{hinge}(f, \boldsymbol{x}, y) = \max\{0, 1 - y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b)\}$$

  qui correspond à la valeur prise par ξ_i à l'optimum du problème en Équation 14.3. En effet, puisque ξ_i est minimisé, il prend sa valeur minimale qui est donnée par l'une ou l'autre des contraintes.

```
u = np.arange(-1,3,0.1)
l = (1 - u) * (1-u >= 0)
plt.plot(u,l)
plt.grid()
plt.xlabel("y(w^T x + b)")
t=plt.title("Fonction de perte charnière")
```



14.4 Version duale de l'apprentissage SVM

L'apprentissage des SVM peut aussi être formulé comme le problème de programmation quadratique dual :

$$\max_{\boldsymbol{\alpha}\in\mathbb{R}^m} \; -\frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y_i y_j \langle \boldsymbol{x}_i, \boldsymbol{x}_j\rangle + \sum_{i=1}^{m}\alpha_i$$

$$s.c. \sum_{i=1}^{m}\alpha_i y_i = 0$$

$$0 \leq \alpha_i \leq c$$

Quelques remarques :

- La notation $\langle \boldsymbol{x}_i, \boldsymbol{x}_j\rangle$ du produit scalaire $\boldsymbol{x}_i^T\boldsymbol{x}_j$ est utilisée ici car plus générale.

- Le nombre de variables pour l'optimisation est ici m (le nombre d'exemples), alors qu'il est $d+1+m$ dans la version primale. Ceci peut représenter un gain important lors se l'apprentissage sur des **données en grande dimension** d.
- Les **vecteurs supports** peuvent être retrouvés simplement : ce sont les $\boldsymbol{x}_i$ pour lesquels $\alpha_i \neq 0$.
- Les paramètres $\boldsymbol{w}$ du modèle linéaire sont retrouvés par

$$\boldsymbol{w} = \sum_{i=1}^{m} \alpha_i y_i \boldsymbol{x}_i$$

et le classifieur peut donc s'écrire en fonction des variables duales α_i ainsi :

$$f(\boldsymbol{x}) = \text{signe}\left(\sum_{i=1}^{m} \alpha_i y_i \boldsymbol{x}_i^T \boldsymbol{x} + b\right) \tag{14.4}$$

Cette formulation duale est obtenue par dualité lagrangienne.

Preuve

Commençons par écrire le lagrangien du problème Équation 14.3 :

$$L(\boldsymbol{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\mu}) = \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i=1}^{m}\xi_i - \sum_{i=1}^{m}\alpha_i[y_i(\langle \boldsymbol{w}, \boldsymbol{x}_i\rangle + b) - 1 + \xi_i] - \sum_{i=1}^{m}\mu_i\xi_i$$

où $\alpha_i \geq 0$ et $\mu_i \geq 0$ sont les **variables duales** (multiplicateurs de Lagrange).
Le lagrangien dual est ensuite obtenu au point selle du lagrangien (le minimum de $L(\boldsymbol{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\mu})$ par rapport aux variables primales $\boldsymbol{w}, b, \xi_i$) :

$$L_D(\boldsymbol{\alpha}, \boldsymbol{\mu}) = \inf_{\boldsymbol{w}, b, \boldsymbol{\xi}} L(\boldsymbol{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\mu})$$

Ce point selle est caratérisé par des dérivées partielles nulles :

$$\begin{aligned}
\frac{\partial L(\boldsymbol{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\mu})}{\partial \boldsymbol{w}} &= \boldsymbol{w} - \sum_{i=1}^{m}\alpha_i y_i \boldsymbol{x}_i = \boldsymbol{0} \quad \Rightarrow \quad \boldsymbol{w} = \sum_{i=1}^{m}\alpha_i y_i \boldsymbol{x}_i \\
\frac{\partial L(\boldsymbol{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\mu})}{\partial b} &= -\sum_{i=1}^{m}\alpha_i y_i = 0 \quad \Rightarrow \quad \sum_{i=1}^{m}\alpha_i y_i = 0 \\
\frac{\partial L(\boldsymbol{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\mu})}{\partial \xi_i} &= c - \alpha_i - \mu_i = 0 \quad \Rightarrow \quad \mu_i = c - \alpha_i \quad \Rightarrow \quad 0 \leq \alpha_i \leq c
\end{aligned}$$

Cela conduit à

$$\begin{aligned}
L_D(\boldsymbol{\alpha}) = &\frac{1}{2}\left\|\sum_{i=1}^{m}\alpha_i y_i \boldsymbol{x}_i\right\|^2 + c\sum_{i=1}^{m}\xi_i - \sum_{i=1}^{m}\alpha_i\left[y_i\left(\left\langle\sum_{j=1}^{m}\alpha_j y_j \boldsymbol{x}_j\ ,\ \boldsymbol{x}_i\right\rangle + b\right) - 1 + \xi_i\right] \\
&- \sum_{i=1}^{m}(c - \alpha_i)\xi_i
\end{aligned}$$

et, après élimination des termes en ξ_i,

$$L_D(\boldsymbol{\alpha}) = \frac{1}{2}\left\|\sum_{i=1}^{m}\alpha_i y_i \boldsymbol{x}_i\right\|^2 - \sum_{i=1}^{m}\alpha_i\left[y_i\left(\left\langle\sum_{j=1}^{m}\alpha_j y_j \boldsymbol{x}_j\ ,\ \boldsymbol{x}_i\right\rangle + b\right) - 1\right]$$

Par ailleurs, on a

$$\sum_{i=1}^{m}\alpha_i y_i\left\langle\sum_{j=1}^{m}\alpha_j y_j \boldsymbol{x}_j\ ,\ \boldsymbol{x}_i\right\rangle = \sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y_i y_j \langle \boldsymbol{x}_i, \boldsymbol{x}_j\rangle$$

$$\textit{et}\quad \left\|\sum_{i=1}^{m}\alpha_i y_i \boldsymbol{x}_i\right\|^2 = \left\langle\sum_{i=1}^{m}\alpha_i y_i \boldsymbol{x}_i\ ,\ \sum_{j=1}^{m}\alpha_j y_j \boldsymbol{x}_j\right\rangle = \sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y_i y_j \langle \boldsymbol{x}_i, \boldsymbol{x}_j\rangle$$

Donc

$$\begin{aligned} L_D(\boldsymbol{\alpha}) &= -\frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y_i y_j \langle \boldsymbol{x}_i, \boldsymbol{x}_j\rangle - b\sum_{i=1}^{m}\alpha_i y_i + \sum_{i=1}^{m}\alpha_i \\ &= -\frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y_i y_j \langle \boldsymbol{x}_i, \boldsymbol{x}_j\rangle + \sum_{i=1}^{m}\alpha_i \qquad \left(\text{car } \sum_{i=1}^{m}\alpha_i y_i = 0\right) \end{aligned}$$

et ce lagrangien dual devra être maximisé sous les contraintes, $0 \leq \alpha_i \leq c$ et $\sum_{i=1}^{m}\alpha_i y_i = 0$, obtenues précédemment.

14.4.1 Vecteurs supports et forme parcimonieuse du modèle

Les conditions de relâchement supplémentaires de Karush-Kuhn-Tucker (KKT) stipulent qu'à l'optimum

$$\alpha_i[y_i(\langle \boldsymbol{w}, \boldsymbol{x}_i\rangle + b) - 1 + \xi_i] = 0$$

Cela signifie que, pour tout exemple $(\boldsymbol{x}_i, y_i)$ bien classé en dehors de la marge qui vérifie donc

$$y_i(\langle \boldsymbol{w}, \boldsymbol{x}_i\rangle + b) > 1$$

on a (puisque $\xi_i \geq 0$) : $\alpha_i = 0$. Ainsi, seuls les exemples mal classés ou dans la marge, c'est-à-dire les **vecteurs supports**, sont associés à des $\alpha_i \neq 0$ et donc interviennent dans la forme duale du modèle en Équation 14.4 qui devient :

$$f(\boldsymbol{x}) = \text{signe}\left(\sum_{i\in SV}\alpha_i y_i \boldsymbol{x}_i^T \boldsymbol{x} + b\right)$$

où

$$SV = \{i : y_i(\langle \boldsymbol{w}, \boldsymbol{x}_i\rangle + b) \leq 1\} = \{i : \alpha_i \neq 0\}.$$

14.4.2 Calcul de b

Les conditions de KKT permettent aussi de récupérer la valeur de b. Pour tout $i \in SV$, on a

$$y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b) - 1 + \xi_i = 0$$

et pour les vecteurs supports exactement sur le bord de la marge, $\xi_i = 0$ et donc

$$b = \frac{1 - y_i \boldsymbol{w}^T \boldsymbol{x}_i}{y_i} = y_i - \boldsymbol{w}^T \boldsymbol{x}_i \tag{14.5}$$

car multiplier ou diviser par $y_i \in \{-1, +1\}$ revient au même.

En pratique, les vecteurs supports sur le bord de la marge peuvent être repérés simplement par la valeur de α_i associée. En effet, une autre condition KKT associées aux contraintes $\xi_i \geq 0$ implique qu'à l'optimum

$$\mu_i \xi_i = 0 \quad \text{et donc } \mu_i \neq 0 \Rightarrow \xi_i = 0$$

et puisque $\mu_i = c - \alpha_i$,

$$\alpha_i < c \quad \Rightarrow \quad \xi_i = 0$$

et le point est sur le bord de la marge.

Au final, b est calculé avec la formule Équation 14.5 sur un point tel que $0 < \alpha_i < c$, ou plutôt avec une moyenne sur tous ces points pour limiter l'influence des erreurs numériques.

14.5 SVM non linéaires

Les SVM sont un très bon exemple de méthode à noyaux pour lesquelles l'extension au cas non linéaire est presque immédiate.

En effet, la forme duale de l'apprentissage SVM ne fait intervenir les données $\boldsymbol{x}_i$ qu'au travers de produits scalaires entre elles. Projeter ces données avec une fonction non linéaire $\phi(\boldsymbol{x})$ avant l'apprentissage revient donc à calculer des produits

$$\langle \phi(\boldsymbol{x}_i), \phi(\boldsymbol{x}_j) \rangle = K(\boldsymbol{x}_i, \boldsymbol{x}_j)$$

ce qui peut se faire efficacement grâce à un noyau $K(\cdot, \cdot)$. C'est l'astuce du noyau.

Le classifieur en Équation 14.4 s'exprime alors comme

$$f(\boldsymbol{x}) = \text{signe}\left(\sum_{i \in SV} \alpha_i y_i K(\boldsymbol{x}_i, \boldsymbol{x}) + b \right)$$

et l'apprentissage revient à résoudre le problème de programmation quadratique

$$\begin{aligned} \max_{\boldsymbol{\alpha}\in\mathbb{R}^m} \quad & -\frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y_i y_j K(\boldsymbol{x}_i,\boldsymbol{x}_j)+\sum_{i=1}^{m}\alpha_i \\ s.c. \quad & \sum_{i=1}^{m}\alpha_i y_i = 0 \\ & 0 \leq \alpha_i \leq c \end{aligned}$$

où le nombre de variables est identique au cas linéaire.

En considérant $f(\boldsymbol{x}) = \text{signe}(g(\boldsymbol{x}))$, pour une fonction non linéaire quelconque g, il est aussi possible de formuler les SVM non linéaires comme un algorithme d'apprentissage de la fonction $g \in \mathcal{H}$ dans un espace de Hilbert à noyau reproduisant.

15 Régression

En régression, l'étiquette $y \in \mathcal{Y}$ peut prendre une infinité de valeurs, en général à l'intérieur d'un intervalle de $\mathbb{R}$.

L'erreur d'un modèle de régression sur un exemple est (en général) mesurée par la *fonction de perte quadratique* :

$$\ell(f, \boldsymbol{x}, y) = (y - f(\boldsymbol{x}))^2$$

qui est bien positive, et nulle lorsque $f(\boldsymbol{x}) = y$. Le risque de régression est donc l'**erreur quadratique moyenne**

$$R(f) = \mathbb{E}(Y - f(\boldsymbol{X}))^2$$

15.1 Modèle optimal : la fonction de régression

La fonction de régression retourne, pour tout $\boldsymbol{x}$, l'espérance conditionnelle de Y sachant $\boldsymbol{X}$ au point $\boldsymbol{X} = \boldsymbol{x}$:

$$f_{reg}(\boldsymbol{x}) = \mathbb{E}[Y|\boldsymbol{X} = \boldsymbol{x}]$$

Cette fonction est le meilleur modèle de régression possible dans le sens où elle minimise le risque :

$$R(f_{reg}) = \min_{f:\mathcal{X}\to\mathcal{Y}} R(f)$$

Elle reste cependant théorique, car pour pouvoir la calculer, il faut soit connaître la loi de probabilité de $(\boldsymbol{X}, Y)$ supposée inconnue, soit avoir accès à une infinité de tirages de Y pour chaque $\boldsymbol{x}$.

Preuve de l'optimalité de la fonction de régression

Nous allons chercher le modèle optimal

$$f^* = \arg\min_{f:\mathcal{X}\to\mathcal{Y}} \mathbb{E}(Y - f(\boldsymbol{x}))^2$$

Chercher une fonction revient, de manière équivalente, à trouver sa valeur $f^*(\boldsymbol{x})$ pour tout $\boldsymbol{x} \in \mathcal{X}$. De plus, le modèle optimal n'est pas contraint à une forme particulière et toutes ses valeurs $f^*(\boldsymbol{x})$ peuvent donc être déterminées indépendamment les unes des autres.
Par ailleurs, le théorème de l'espérance totale permet de réécrire le risque comme la moyenne par rapport à $\boldsymbol{X}$ de la moyenne par rapport à Y à $\boldsymbol{X}$ fixé :

$$R(f) = \mathbb{E}(Y - f(\boldsymbol{X}))^2 = \mathbb{E}\mathbb{E}[(Y - f(X))^2|\boldsymbol{X}] = \int_{\mathcal{X}} \mathbb{E}[(Y - f(X))^2|\boldsymbol{X} = \boldsymbol{x}]p(\boldsymbol{x})d\boldsymbol{x}$$

Pour minimiser cette somme de termes indépendants, il suffit donc de trouver la valeur $y = f^*(\boldsymbol{x})$ qui minimise $\mathbb{E}[(Y - f(\boldsymbol{X}))^2 | \boldsymbol{X} = \boldsymbol{x}]$ pour chaque $\boldsymbol{x}$:

$$\begin{aligned}
\forall \boldsymbol{x} \in \mathcal{X},\ f^*(\boldsymbol{x}) &= \arg\min_{y \in \mathbb{R}} \mathbb{E}[(Y - y)^2 | \boldsymbol{X} = \boldsymbol{x}] \\
&= \arg\min_{y \in \mathbb{R}} \mathbb{E}\left[Y^2 - 2yY + y^2 \mid \boldsymbol{X} = \boldsymbol{x}\right] \\
&= \arg\min_{y \in \mathbb{R}} \mathbb{E}[Y^2 | \boldsymbol{X} = \boldsymbol{x}] - 2y\mathbb{E}[Y | \boldsymbol{X} = \boldsymbol{x}] + y^2 \\
&= \arg\min_{y \in \mathbb{R}} \underbrace{-2y\mathbb{E}[Y | \boldsymbol{X} = \boldsymbol{x}] + y^2}_{J(y)}
\end{aligned}$$

où les termes constants par rapport à la variable d'optimisation y peuvent être négligés. La fonction $J(y)$ est quadratique et convexe. Elle peut donc être aisément optimisée et son minimum se trouve au point où la dérivée (ou gradient) est nulle :

$$\frac{\mathrm{d}J(y)}{\mathrm{d}dy} = -2\mathbb{E}[Y | \boldsymbol{X} = \boldsymbol{x}] + 2y = 0 \quad \Rightarrow \quad y = \mathbb{E}[Y | \boldsymbol{X} = \boldsymbol{x}]$$

Ainsi la fonction de régression f_{reg} est bien le modèle optimal f^*.

15.2 Régression linéaire

En régression linéaire avec $\mathcal{X} \subset \mathbb{R}^d$, le modèle prend la forme

$$f(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x}$$

avec un vecteur de paramètres $\boldsymbol{w} \in \mathbb{R}^d$.

Il est aussi possible de considérer un modèle affine $f(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x} + b$, mais celui-ci est équivalent à un modèle linéaire opérant sur des données étendues $\tilde{\boldsymbol{x}} = \begin{bmatrix} \boldsymbol{x} \\ 1 \end{bmatrix}$ en dimension $d + 1$ avec la dernière composante de $\tilde{\boldsymbol{w}}$ égale à b.

L'apprentissage d'un modèle linéaire peut se faire par la méthode des moindres carrés.

16 Méthode des moindres carrés

En régression linéaire, la méthode des moindres carrés permet d'estimer les paramètres $\boldsymbol{w} \in \mathbb{R}^d$ du modèle

$$f(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x}$$

directement par minimisation du risque empirique[1] :

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w} \in \mathbb{R}^d} \sum_{i=1}^{m} (y_i - \boldsymbol{w}^T \boldsymbol{x}_i)^2.$$

En effet, la solution de ce problème d'optimisation est donnée explicitement par

$$\hat{\boldsymbol{w}} = (\boldsymbol{X}^T \boldsymbol{X})^{-1} \boldsymbol{X}^T \boldsymbol{y}$$

où les m exemples d'apprentissages $(\boldsymbol{x}_i, y_i)$ sont concaténés dans

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x}_1^T \\ \vdots \\ \boldsymbol{x}_m^T \end{bmatrix} \in \mathbb{R}^{m \times d}, \qquad \boldsymbol{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} \in \mathbb{R}^m.$$

16.1 Démonstration de la formule des moindres carrés

Nous pouvons commencer par formuler le critère à minimiser en fonction des matrices $\boldsymbol{X}$, $\boldsymbol{y}$:

$$J(\boldsymbol{w}) = \sum_{i=1}^{m} (y_i - \boldsymbol{w}^T \boldsymbol{x}_i)^2 = \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\|^2$$

[1] Minimiser la somme des erreurs est équivalent à minimiser la moyenne

Preuve

Soit le vecteur des erreurs $\boldsymbol{e} = \begin{bmatrix} e_1 \\ \vdots \\ e_m \end{bmatrix}$ avec $e_i = y_i - \boldsymbol{w}^T \boldsymbol{x}_i$. Par la symétrie du produit scalaire, nous avons aussi $e_i = y_i - \boldsymbol{x}_i^T \boldsymbol{w}$, et donc

$$\boldsymbol{e} = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} - \begin{bmatrix} \boldsymbol{x}_1^T \boldsymbol{w} \\ \vdots \\ \boldsymbol{w}^T \boldsymbol{x}_m \end{bmatrix} = \boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}.$$

D'un autre côté, par définition de la norme euclidienne,

$$J(\boldsymbol{w}) = \sum_{i=1}^{m} e_i^2 = \|\boldsymbol{e}\|^2.$$

ce qui conclut la preuve.

La fonction $J(\boldsymbol{w})$ est quadratique et convexe, ce qui signifie que son minimum se situe au(x) point(s) où la dérivée (le gradient si $d > 1$) est nulle. Il suffit donc de résoudre

$$\frac{\mathrm{d}J(\hat{\boldsymbol{w}})}{\mathrm{d}\boldsymbol{w}} = \boldsymbol{0}$$

où, d'après le règle de dérivation en chaîne, avec $\boldsymbol{e} = \boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}$,

$$\frac{\mathrm{d}J(\boldsymbol{w})}{\mathrm{d}\boldsymbol{w}} = \left(\frac{\mathrm{d}\boldsymbol{e}}{\mathrm{d}\boldsymbol{w}}\right)^T \frac{\mathrm{d}J(\boldsymbol{w})}{\mathrm{d}\boldsymbol{e}}$$

La règle de dérivation des fonctions linéaires donne

$$\frac{\mathrm{d}\boldsymbol{e}}{\mathrm{d}\boldsymbol{w}} = -\boldsymbol{X}$$

et la règle de dérivation des fonctions quadratiques

$$\frac{\mathrm{d}J(\boldsymbol{w})}{\mathrm{d}\boldsymbol{e}} = 2\boldsymbol{e}$$

Cela conduit à

$$\frac{\mathrm{d}J(\boldsymbol{w})}{\mathrm{d}\boldsymbol{w}} = -2\boldsymbol{X}^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w})$$

et donc une dérivée nulle lorsque

$$\boldsymbol{X}^T\boldsymbol{X}\hat{\boldsymbol{w}} = \boldsymbol{X}^T\boldsymbol{y}.$$

Dans le cas où $\boldsymbol{X}^T\boldsymbol{X}$ est inversible, la formule souhaitée est obtenue par multiplication à gauche par son inverse. Dans le cas contraire, la solution s'obtient par résolution du système d'équations linéaires ci-dessus par d'autres techniques (comme la factorisation QR, ou la pseudo-inverse par exemple). Mais en pratique, il est toujours préférable et plus efficace de résoudre le système linéaire ci-dessus plutôt que de calculer explicitement l'inverse de la matrice.

17 Moindres carrés régularisés (régression *ridge*)

En régression linéaire, la méthode des moindres carrés permet d'estimer les paramètres $\boldsymbol{w} \in \mathbb{R}^d$ du modèle

$$f(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x} \tag{17.1}$$

directement par minimisation du risque empirique, et donc de trouver le modèle linéaire qui « colle » le plus possible aux données. Cependant, lorsque la dimension d des données est trop grande par rapport au nombre m d'exemples, cela peut conduire au surapprentissage.

Pour éviter cela, la régularisation consiste à ajouter un terme pénalisant la complexité du modèle dans la minimisation. La **régression ridge** propose de mesurer cette complexité par la norme euclidienne du vecteur de paramètres. Cela conduit à formuler l'apprentissage ainsi :

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w} \in \mathbb{R}^d} \sum_{i=1}^{m} (y_i - \boldsymbol{w}^T \boldsymbol{x}_i)^2 + \lambda \|\boldsymbol{w}\|^2$$

où l'hyperparamètre $\lambda > 0$ règle le compromis entre l'attache aux données et la régularisation.

Comme pour les moindres carrés classiques, ce problème d'optimisation possède une solution explicite :

$$\hat{\boldsymbol{w}} = (\boldsymbol{X}^T \boldsymbol{X} + \lambda \boldsymbol{I})^{-1} \boldsymbol{X}^T \boldsymbol{y}$$

où $\boldsymbol{I}$ est la matrice identité de taille $d \times d$ et les m exemples d'apprentissages $(\boldsymbol{x}_i, y_i)$ sont concaténés dans

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x}_1^T \\ \vdots \\ \boldsymbol{x}_m^T \end{bmatrix} \in \mathbb{R}^{m \times d}, \qquad \boldsymbol{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} \in \mathbb{R}^m.$$

La seule différence par rapport à la version non régularisée est donc l'ajout du coefficient de régularisation λ sur la diagonale de $\boldsymbol{X}^T \boldsymbol{X}$.

Preuve

En suivant le cas non régularisé, nous pouvons commencer par formuler le critère à minimiser en fonction des matrices $\boldsymbol{X}$, $\boldsymbol{y}$:

$$J(\boldsymbol{w}) = \sum_{i=1}^{m} (y_i - \boldsymbol{w}^T \boldsymbol{x}_i)^2 + \lambda \|\boldsymbol{w}\|^2 = \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\|^2 + \lambda \|\boldsymbol{w}\|^2$$

Il suffit ensuite de calculer le gradient de cette fonction et de déterminer le point $\hat{\boldsymbol{w}}$ où

il s'annule :

$$\frac{\mathrm{d}J(\boldsymbol{w})}{\mathrm{d}\boldsymbol{w}} = -2\boldsymbol{X}^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}) + 2\lambda\boldsymbol{w} = -2\boldsymbol{X}^T\boldsymbol{y} + 2(\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I})\boldsymbol{w}$$

Et donc

$$\frac{\mathrm{d}J(\hat{\boldsymbol{w}})}{\mathrm{d}\boldsymbol{w}} = \boldsymbol{0} \quad \Rightarrow \quad (\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I})\hat{\boldsymbol{w}} = \boldsymbol{X}^T\boldsymbol{y}$$

ce qui conduit à la solution par multiplication à gauche par la matrice inverse de $(\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I})$.

17.1 Lien entre la complexité du modèle et la norme des paramètres

La complexité du modèle peut se mesurer en termes de variations de sa sortie par rapport aux variations de son entrée, autrement dit par l'amplitude de sa dérivée, ou plus précisément pour $d > 1$, par la norme de son gradient :

$$\left\| \frac{\mathrm{d}f(\boldsymbol{x})}{\mathrm{d}\boldsymbol{x}} \right\|$$

Pour le modèle linéaire en Équation 17.1, le gradient correspond simplement au vecteur des paramètres :

$$\frac{\mathrm{d}f(\boldsymbol{x})}{\mathrm{d}\boldsymbol{x}} = \boldsymbol{w}$$

puisque chaque composante w_k correspond au coefficient directeur de l'hyperplan correspondant au graphique de $f(\boldsymbol{x})$ selon l'axe x_k.

18 LASSO

En régression linéaire, la méthode des moindres carrés permet d'estimer les paramètres $\boldsymbol{w} \in \mathbb{R}^d$ du modèle

$$f(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x} \tag{18.1}$$

directement par minimisation du risque empirique, et donc de trouver le modèle linéaire qui « colle » le plus possible aux données. Cependant, lorsque la dimension d des données est trop grande par rapport au nombre m d'exemples, cela peut conduire au surapprentissage.

Pour éviter cela, la régularisation consiste à ajouter un terme pénalisant la complexité du modèle dans la minimisation. La régression ridge propose de mesurer cette complexité par la norme euclidienne du vecteur des paramètres.

La méthode **LASSO** propose quant-à elle de mesurer cette complexité par la norme ℓ_1 des paramètres, c'est-à-dire la somme des valeurs absolues des paramètres w_k. Cela conduit à formuler l'apprentissage ainsi :

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w} \in \mathbb{R}^d} \sum_{i=1}^{m} (y_i - \boldsymbol{w}^T \boldsymbol{x}_i)^2 + \lambda \|\boldsymbol{w}\|_1$$

où l'hyperparamètre $\lambda > 0$ règle le compromis entre l'attache aux données et la régularisation.

Contrairement aux moindres carrés classiques et la régression ridge, ce problème d'optimisation ne possède une solution explicite que dans certains cas particuliers. Dans le cas général, le problème se reformule comme un problème de programmation quadratique :

$$\begin{aligned} \hat{\boldsymbol{w}} = \arg \min_{\boldsymbol{w} \in \mathbb{R}^d, a_k \geq 0} & \sum_{i=1}^{m} (y_i - \boldsymbol{w}^T \boldsymbol{x}_i)^2 + \lambda \sum_{k=1}^{d} t_k \\ s.c. \quad & -t_k \leq w_k \leq t_k, \quad k = 1, \ldots, d \end{aligned}$$

où les variables auxiliaires t_k représentent les valeurs absolues des w_k au travers de contraintes linéaires.

18.1 Modèles parcimonieux

La régularisation par la norme ℓ_1 du LASSO conduit en général à des modèles plus parcimonieux, c'est-à-dire avec beaucoup de paramètres nuls. Cette parcimonie peut se mesurer par la pseudo-norme ℓ_0 :

$$\|\boldsymbol{w}\|_0 = |\{k : w_k \neq 0\}|$$

avec typiquement $\|\boldsymbol{w}\|_0 < d/3$ pour un modèle parcimonieux.

L'intérêt d'un modèle parcimonieux est multiple :

- la sortie du modèle peut être calculée plus efficacement car elle n'implique qu'un petit nombre de termes :

$$f(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} = \sum_{k=1}^{d} w_k x_k = \sum_{k:w_k\neq 0} w_k x_k$$

- le modèle effectue une sélection de variables : les composantes x_k telles que $w_k = 0$ n'influencent pas la sortie $f(\boldsymbol{x})$;
- ces variables exclues du modèle, typiquement issues de mesures physiques sur les objets d'intérêt, n'ont pas besoin d'être mesurées : les capteurs correspondants peuvent donc être économisés.

Obtenir la parcimonie par minimisation d'une norme ℓ_1 : intuition

Pour comprendre pourquoi minimiser la somme des valeurs absolues conduit plus souvent à des solutions parcimonieuses comparativement à la minimisation d'une norme euclidienne (comme en régression ridge), nous prendrons un exemple plus simple à étudier.
Soit un problème de résolution d'équation linéaire sous-déterminée :

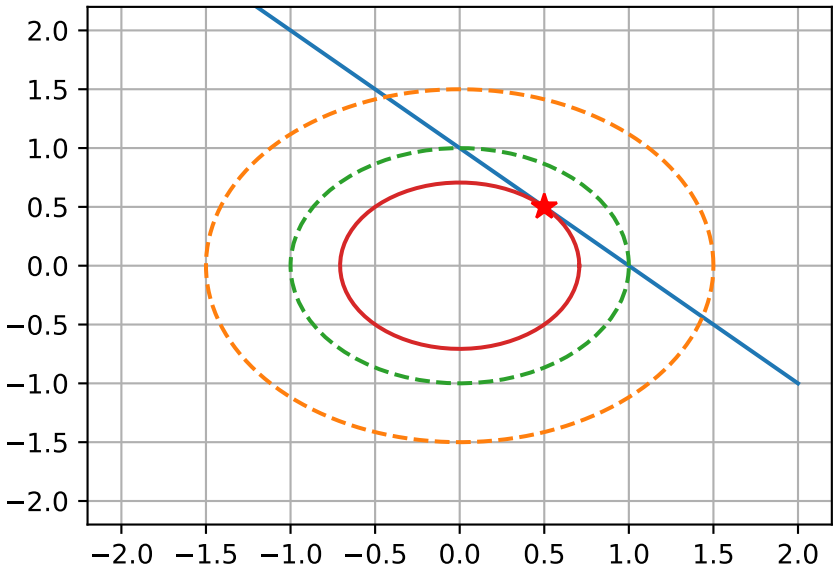
$$\boldsymbol{a}^T\boldsymbol{w} = b$$

avec $\boldsymbol{a}, \boldsymbol{w} \in \mathbb{R}^2$. Ici, nous avons deux variables w_1 et w_2 (les composantes de $\boldsymbol{w}$) et une seule équation. Il existe donc une infinité de solutions. Pour retrouver l'unicité, nous allons régulariser en cherchant la solution de norme minimale.
Pour la norme euclidienne, cela donne

$$\min_{\boldsymbol{w}\in\mathbb{R}^2} \|\boldsymbol{w}\|, \quad s.c.\ \boldsymbol{a}^T\boldsymbol{w} = b$$

Dans l'espace des variables, la contrainte linéaire correspondant à une droite sur laquelle toutes les solutions se trouvent. Dans cet espace, nous pouvons dessiner les courbes de niveau de $\|\boldsymbol{w}\|$, c'est-à-dire les lieux des points où cette norme est constante. Les solutions du problème de minimisation se trouvent donc à l'intersection de ces courbes avec la droite, pour la courbe correspondant à la plus petite norme. Les courbes de niveau d'une norme euclidienne sont des cercles où le rayon égale $\|\boldsymbol{w}\|$, et le cercle de plus petit rayon qui intersecte la droite donne la solution. Celui-ci est le cercle tangent à la droite.

```
a = np.array([1, 1])
b = 1
x = np.array([-2,2])
plt.plot(x, (-a[0]*x + b) / a[1] )
t = np.linspace( 0 , 2 * np.pi , 150 )
plt.plot(1.5*np.cos(t), 1.5*np.sin(t), "--")
plt.plot(np.cos(t), np.sin(t), "--")
plt.plot(np.sqrt(0.5)*np.cos(t), np.sqrt(0.5)*np.sin(t), "-")
plt.plot(0.5,0.5, "*r", markersize=10)
plt.grid()
plt.axis([-2.2, 2.2, -2.2, 2.2])
```

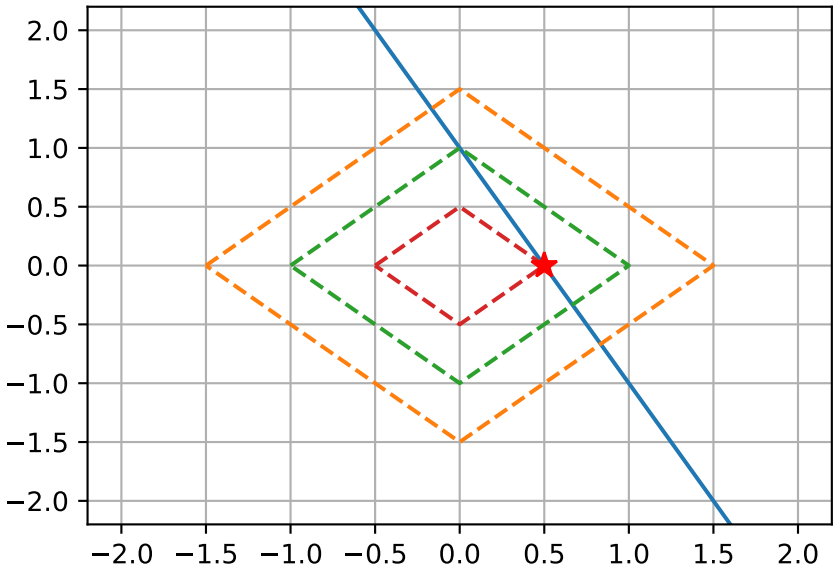


Sur cette illustration, il est clair que la solution n'est pas parcimonieuse : $w_1 \neq 0$ et $w_2 \neq 0$. De plus, la plupart des droites possibles (pour d'autres valeurs de $\boldsymbol{a}$ et b) donnent le même type de solutions. Seules les 4 droites parallèles aux axes donneraient des solutions avec un composante à 0.
Considérons maintenant la minimisation de la norme ℓ_1 :

$$\min_{\boldsymbol{w}\in\mathbb{R}^2} \|\boldsymbol{w}\|_1, \quad s.c.\ \boldsymbol{a}^T\boldsymbol{w} = b$$

Ce problème peut être illustré de la même manière mais avec des « boules ℓ_1 » pour les courbes de niveau, c'est-à-dire des carré dont les pointes se situent sur les axes.

```
a = np.array([2, 1])
b = 1
x = np.array([-2,2])
plt.plot(x, (-a[0]*x + b) / a[1] )
plt.plot([-1.5, 0, 1.5, 0, -1.5], [0 , 1.5, 0 , -1.5, 0], "--")
plt.plot([-1, 0, 1, 0, -1], [0 , 1, 0 , -1, 0], "--")
plt.plot([-0.5, 0, 0.5, 0, -0.5], [0 , 0.5, 0 , -0.5, 0], "--")
plt.plot(0.5,0, "*r", markersize=10)
plt.grid()
plt.axis([-2.2, 2.2, -2.2, 2.2])
```



Ici, la solution au point tangent est bien parcimonieuse : $w_2 = 0$. De plus, la plupart des autres droites auraient donné le même type de solution, sauf pour les cas très particuliers de droites à 45 degrés parallèles à une face du carré.

19 Méthodes à noyaux

Les méthodes à noyaux offrent une extension simple de méthodes linéaires au cas non linéaire, pour construire des classifieurs plus complexes que des hyperplans ou des modèles de régression plus flexibles.

Elles reposent sur la projection des données dans un espace de grande dimension et sur l'astuce du noyau qui permet de calculer en très grande dimension efficacement pour créer des modèles non linéaires avec des algorithmes d'apprentissage linéaires.

Les SVM, la régression ridge à noyau et la PCA non linéaire sont des bons exemples de telles méthodes.

19.1 Projection non linéaire

Pour créer un modèle non linéaire, nous pouvons projeter les données dans un nouvel espace de représentation où le problème devient linéaire.

Par exemple, le modèle non linéaire

$$f(x) = w_1 \sin(x) + w_2 x^2 - w_3 x^5$$

est équivalent au modèle linéaire

$$f(x) = \boldsymbol{w}^T \phi(x), \qquad \boldsymbol{w} = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}, \ \phi(x) = \begin{bmatrix} \sin(x) \\ x^2 \\ -x^5 \end{bmatrix}$$

après projection des données x en $\phi(x)$. Ainsi, apprendre le modèle non linéaire f revient à estimer les paramètres $\boldsymbol{w}$ du modèle linéaire.

La procédure globale peut donc s'écrire :

1. Choisir une projection non linéaire $\phi : \mathcal{X} \to \mathcal{X}_\phi$.
2. Projeter toutes les données : $\boldsymbol{\phi}_i = \phi(\boldsymbol{x}_i)$, $i = 1, \ldots, m$.
3. Appliquer une méthode linéaire aux exemples $(\boldsymbol{\phi}_i, y_i)$ au lieu de $(\boldsymbol{x}_i, y_i)$.

En général, la dimension de l'espace de représentation $\mathcal{X}_\phi$ augmente par rapport à celle de $\mathcal{X}$. En augmentant suffisamment cette dimension, il est possible d'approcher n'importe quelle non-linéarité : *la complexité du modèle non linéaire dépend directement de la dimension de l'espace de représentation.*

Cependant, la régularisation (ou le contrôle de la complexité) devient cruciale pour éviter le surapprentissage avec des modèles non linéaires très flexibles.

Exemple de la régression polynomiale

La régression polynomiale cherche un modèle de degré D de la forme

$$f(x) = \sum_{k=0}^{D} w_k x^k$$

Bien que non linéaire, ce modèle peut s'exprimer linéairement par rapport à la projection de x dans l'espace des monomes :

$$f(x) = \boldsymbol{w}^T \phi(x), \quad avec\ \phi(x) = \begin{bmatrix} 1 \\ x \\ x^2 \\ \vdots \\ x^D \end{bmatrix}$$

Ainsi, apprendre les coefficients du modèle polynomial revient simplement à appliquer la méthode des moindres carrés aux données prétraitées $(\phi(x_i), y_i)$. Avec la matrice

$$\boldsymbol{\Phi} = \begin{bmatrix} \phi(x_1)^T \\ \vdots \\ \phi(x_m)^T \end{bmatrix} = \begin{bmatrix} 1 & x_1 & x_1^2 & \dots & x_1^D \\ \vdots & & & & \\ 1 & x_m & x_m^2 & \dots & x_m^D \end{bmatrix}$$

cela signifie calculer directement

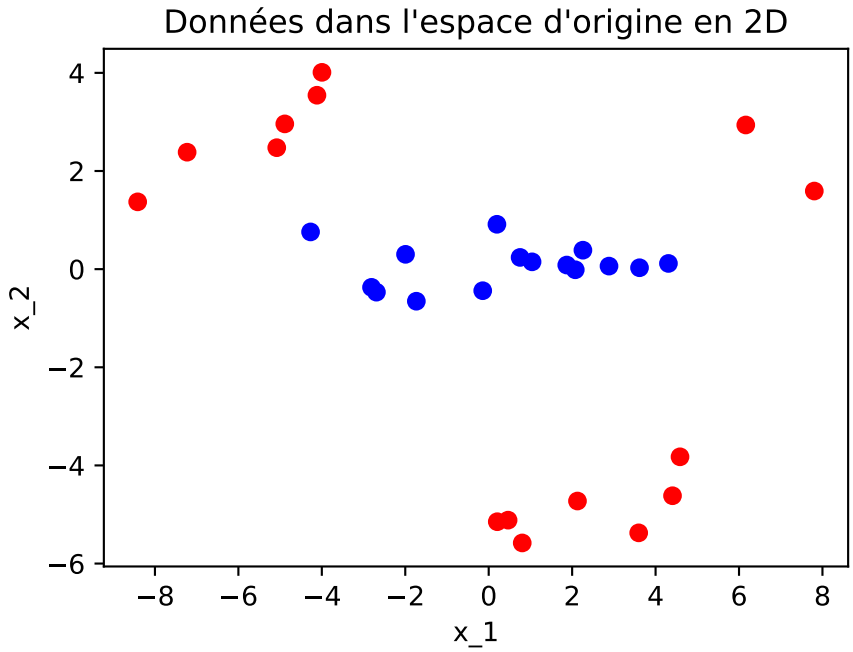
$$\boldsymbol{w} = (\boldsymbol{\Phi}^T \boldsymbol{\Phi})^{-1} \boldsymbol{\Phi}^T \boldsymbol{y}$$

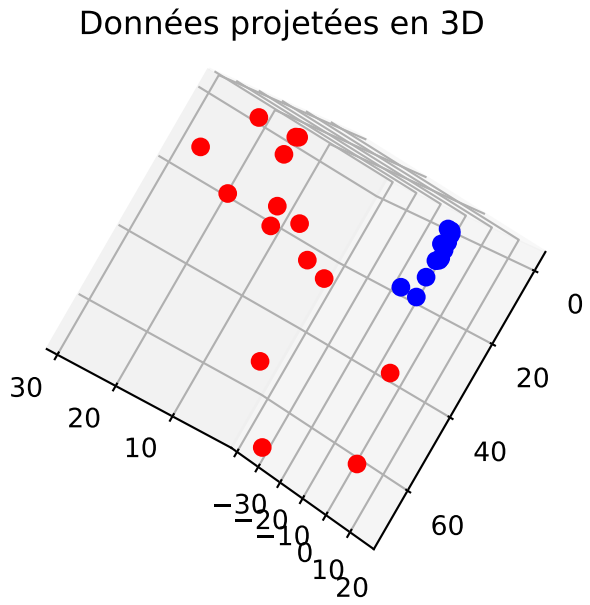
Exemple de classification non linéaire en 2D

Les données suivantes sont non linéairement séparables dans le plan mais deviennent linéairement séparables une fois projetées en 3D avec

$$\phi(\boldsymbol{x}) = \begin{bmatrix} x_1^2 \\ \sqrt{2} x_1 x_2 \\ x_2^2 \end{bmatrix}$$

```
m1 = 15
x1 = 10*np.random.rand(m1) - 5
x2 = np.random.randn(m1) * (5-x1) / 8;
X1 = np.vstack([x1,x2]).T
x1 = 18 * np.random.rand(m1) - 9
x2 = 0.5*np.random.randn(m1) - 0.2
r = np.random.rand(m1) > 0.5
x2[r] += 5* np.cos(0.15*x1[r])
x2[~r] -=  5* np.cos(0.15*x1[~r])
X2 = np.vstack([x1,x2]).T
X = np.vstack([X1,X2])
y = np.zeros(2*m1)
y[m1:] = 1

PHI = np.vstack([ X[:,0]**2, np.sqrt(2) * X[:,0] * X[:,1], X[:,1]**2 ]).T

plt.plot(X[y==0,0], X[y==0,1], "ob")
plt.plot(X[y==1,0], X[y==1,1], "or")
plt.xlabel("x_1")
plt.ylabel("x_2")
plt.title("Données dans l'espace d'origine en 2D")

ax = plt.figure().add_subplot(projection='3d')
ax.plot(PHI[y==0,0],PHI[y==0,1],PHI[y==0,2], "ob")
ax.plot(PHI[y==1,0],PHI[y==1,1],PHI[y==1,2], "or")
ax.view_init(30, 90, 60)
t=plt.title("Données projetées en 3D")
```





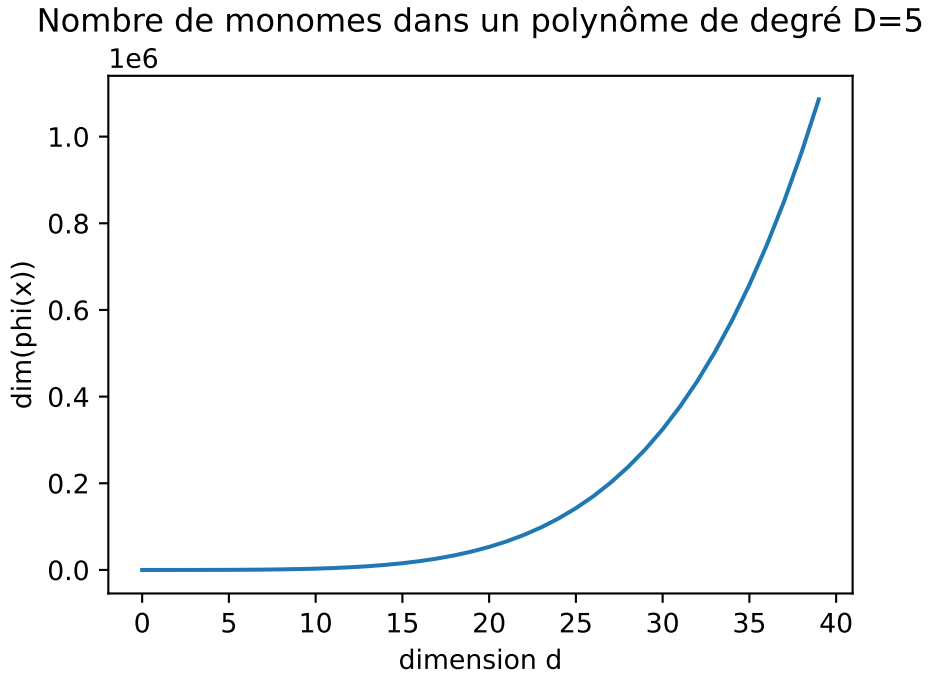
19.2 Le problème de la dimension

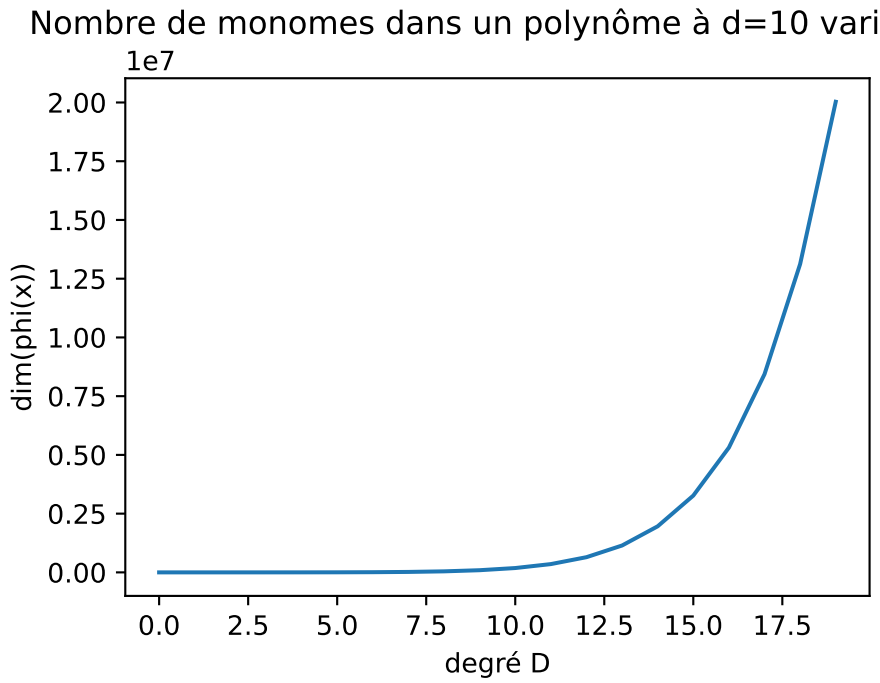
L'exemple ci-dessus considère une seule variable $x \in \mathbb{R}$ et ne pose pas de souci particulier. En revanche, pour $\boldsymbol{x} \in \mathbb{R}^d$, le modèle polynomial inclut tous les termes de degré $\leq D$ que l'on peut construire avec toutes les combinaisons de puissances des d variables x_k.

Très vite, la dimension du nouvel espace de représentation et des $\phi(\boldsymbol{x})$ devient trop grande pour être traitée efficacement :

$$dim(\phi(\boldsymbol{x})) = \binom{d+D}{d} = \frac{(d+D)!}{d!D!} = O\left(\frac{(d+D)^d}{d!}\right)$$

```
from scipy.special import comb

D = 5
d = np.arange(40)
dphi = comb(d+D, d)
plt.plot(d, dphi)
plt.xlabel("dimension d")
plt.ylabel("dim(phi(x))")
t=plt.title("Nombre de monomes dans un polynôme de degré D=5")

D = np.arange(20)
d = 10
dphi = comb(d+D, d)
plt.figure()
plt.plot(D, dphi)
plt.xlabel("degré D")
plt.ylabel("dim(phi(x))")
t=plt.title("Nombre de monomes dans un polynôme à d=10 variables")
```





19.3 Noyaux

Un noyau est une fonction $K : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ qui calcule à partir de deux points $\boldsymbol{x}$ et $\boldsymbol{x}'$ l'équivalent d'un produit scalaire entre les images $\phi(\boldsymbol{x})$ et $\phi(\boldsymbol{x}')$ de ces points :

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

Nous utilisons ici la notation $\langle \cdot, \cdot \rangle$ pour le produit scalaire au lieu du produit matriciel avec la transposée car elle est plus générale et permet de travailler avec des vecteurs en dimension infinie.

Exemple d'une projection 2D -> 3D

Soit $\mathcal{X} = \mathbb{R}^2$ et

$$\phi : \boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \mapsto \phi(\boldsymbol{x}) = \begin{bmatrix} x_1^2 \\ \sqrt{2} x_1 x_2 \\ x_2^2 \end{bmatrix}$$

Alors, le produit scalaire dans $\mathcal{X}_\phi \subset \mathbb{R}^3$ est

$$\begin{aligned} \langle \phi(\boldsymbol{x}), \phi(\boldsymbol{x}') \rangle &= \left\langle \begin{bmatrix} x_1^2 \\ \sqrt{2} x_1 x_2 \\ x_2^2 \end{bmatrix}, \begin{bmatrix} {x'_1}^2 \\ \sqrt{2} x'_1 x'_2 \\ {x'_2}^2 \end{bmatrix} \right\rangle = x_1^2 {x'_1}^2 + 2 x_1 x_2 x'_1 x'_2 + x_2^2 {x'_2}^2 \\ &= (x_1 x'_1 + x_2 x'_2)^2 \\ &= \langle \boldsymbol{x}, \boldsymbol{x}' \rangle^2 \end{aligned}$$

On peut donc définir une fonction $K(\boldsymbol{x}, \boldsymbol{x}') = \langle \boldsymbol{x}, \boldsymbol{x}' \rangle^2$ qui calcule les produits scalaires dans $\mathbb{R}^3$ à partir d'un produit scalaire dans $\mathbb{R}^2$ et une multiplication, sans jamais avoir à calculer les projections $\phi(\boldsymbol{x})$.

Les noyaux usuels sont les suivants.

- Le **noyau linéaire** : $K(\boldsymbol{x}, \boldsymbol{x}') = \langle \boldsymbol{x}, \boldsymbol{x}' \rangle$; ce noyau considère en fait la projection identité $\phi(\boldsymbol{x}) = \boldsymbol{x}$ et ne sert qu'à exprimer les modèles linéaires avec une formulation (ou un logiciel) unifiée avec les modèles non linéaires.
- Le **noyau polynomial** homogène, $K(\boldsymbol{x}, \boldsymbol{x}') = \langle \boldsymbol{x}, \boldsymbol{x}' \rangle^D$, ou inhomogène, $K(\boldsymbol{x}, \boldsymbol{x}') = (\langle \boldsymbol{x}, \boldsymbol{x}' \rangle + 1)^D$; ce noyau permet de construire des modèles polynomiaux.
- Le **noyau gaussien** (ou *« RBF »*)

$$K(\boldsymbol{x}, \boldsymbol{x}') = \exp\left(\frac{-\|\boldsymbol{x} - \boldsymbol{x}'\|^2}{2\sigma^2}\right) \tag{19.1}$$

  est le plus utilisé et correspond à une projection dans un espace de dimension infinie.

19.3.1 Noyaux définis positifs

Un noyau K est un noyau valide pour l'apprentissage s'il correspond à un produit scalaire dans un certain espace. Cela est vérifié si c'est une fonction symétrique $K : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ définie positive, c'est-à-dire telle que

$$\forall n \in \mathbb{N},\ \forall \{\boldsymbol{x}_i\}_{i=1}^n \in \mathcal{X}^n, \forall \{\alpha_i\}_{i=1}^n \in \mathbb{R}^n,\ \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j K(\boldsymbol{x}_i, \boldsymbol{x}_j) \geq 0 \tag{19.2}$$

En effet, dans ce cas, il existe ϕ telle que

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

Le critère Équation 19.2 peut être formulé de manière équivalente comme

$$\forall n \in \mathbb{N},\ \forall \{\boldsymbol{x}_i\}_{i=1}^n \in \mathcal{X}^n, \forall \boldsymbol{\alpha} \in \mathbb{R}^n,\ \boldsymbol{\alpha}^T \boldsymbol{K} \boldsymbol{\alpha} \geq 0$$

où $\boldsymbol{K}$ est la **matrice de noyau**

$$\boldsymbol{K} = \begin{bmatrix} K(\boldsymbol{x}_1, \boldsymbol{x}_1) & \dots & K(\boldsymbol{x}_1, \boldsymbol{x}_m) \\ \vdots & & \\ K(\boldsymbol{x}_m, \boldsymbol{x}_1) & \dots & K(\boldsymbol{x}_m, \boldsymbol{x}_m) \end{bmatrix}$$

qui doit donc être semi-définie positive.

19.3.2 Construction de noyaux

Si K_1 et K_2 sont des noyaux valides, alors les noyaux suivants sont aussi valides :

- $K(\boldsymbol{x}, \boldsymbol{x}') = aK_1(\boldsymbol{x}, \boldsymbol{x}')$ avec $a \geq 0$
- $K(\boldsymbol{x}, \boldsymbol{x}') = K_1(\boldsymbol{x}, \boldsymbol{x}') + a$ avec $a \geq 0$
- $K(\boldsymbol{x}, \boldsymbol{x}') = K_1(\boldsymbol{x}, \boldsymbol{x}') + K_2(\boldsymbol{x}, \boldsymbol{x}')$
- $K(\boldsymbol{x}, \boldsymbol{x}') = K_1(\boldsymbol{x}, \boldsymbol{x}')K_2(\boldsymbol{x}, \boldsymbol{x}')$
- $K(\boldsymbol{x}, \boldsymbol{x}') = \exp\left(K_1(\boldsymbol{x}, \boldsymbol{x}')\right)$

Pour le démontrer, il suffit de vérifier le critère Équation 19.2, qui est évident pour le premier.

Pour le second, on peut aussi identifier la projection correspondante : pour $\phi(\boldsymbol{x}) = \begin{bmatrix} \phi_1(\boldsymbol{x}) \\ \sqrt{a} \end{bmatrix}$ avec ϕ_1 la projection associée à K_1, on a

$$\phi(\boldsymbol{x})^T \phi(\boldsymbol{x}') = \phi_1(\boldsymbol{x})^T \phi_1(\boldsymbol{x}') + a = K_1(\boldsymbol{x}, \boldsymbol{x}') + a = K(\boldsymbol{x}, \boldsymbol{x}'),$$

donc K correspond bien à un produit scalaire.

Pour la somme de deux noyaux :

$$\sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i\alpha_j K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i\alpha_j K_1(\boldsymbol{x}_i, \boldsymbol{x}_j) + \sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i\alpha_j K_2(\boldsymbol{x}_i, \boldsymbol{x}_j)$$

qui est bien positif si le critère est validé pour K_1 et K_2.

Preuve pour le produit de noyaux

Pour le produit de noyaux, la matrice de noyau est donnée par le produit « case par case » $\odot$:

$$\boldsymbol{K} = \boldsymbol{K}_1 \odot \boldsymbol{K}_2 \quad \Leftrightarrow \quad (\boldsymbol{K})_{ij} = (\boldsymbol{K}_1)_{ij}(\boldsymbol{K}_2)_{ij}$$

pour tous les indices i et j.
Puisque que K_1 et K_2 sont définis positifs, leur matrice de noyau est semi-définie positive et leurs valeurs propres sont réelles et positives :

$$\boldsymbol{K}_1 = \sum_{k=1}^{n} \lambda_k \boldsymbol{u}_k \boldsymbol{u}_k^T, \quad \boldsymbol{K}_2 = \sum_{l=1}^{n} \mu_l \boldsymbol{v}_l \boldsymbol{v}_l^T$$

avec $\lambda_k \geq 0$, $\mu_l \geq 0$. Donc,

$$\begin{aligned}(\boldsymbol{K})_{ij} &= \left(\sum_{k=1}^{n} \lambda_k \boldsymbol{u}_k \boldsymbol{u}_k^T\right)_{ij} \left(\sum_{l=1}^{n} \mu_l \boldsymbol{v}_l \boldsymbol{v}_l^T\right)_{ij} \\ &= \sum_{k=1}^{n} \lambda_k \left(\boldsymbol{u}_k \boldsymbol{u}_k^T\right)_{ij} \left[\sum_{l=1}^{n} \mu_l \left(\boldsymbol{v}_l \boldsymbol{v}_l^T\right)_{ij}\right] \\ &= \sum_{k=1}^{n} \sum_{l=1}^{n} \lambda_k \mu_l (\boldsymbol{u}_k)_i (\boldsymbol{u}_k)_j (\boldsymbol{v}_l)_i \boldsymbol{v}_l)_j \\ &= \sum_{k=1}^{n} \sum_{l=1}^{n} \lambda_k \mu_l (\boldsymbol{u}_k \odot \boldsymbol{v}_l)_i (\boldsymbol{u}_k \odot \boldsymbol{v}_l)_j\end{aligned}$$

En posant $\gamma_p = \lambda_k \mu_l$ et $\boldsymbol{z}_p = \boldsymbol{u}_k \odot v_l$ pour $p = nk + l$, cela donne

$$(\boldsymbol{K})_{ij} = \sum_{p=1}^{n^2} \gamma_p (\boldsymbol{z}_p)_i (\boldsymbol{z}_p)_l$$

et

$$\boldsymbol{K} = \sum_{p=1}^{n^2} \gamma_p \boldsymbol{z}_p \boldsymbol{z}_p^T$$

Ainsi, pour tout $\boldsymbol{\alpha} \in \mathbb{R}^n$,

$$\boldsymbol{\alpha}^T \boldsymbol{K} \boldsymbol{\alpha} = \sum_{p=1}^{n^2} \gamma_p \boldsymbol{\alpha}^T \boldsymbol{z}_p \boldsymbol{z}_p^T \boldsymbol{\alpha} = \sum_{p=1}^{n^2} \gamma_p (\boldsymbol{\alpha}^T \boldsymbol{z}_p)^2 \geq 0$$

car $\gamma_p = \lambda_k \mu_l \geq 0$.

Pour terminer avec l'exponentielle d'un noyau valide, il faut tout d'abord considérer les polynômes à *coefficients positifs* de $K_1(\boldsymbol{x}, \boldsymbol{x}')$ qui sont valides par combinaison des résultats précédents. Ensuite, l'exponentielle peut s'exprimer comme une série de terme polynomial à coefficients positifs.

19.4 Astuce du noyau

L'astuce du noyau consiste à utiliser un noyau défini positif pour faire des calculs linéaires (des produits scalaires) dans un espace de (très) grande dimension dans lequel les données sont projetées sans jamais n'avoir à effectivement projeter les données ni faire ces calculs explicitement.

Pour l'apprentissage, cela signifie qu'il est possible d'apprendre des modèles non linéaires très complexes presque aussi efficacement que des modèles linéaires.

En effet, si un algorithme d'apprentissage linéaire ne fait intervenir les données $\boldsymbol{x}_i$ qu'au travers de produits scalaires entre elles (du type $\boldsymbol{x}_i^T \boldsymbol{x}_j$), alors créer un modèle non linéaire revient à appliquer cet algorithme en changeant les $\boldsymbol{x}_i^T \boldsymbol{x}_j$ en $\langle \phi(\boldsymbol{x}_i), \phi(\boldsymbol{x}_j) \rangle$ qui pourront être calculés efficacement par $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$.

19.5 Cadre fonctionnel et espace de Hilbert à noyau reproduisant

Il est aussi courant de reformuler les méthodes à noyau dans le contexte de l'apprentissage d'une fonction dans un espace de Hilbert à noyau reproduisant.

19.5.1 Espace de Hilbert

Les espaces de Hilbert peuvent être vus comme une généralisation des espaces euclidens classiques comme $\mathbb{R}^n$ permettant de traiter le cas de la dimension infinie, ou les espaces de fonctions.

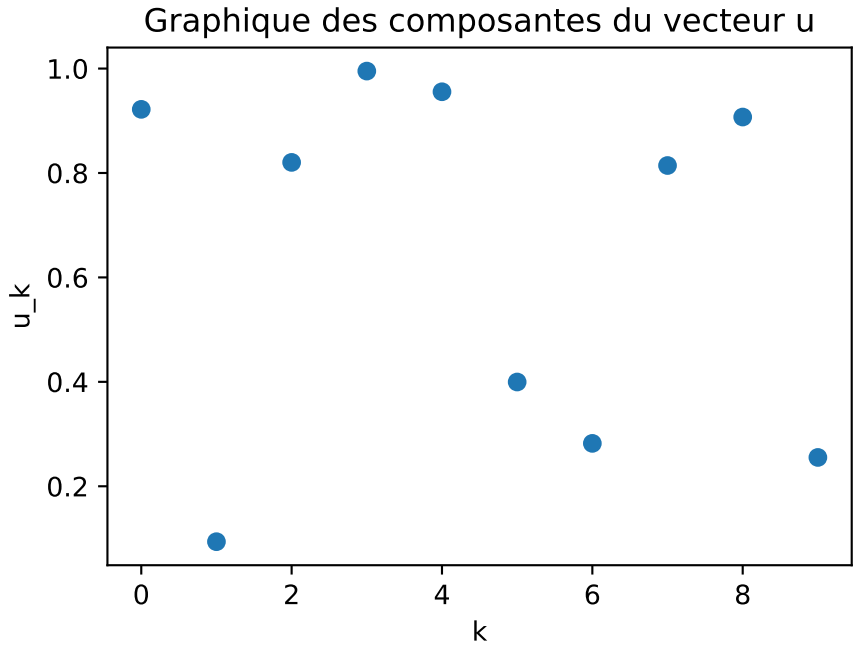
Espace de fonctions

Un espace vectoriel de dimension infinie est l'équivalent d'un espace de fonctions. Pour le voir, considérons les différentes représentations possibles d'un vecteur $\boldsymbol{u} \in \mathbb{R}^n$. Il y a tout d'abord la liste de ses composantes u_k, c'est-à-dire le tableau

$$\boldsymbol{u} = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{bmatrix}$$

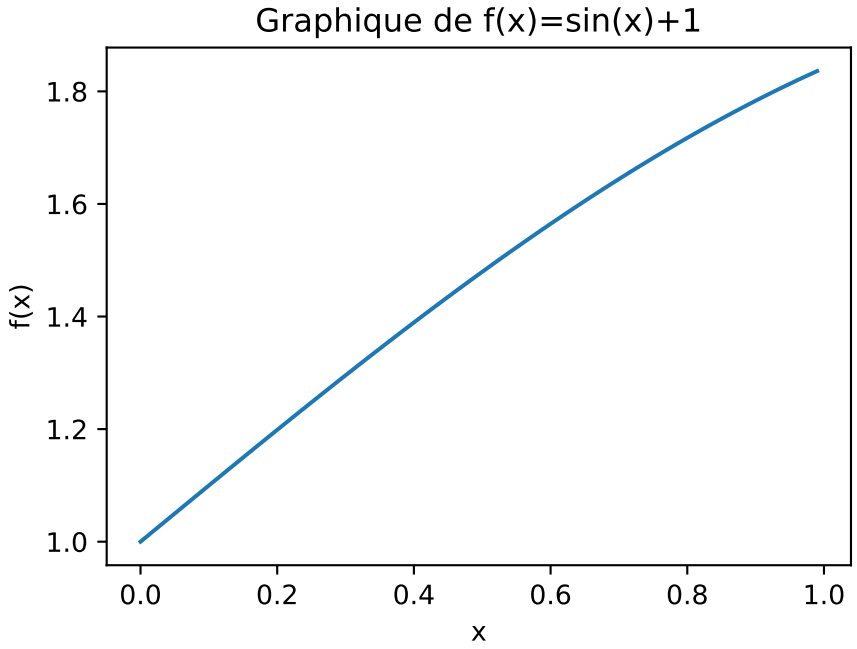
qui est en fait équivalent au graphique suivant :

```
u = np.random.rand(10)
plt.plot(u,"o")
plt.xlabel("k")
plt.ylabel("u_k")
t=plt.title("Graphique des composantes du vecteur u")
```



Pour une fonction f définie par exemple sur l'intervalle $[0, 1]$, nous avons l'habitude de travailler à partir de sa définition, par exemple $f(x) = \sin(x) + 1$, qui nous permet de connaître sa valeur $f(x)$ pour tout $x \in [0, 1]$. Mais le graphique de la fonction nous donne exactement la même information :

```
x = np.arange(0,1,0.01)
plt.plot(x, np.sin(x)+1)
plt.xlabel("x")
plt.ylabel("f(x)")
t=plt.title("Graphique de f(x)=sin(x)+1")
```



Si l'on fait le parallèle avec le vecteur $\boldsymbol{u}$, ce graphique est aussi équivalent au tableau de valeurs

$$\begin{bmatrix} f(0) \\ f(\epsilon) \\ \vdots \\ f(1) \end{bmatrix},$$

la principale différence étant que ce tableau possède une infinité de cases.
Ainsi, une fonction f peut être représentée par un tableau de taille infinie et donc comme un vecteur en dimension infinie.

Un espace de Hilbert est un espace vectoriel (potentiellement de dimension infinie) muni d'un produit scalaire $\langle \cdot, \cdot \rangle_{\mathcal{H}}$ qui induit une norme $\|x\|_{\mathcal{H}} = \sqrt{\langle x, x \rangle_{\mathcal{H}}}$ par rapport à laquelle il est complet[1].

Produit scalaire et norme de fonctions

Puisqu'une fonction f peut être vue comme un vecteur, il est possible de définir les opérations classiques sur les vecteurs, telles que $f = g + h$ qui signifie que pour tout x, $f(x) = (g + h)(x) = g(x) + h(x)$.
Le produit scalaire de deux vecteurs $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^n$ est la somme des produits de leurs composantes : $\langle \boldsymbol{u}, \boldsymbol{v} \rangle = \sum_{k=1}^{n} u_k v_k$. Par analogie, et par extension à une infinité de composantes indicées par $x \in [0, 1]$, nous pouvons définir le produit scalaire de deux

[1] Ici, l'espace est complet par rapport à la norme $\|\cdot\|_{\mathcal{H}}$ si toute séquence (x_n) d'éléments de cet espace qui converge, c'est-à-dire telle que $\lim_{n\to+\infty} \sup_{i,j>n} \|x_i - x_j\|_{\mathcal{H}} = 0$, convergence vers un élément de $\mathcal{H}$

fonctions définies sur $[0, 1]$ ainsi :

$$\langle f, g \rangle = \int_0^1 f(x)g(x)\ dx$$

Ce produit induit une norme naturelle pour l'espace de fonctions (l'équivalent de la norme euclidienne dans $\mathbb{R}^n$ donnée par $\|\boldsymbol{u}\| = \sqrt{\langle \boldsymbol{u}, \boldsymbol{v} \rangle}$) :

$$\|f\|_{\mathcal{H}} = \sqrt{\langle f, f \rangle_{\mathcal{H}}} = \sqrt{\int_0^1 f^2(x)\ dx}$$

19.5.2 Propriété de reproduction

Chaque noyau valide $K : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ (au sens de l'Équation 19.2) défini implicitement **un espace de Hilbert à noyau reproduisant** (ou *RKHS* en anglais) tel que, pour tout $\boldsymbol{x} \in \mathcal{X}$,

- la fonction $K(\boldsymbol{x}, \cdot)$ de son second argument et paramétrée par $\boldsymbol{x}$ appartient à l'espace $\mathcal{H}$;
- pour toute fonction $f \in \mathcal{H}$, la propriété de reproduction du noyau permet de calculer sa valeur comme un produit scalaire avec la fonction de noyau :

$$\langle f, K(\boldsymbol{x}, \cdot) \rangle_{\mathcal{H}} = f(\boldsymbol{x})$$

En particulier, en combinant ces deux propriétés, il vient

$$K(\boldsymbol{x}, \boldsymbol{x}') = \langle K(\boldsymbol{x}, \cdot), K(\boldsymbol{x}', \cdot) \rangle_{\mathcal{H}} \tag{19.3}$$

L'espace $\mathcal{H}$ peut être construit comme l'ensemble de toutes les combinaisons linéaires de fonctions de noyaux de norme finie :

$$\mathcal{H} = \left\{ f \in \mathbb{R}^{\mathcal{X}} \mid f = \sum_{i=1}^{\infty} \beta_i K(\boldsymbol{x}_i, \cdot), \beta_i \in \mathbb{R},\ \boldsymbol{x}_i \in \mathcal{X}, \|f\|_{\mathcal{H}} < \infty \right\}$$

En utilisant l'Équation 19.3, cela conduit à la formulation suivante pour la norme des fonctions de $\mathcal{H}$:

$$\|f\|_{\mathcal{H}}^2 = \langle f, f \rangle_{\mathcal{H}} = \sum_{i=1}^{\infty} \sum_{j=1}^{\infty} \beta_i \beta_j K(\boldsymbol{x}_i, \boldsymbol{x}_j)$$

19.5.3 Apprentissage dans un RKHS

Le théorème de reproduction permet de résoudre des problèmes d'apprentissage de fonctions dans un RKHS $\mathcal{H}$ (et donc un espace de dimension infinie) par des calculs en dimension finie. En particulier, il garantie que la solution du problème d'apprentissage régularisé

$$\min_{f \in \mathcal{H}} \sum_{i=1}^{m} \ell(f, \boldsymbol{x}_i, y_i) + \lambda \|f\|_{\mathcal{H}}$$

formulé comme un problème d'optimisation fonctionnelle (équivalent à un problème avec une infinité de variables correspondant aux valeurs de $f(\boldsymbol{x})$ pour tout $\boldsymbol{x} \in \mathcal{X}$) peut s'exprimer en fonction uniquement des m données $\boldsymbol{x}_i$ de la base d'apprentissage :

$$f^* = \sum_{i=1}^{m} \beta_i K(\boldsymbol{x}_i, \cdot)$$

Si l'on injecte cette solution dans le problème, alors il ne reste plus qu'à résoudre un problème d'optimisation à m variables β_i.

Preuve

Chaque fonction $f \in \mathcal{H}$ peut être décomposée en la somme de deux fonctions,

$$f = u + v,$$

où u est la projection de f sur le sous-espace S engendré par les $K(\boldsymbol{x}_i, \cdot)$ pour les $\boldsymbol{x}_i$ de la base d'apprentissage, c'est-à-dire

$$u = \sum_{i=1}^{m} \beta_i K(\boldsymbol{x}_i, \cdot),$$

et v contient tout ce qui de f n'a pas pu s'exprimer dans u : v appartient au complément orthogonal du sous-espace S, c'est-à-dire que $v \perp K(\boldsymbol{x}_i, \cdot)$ au sens où $\langle v, K(\boldsymbol{x}_i, \cdot)\rangle_{\mathcal{H}} = 0$. Ainsi, pour tout $\boldsymbol{x}_i$ de la base d'apprentissage, la propriété de reproduction donne

$$f(\boldsymbol{x}_i) = u(\boldsymbol{x}_i) + v(\boldsymbol{x}_i) = u(\boldsymbol{x}_i) + \langle v, K(\boldsymbol{x}_i, \cdot)\rangle_{\mathcal{H}} = u(\boldsymbol{x}_i)$$

et les valeurs de $f(\boldsymbol{x}_i)$ et donc le terme d'erreur dans la fonction objectif ne dépendent pas de v. Il s'agit donc simplement de trouver v qui minimise la norme de f, où

$$\|f\|_{\mathcal{H}}^2 = \|u + v\|_{\mathcal{H}}^2 = \langle u + v, u + v\rangle_{\mathcal{H}} = \langle u, u\rangle_{\mathcal{H}} + \langle v, v\rangle_{\mathcal{H}} + 2\langle u, v\rangle_{\mathcal{H}}$$

Or, la forme de u et v implique qu'ils sont orthogonaux et donc $\langle u, v\rangle_{\mathcal{H}} = 0$, ce qui donne

$$\|f\|_{\mathcal{H}}^2 = \langle u, u\rangle_{\mathcal{H}} + \langle v, v\rangle_{\mathcal{H}} = \|u\|_{\mathcal{H}}^2 + \|v\|_{\mathcal{H}}^2$$

Ainsi, la norme de f est minimisée en choisissant $v = 0$ et donc $f = u$.

20 Régression ridge à noyau (KRR)

La régression ridge est une méthode de régularisation basée sur la méthode des moindres carrés pour la régression linéaire.

La régression ridge à noyau (*kernel ridge regression*, KRR) étend cette méthode à la régression non linéaire et non paramétrique, c'est-à-dire que la structure même du modèle est déterminée à partir des données et non fixée à l'avance, grâce aux noyaux.

Le modèle créée s'exprime ainsi :

$$\hat{f}(\boldsymbol{x}) = \sum_{i=1}^{m} \beta_i K(\boldsymbol{x}_i, \boldsymbol{x})$$

avec l'apprentissage des paramètres β_i réalisé simplement par

$$\boldsymbol{\beta} = (\boldsymbol{K} + \lambda \boldsymbol{I})^{-1} \boldsymbol{y}$$

(voir ci-dessous pour les détails).

20.1 Formulation dans le RKHS

Apprendre un modèle de régression non linéaire peut être formulé comme l'apprentissage d'une fonction f dans un certain espace de fonctions. La méthode KRR se place dans un espace de Hilbert à noyau reproduisant (RKHS) $\mathcal{H}$ engendré par un noyau K et cherche la fonction qui minimise à la fois l'erreur quadratique moyenne et un terme de régularisation donné par la norme de cette fonction au carré :

$$\hat{f} = \arg\min_{f \in \mathcal{H}} \sum_{i=1}^{m} (y_i - f(\boldsymbol{x}_i))^2 + \lambda \|f\|^2$$

où l'hyperparamètre $\lambda > 0$ règle le compromis entre l'attache aux données et la régularisation.

20.2 Formulation duale

Le problème d'apprentissage KRR peut être résolu efficacement grâce au théorème de représentation qui s'applique dans le RKHS. Celui-ci garantit que la solution est de la forme

$$\hat{f} = \sum_{j=1}^{m} \beta_j K(\boldsymbol{x}_j, \cdot)$$

avec les m exemples $\boldsymbol{x}_j$ de la base d'apprentissage. Cela nous permet de reformuler la norme de f comme

$$\begin{aligned}\|f\|^2 &= \left\langle \sum_{i=1}^{m} \beta_i K(\boldsymbol{x}_i, \cdot), \sum_{j=1}^{m} \beta_j K(\boldsymbol{x}_j, \cdot) \right\rangle \\ &= \sum_{i=1}^{m} \beta_i \sum_{j=1}^{m} \beta_j \left\langle K(\boldsymbol{x}_i, \cdot), K(\boldsymbol{x}_j, \cdot) \right\rangle = \sum_{i=1}^{m} \sum_{j=1}^{m} \beta_i \beta_j K(\boldsymbol{x}_i, \boldsymbol{x}_j)\end{aligned}$$

où la dernière étape vient de la propriété de reproduction du noyau.

En réinjectant cette forme dans le problème, nous obtenons la **forme duale** exprimée par rapport aux m variables réelles β_i :

$$\min_{\beta_1,\ldots,\beta_m} \sum_{i=1}^{m} \left(y_i - \sum_{j=1}^{m} \beta_j K(\boldsymbol{x}_j, \boldsymbol{x}_i) \right)^2 + \lambda \sum_{i=1}^{m} \sum_{j=1}^{m} \beta_i \beta_j K(\boldsymbol{x}_i, \boldsymbol{x}_j)$$

ou, sous forme matricielle avec $\boldsymbol{\beta} \in \mathbb{R}^m$ concaténant tous les β_i :

$$\min_{\boldsymbol{\beta} \in \mathbb{R}^m} \|\boldsymbol{y} - \boldsymbol{K}\boldsymbol{\beta}\|^2 + \lambda \boldsymbol{\beta}^T \boldsymbol{K} \boldsymbol{\beta}$$

avec la matrice de noyau

$$\boldsymbol{K} = \begin{bmatrix} K(\boldsymbol{x}_1, \boldsymbol{x}_1) & \ldots & K(\boldsymbol{x}_1, \boldsymbol{x}_m) \\ \vdots & & \\ K(\boldsymbol{x}_m, \boldsymbol{x}_1) & \ldots & K(\boldsymbol{x}_m, \boldsymbol{x}_m) \end{bmatrix}$$

20.3 Solution

La solution de la forme duale peut être calculée simplement en cherchant le vecteur $\boldsymbol{\beta}$ qui annule le gradient de l'objectif

$$J(\boldsymbol{\beta}) = \|\boldsymbol{y} - \boldsymbol{K}\boldsymbol{\beta}\|^2 + \lambda \boldsymbol{\beta}^T \boldsymbol{K} \boldsymbol{\beta}$$

à minimiser, comme pour la régression ridge classique :

$$\begin{aligned}\frac{\mathrm{d}J(\boldsymbol{\beta})}{\mathrm{d}\boldsymbol{\beta}} &= -2\boldsymbol{K}^T(\boldsymbol{y} - \boldsymbol{K}\boldsymbol{\beta}) + 2\lambda \boldsymbol{K}\boldsymbol{\beta} \\ &= -2\boldsymbol{K}^T\boldsymbol{y} + 2(\boldsymbol{K}^T\boldsymbol{K} + \lambda \boldsymbol{K})\boldsymbol{\beta} \\ &= -2\boldsymbol{K}\boldsymbol{y} + 2(\boldsymbol{K}\boldsymbol{K} + \lambda \boldsymbol{K})\boldsymbol{\beta}\end{aligned}$$

où la dernière étape utilise la symétrie de la matrice de noyau (elle-même induite par la symétrie de la fonction noyau K).

Cela donne une solution caractérisée par

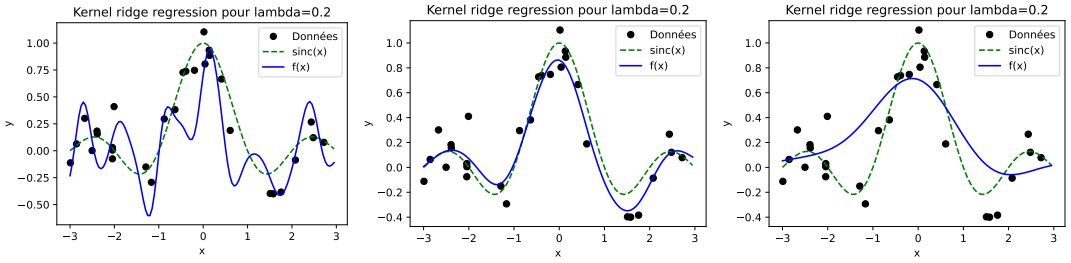
$$\boldsymbol{\beta} = (\boldsymbol{K}\boldsymbol{K} + \lambda \boldsymbol{K})^{-1}\boldsymbol{K}\boldsymbol{y} = [\boldsymbol{K}(\boldsymbol{K} + \lambda \boldsymbol{I})]^{-1}\boldsymbol{K}\boldsymbol{y} = (\boldsymbol{K} + \lambda \boldsymbol{I})^{-1}\boldsymbol{K}^{-1}\boldsymbol{K}\boldsymbol{y}$$

et donc un vecteur de paramètres obtenu simplement par

$$\boldsymbol{\beta} = (\boldsymbol{K} + \lambda \boldsymbol{I})^{-1} \boldsymbol{y}$$

Exemple

```
def kernel(X1,X2,sigma):
    K = np.zeros((len(X1),len(X2)))
    for i in range(len(X1)):
        for j in range(len(X2)):
            K[i,j] = np.exp(-np.sum((X1[i] - X2[j])**2)/(2*sigma**2))

    return K

def krr(X,y, sigma, Lambda):
    K = kernel(X,X,sigma)
    beta = np.linalg.inv(K + Lambda * np.eye(len(y))) @ y
    return beta

def krrpred(X,Xt,beta,sigma):
    K = kernel(Xt,X,sigma)
    return K@beta

x = 6*np.random.rand(30)-3
y = np.sinc(x) + np.random.randn(30)*0.15

Lambda = 0.5
xt = np.arange(-3,3,0.05)
for sigma in [0.2, 0.5, 1]:
    beta = krr(x,y, Lambda, sigma)
    plt.figure()
    plt.plot(x,y, "ok")
    plt.plot(xt,np.sinc(xt),"--g")
    plt.plot(xt, krrpred(x, xt,beta, sigma), "-b")
    plt.legend(["Données", "sinc(x)", "f(x)"])
    plt.xlabel("x")
    plt.ylabel("y")
    t=plt.title("Kernel ridge regression pour lambda=0.2")
```


Kernel ridge regression pour lambda=0.2
Données
sinc(x)
f(x)
x
y
Kernel ridge regression pour lambda=0.2
Données
sinc(x)
f(x)
x
y
Kernel ridge regression pour lambda=0.2
Données
sinc(x)
f(x)
x
y

21 Réseaux de neurones

Un réseau de neurones est un modèle de prédiction adapté à la classification ou la régression.

Son architecture est définie par un graphe orienté dans lequel les connexions entre les nœuds sont pondérées. Chaque nœud est un neurone artificiel qui implémente un calcul simple pour produire une sortie à partir de ses entrées.

Les neurones sont typiquement organisés en couches qui peuvent être de différents types.

21.1 Unité de calcul de base : le neurone (artificiel)

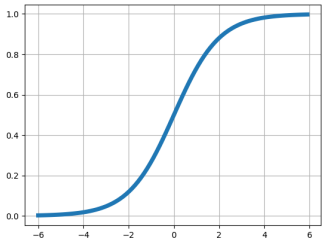
Un neurone artificiel implémente la fonction

$$f(\boldsymbol{z}) = \sigma\left(\sum_{k=1}^{n} w_k z_k + b\right) = \sigma(\boldsymbol{w}^T \boldsymbol{z} + b) = \sigma(u)$$

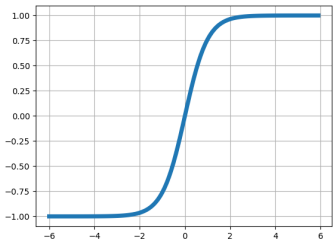
où

- $\boldsymbol{z} \in \mathbb{R}^n$ est le vecteur des entrées
- $\boldsymbol{w} \in \mathbb{R}^n$ est le vecteur des poids *(paramètre à apprendre)*
- b est le terme de biais *(paramètre à apprendre)*
- σ est la fonction d'activation
- u est le niveau d'activation (somme pondérée des entrées)

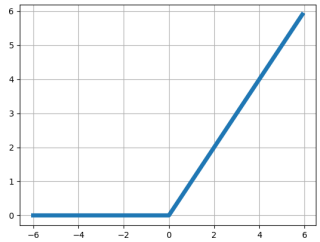
Il existe beaucoup de fonctions d'activation différentes, dont certaines peuvent originellement être vues comme des approximations lisses et dérivables de la fonction seuil utilisée par le perceptron. Voici les plus courantes :



(a) sigmoïde : $\sigma(u) = \frac{1}{1+e^{-u}}$



(b) tangente hyperbolique : $\sigma(u) = \tanh(u)$

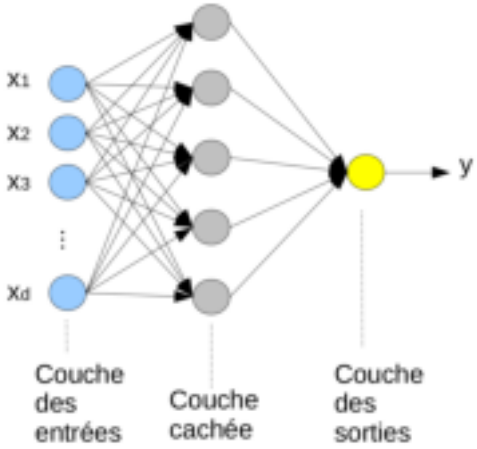


(c) ReLU : $\sigma(u) = \max\{0, u\}$

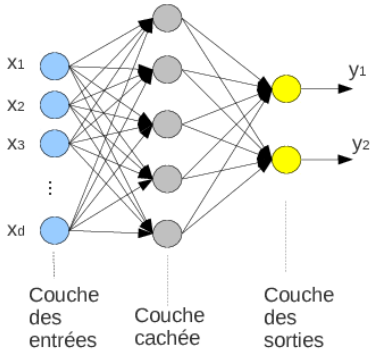
21.2 Perceptron multicouches (MLP)

Le perceptron multicouches (*multilayer perceptron, MLP*) est le réseau de neurones le plus simple, composé de *couches denses*, c'est-à-dire des couches de neurones dans lesquelles chaque neurone est connecté à tous les neurones de la couche précédente par ses entrées et tous ceux de la couche suivante par sa sortie.

Un réseau de neurones peut aisément prédire plusieurs étiquettes simultanément en ajoutant des neurones sur sa dernière couche, appelée couche de sortie. Les couches intermédiaires entre les entrées et les sorties sont appelées *couches cachées*, car le réseau de neuronne peut être vu comme un modèle « boîte noire » dont le comportement interne n'est pas nécessairement représentatif du processus modélisé.



(a) Perceptron multicouches



(b) MLP à sorties multiples

En général, tous les neurones d'une couche sont du même type (avec la même fonction d'activation), mais ils peuvent être différents d'une couche à l'autre.

La fonction calculée par le réseau se définit alors de manière récursive par composition, par exemple :

$$f(\boldsymbol{x}) = \sigma_2\left(\boldsymbol{w}_2^T \sigma_1(\boldsymbol{W}_1\boldsymbol{x} + b_1) + b_2\right)$$

où σ_1, $\boldsymbol{W}_1$, $\boldsymbol{b}_1$ sont la fonction d'activation et les paramètres de l'unique couche cachée et σ_2, $\boldsymbol{w}_2$, b_2 ceux utilisés par la couche de sortie. Ici,

$$\boldsymbol{u}_1 = \boldsymbol{W}_1\boldsymbol{x} + b_1$$

représente le niveau d'activation de l'ensemble des n neurones de la couche cachée de paramètres

$$\boldsymbol{W}_1 = \begin{bmatrix} \boldsymbol{w}_{1,1}^T \\ \vdots \\ \boldsymbol{w}_{1,n}^T \end{bmatrix}, \quad \boldsymbol{b}_1 = \begin{bmatrix} b_{1,1} \\ \vdots \\ b_{1,n} \end{bmatrix}$$

où $\boldsymbol{w}_{1,j}$, $b_{1,j}$ sont les paramètres du $j$ème neurone de cette couche.

De manière résumée, l'**apprentissage d'un réseau de neurones** revient à déterminer la structure du réseau et les valeurs des poids (et biais) affectés aux connexions. En général, cela se fait en deux temps :

1. fixer la structure du réseau et choisir les fonctions d'activations ;
2. calculer les poids de manière à minimiser les erreurs de prédiction sur la base d'apprentissage.

21.2.1 Apprentissage des poids : minimisation de l'erreur sur les données

L'apprentissage des poids d'un réseau de neurones vise à sélectionner les paramètres qui conduisent à la meilleure prédiction possible sur la base d'aprentissage. Pour une fonction de perte ℓ, par exemple $\ell(y_i, f(\boldsymbol{x}_i)) = (y_i - f(\boldsymbol{x}_i))^2$ pour la régression, l'apprentissage revient à résoudre le problème d'optimisation

$$\min_{\boldsymbol{w}} J(\boldsymbol{w}) = \sum_{i=1}^{m} \ell(y_i, f(\boldsymbol{x}_i))$$

où f représente la fonction calculée par le réseau de neurones et $\boldsymbol{w}$ concatène tous ses paramètres (y compris les biais).

La méthode de base utilisée pour minimiser l'erreur est la descente de grandient, car c'est une des méthodes les plus simples pour minimiser une fonction sans contraintes. Par ailleurs, la taille des réseaux de neurones (et la dimension du problème d'optimisation) empêche bien souvent d'utiliser des méthodes plus avancées basées sur les dérivées du second ordre par exemple.

À l'inverse, la descente de gradient ne nécessite que le calcul des dérivées premières et consiste à appliquer les itérations

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \mu \frac{\mathrm{d}J(\boldsymbol{w})}{\mathrm{d}\boldsymbol{w}} = \boldsymbol{w} - \sum_{i=1}^{m} \mu \frac{\mathrm{d}\ell(y_i, f(\boldsymbol{x}_i))}{\mathrm{d}\boldsymbol{w}}$$

où $\mu \in (0, 1)$ est un hyperparamètre appelé taux d'apprentissage.

Cependant, ces itérations requièrent le calcul de m dérivées à chaque itération, ce qui peut se révéler trop fastidieux sur les jeux de données avec un grande nombre m d'exemples.

21.2.2 Descente de gradient stochastique

La descente de gradient stochastique offre une approximation de la descente de gradient permettant d'accélérer les itérations. L'idée est simple : au lieu de calculer le gradient de l'erreur totale, *chaque itération ne s'intéresse qu'à l'erreur sur un seul exemple* : pour i tiré aléatoirement dans $\{1, \ldots, m\}$,

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \mu \frac{\mathrm{d}\ell(y_i, f(\boldsymbol{x}_i))}{\mathrm{d}\boldsymbol{w}}$$

Les calculs sont ici beaucoup plus légers à chaque itération, mais la convergence peut nécessiter en contrepartie beaucoup plus d'itérations.

21.2.3 Calcul des dérivées et rétropropagation du gradient

Dans un réseau de neurones, les calculs de dérivées peuvent être largement optimisés dans le cadre de la rétropropagation du gradient.

21.2.3.1 Dérivées en chaîne (avec une seule variable)

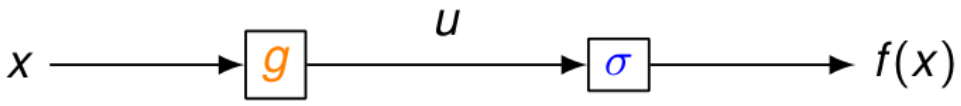


Figure 21.3: Composition de deux fonctions d'une variable

Pour

$$f(x) = \sigma(u) = \sigma(\ g(x)\ ), \quad avec\ u = g(x)$$

la règle de dérivation en chaîne donne

$$\frac{\mathrm{d}f(x)}{\mathrm{d}x} = \frac{\mathrm{d}f(x)}{\mathrm{d}u}\frac{\mathrm{d}u}{\mathrm{d}x} = \sigma'(u)g'(x)$$

Exemple : calculer $\frac{\mathrm{d}f(x)}{\mathrm{d}x}$ pour

$$f(x) = (5 - 3x)^2 = u^2, \quad u = g(x) = 5 - 3x$$

Solution

$$\sigma'(u) = 2u, \quad g'(x) = -3$$
$$\Rightarrow \quad \frac{\mathrm{d}f(x)}{\mathrm{d}x} = -6(5 - 3x)$$

Avec plusieurs compositions :

$$f(x) = \sigma(\ g(\ h(x)\ )\ ) = \sigma(u), \quad u = g(v), \quad v = h(x)$$
$$\Rightarrow \frac{\mathrm{d}f(x)}{\mathrm{d}x} = \frac{\mathrm{d}f(x)}{\mathrm{d}u}\frac{\mathrm{d}u}{\mathrm{d}x} = \frac{\mathrm{d}f(x)}{\mathrm{d}u}\frac{\mathrm{d}u}{\mathrm{d}v}\frac{\mathrm{d}v}{\mathrm{d}x} = \sigma'(u)g'(v)h'(x)$$

Si la dérivée de f par rapport à u ou à v a déjà été calculée, on peut donc économiser des calculs.

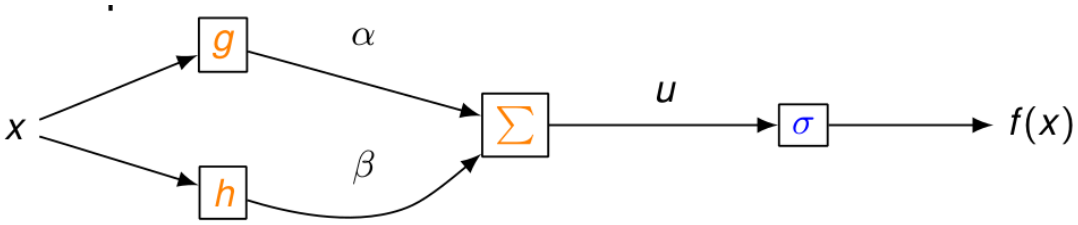
Exemple : calculer $\frac{\mathrm{d}f(x)}{\mathrm{d}x}$ pour

$$f(x) = (5 - 3e^{-4x})^2 = u^2, \quad u = 5 - 3v, \quad v = h(x) = e^{-4x}$$

Solution

$$\sigma'(u) = 2u, \quad g'(v) = -3, \quad h'(x) = -4e^{-4x}$$
$$\Rightarrow \quad \frac{\mathrm{d}f(x)}{\mathrm{d}x} = 24e^{-4x}(5 - 3e^{-4x})$$

Avec plusieurs chemins :



$$f(x) = \sigma(u), \quad u = \alpha g(x) + \beta h(x)$$
$$\Rightarrow \frac{\mathrm{d}f(x)}{\mathrm{d}x} = \frac{\mathrm{d}f(x)}{\mathrm{d}u}\frac{\mathrm{d}u}{\mathrm{d}x} = \sigma'(u)(\alpha g'(x) + \beta h'(x))$$

Exemple : calculer $\frac{\mathrm{d}f(x)}{\mathrm{d}x}$ pour

$$f(x) = \exp\left(2(5 - 3x)^2 + 0.5(2 + 3x)^3\right) = \exp(u) = \sigma(u),$$
$$g(x) = (5 - 3x)^2, \quad h(x) = (2 + 3x)^3, \quad \alpha = 2, \ \beta = 0.5$$

Solution

$$\sigma'(u) = \exp(u), \quad g'(x) = -6(5 - 3x), \quad h'(x) = 9(2 + 3x)^2$$
$$\frac{\mathrm{d}f(x)}{\mathrm{d}x} = \exp\left(2(5 - 3x)^2 + 0.5(2 + 3x)^3\right)\left(-12(5 - 3x) + 4.5(2 + 3x)^2\right)$$

21.2.3.2 Calcul de gradients en chaîne

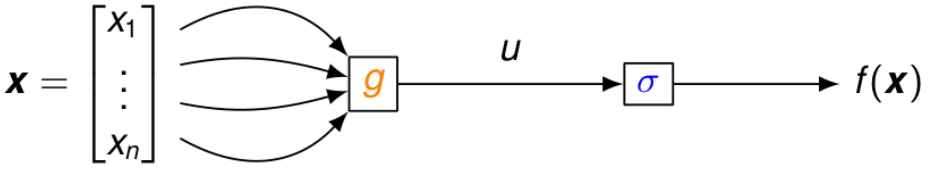


Figure 21.4: Composition de fonctions de plusieurs variables

Pour

$$f(\boldsymbol{x}) = \sigma(\ g(\boldsymbol{x})\ ) = \sigma(u), \quad \textit{avec } u = g(\boldsymbol{x}) = g(x_1, \ldots, x_n)$$

avec plusieurs variables, la dérivée devient un gradient, mais les règles de composition restent identiques car elles s'appliquent aux dérivées partielles :

$$\frac{\partial f(\boldsymbol{x})}{\partial x_k} = \frac{\mathrm{d}f(\boldsymbol{x})}{\mathrm{d}u}\frac{\partial u}{\partial x_k} \quad \Rightarrow \frac{\mathrm{d}f(\boldsymbol{x})}{\mathrm{d}\boldsymbol{x}} = \frac{\mathrm{d}f(\boldsymbol{x})}{\mathrm{d}u}\frac{\mathrm{d}u}{\mathrm{d}\boldsymbol{x}} = \sigma'(u)\frac{\mathrm{d}g(\boldsymbol{x})}{\mathrm{d}\boldsymbol{x}}$$

Exemple : calculer le gradient de f pour

$$f(\boldsymbol{x}) = (5x_1 - 3x_1x_2)^2 = u^2, \quad u = g(\boldsymbol{x}) = 5x_1 - 3x_1x_2$$

Solution

$$\sigma'(u) = 2u, \quad \frac{\mathrm{d}g(\boldsymbol{x})}{\mathrm{d}\boldsymbol{x}} = \begin{bmatrix} 5 - 3x_2 \\ -3x_1 \end{bmatrix}$$

$$\Rightarrow \quad \frac{\mathrm{d}f(\boldsymbol{x})}{\mathrm{d}\boldsymbol{x}} = 2(5x_1 - 3x_1x_2)\begin{bmatrix} 5 - 3x_2 \\ -3x_1 \end{bmatrix}$$

Pour la composition avec une fonction à valeur vectorielle :

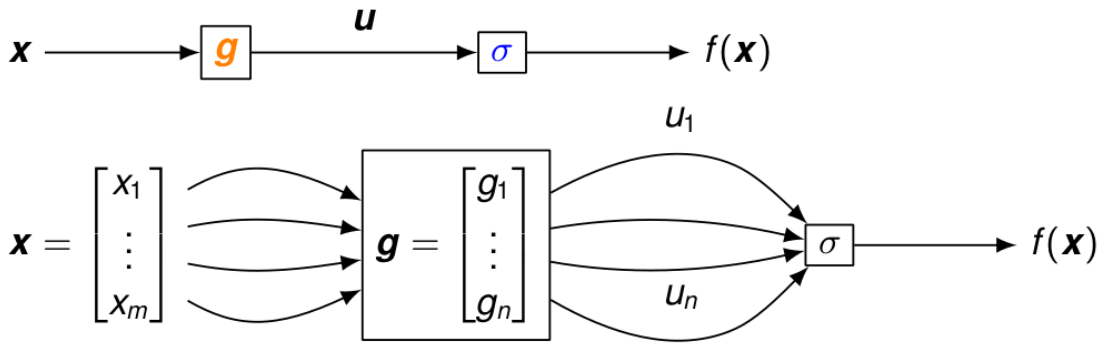


Figure 21.5: Composition avec une fonction vectorielle

$$f(\boldsymbol{x}) = \sigma(\ \boldsymbol{g}(\ \boldsymbol{x}\ )\ ) = \sigma(\boldsymbol{u}) = \sigma(u_1, \ldots, u_n), \quad \boldsymbol{u} = \boldsymbol{g}(\boldsymbol{x}) = \begin{bmatrix} g_1(x_1, \ldots, x_m) \\ \vdots \\ g_n(x_1, \ldots, x_m) \end{bmatrix}$$

La dérivée partielle de $f(\boldsymbol{x})$ par rapport à x_k est donnée par la somme sur tous les chemins de x_k à $f(\boldsymbol{x})$:

$$\frac{\partial f(\boldsymbol{x})}{\partial x_k} = \sum_{j=1}^{n} \frac{\partial f(\boldsymbol{x})}{\partial u_j}\frac{\partial u_j}{\partial x_k} = \begin{bmatrix} \frac{\partial u_1}{\partial x_k} & \cdots & \frac{\partial u_n}{\partial x_k} \end{bmatrix} \begin{bmatrix} \frac{\partial f(\boldsymbol{x})}{\partial u_1} \\ \vdots \\ \frac{\partial f(\boldsymbol{x})}{\partial u_n} \end{bmatrix}$$

$$\Rightarrow \frac{\mathrm{d}f(\boldsymbol{x})}{\mathrm{d}\boldsymbol{x}} = \begin{bmatrix} \frac{\partial f(\boldsymbol{x})}{\partial x_1} \\ \vdots \\ \frac{\partial f(\boldsymbol{x})}{\partial x_m} \end{bmatrix} = \begin{bmatrix} \frac{\partial u_1}{\partial x_1} & \cdots & \frac{\partial u_n}{\partial x_1} \\ \vdots & & \vdots \\ \frac{\partial u_1}{\partial x_m} & \cdots & \frac{\partial u_n}{\partial x_m} \end{bmatrix} \begin{bmatrix} \frac{\partial f(\boldsymbol{x})}{\partial u_1} \\ \vdots \\ \frac{\partial f(\boldsymbol{x})}{\partial u_n} \end{bmatrix}$$

En introduisant la matrice jacobienne,

$$\frac{\mathrm{d}\boldsymbol{u}}{\mathrm{d}\boldsymbol{x}} = \begin{bmatrix} \left(\frac{\mathrm{d}u_1}{\mathrm{d}\boldsymbol{x}}\right)^\top \\ \vdots \\ \left(\frac{\mathrm{d}u_n}{\mathrm{d}\boldsymbol{x}}\right)^\top \end{bmatrix} = \begin{bmatrix} \frac{\partial u_1}{\partial x_1} & \cdots & \frac{\partial u_1}{\partial x_m} \\ \vdots & & \vdots \\ \frac{\partial u_n}{\partial x_1} & \cdots & \frac{\partial u_n}{\partial x_m} \end{bmatrix}$$

on retrouve donc la règle de dérivation en chaîne :

$$\frac{\mathrm{d}f(\boldsymbol{x})}{\mathrm{d}\boldsymbol{x}} = \left(\frac{\mathrm{d}\boldsymbol{u}}{\mathrm{d}\boldsymbol{x}}\right)^\top \frac{\mathrm{d}f(\boldsymbol{x})}{\mathrm{d}\boldsymbol{u}}$$

Il faut cependant faire attention à l'ordre des termes car le produit matriciel n'est pas commutatif.

Pour la composition avec plusieurs fonctions à valeur vectorielle :

Figure 21.6: Composition avec deux fonctions vectorielles

$$f(\boldsymbol{x}) = \sigma(\, \boldsymbol{g}(\, \boldsymbol{h}(\boldsymbol{x})\,)\,) = \sigma(\boldsymbol{u}), \quad \boldsymbol{u} = \boldsymbol{g}(\boldsymbol{v}),\ \boldsymbol{v} = \boldsymbol{h}(\boldsymbol{x})$$
$$\Rightarrow \frac{\mathrm{d}f(\boldsymbol{x})}{\mathrm{d}\boldsymbol{x}} = \left(\frac{\mathrm{d}\boldsymbol{v}}{\mathrm{d}\boldsymbol{x}}\right)^\top \frac{\mathrm{d}f(\boldsymbol{x})}{\mathrm{d}\boldsymbol{v}} = \left(\frac{\mathrm{d}\boldsymbol{v}}{\mathrm{d}\boldsymbol{x}}\right)^\top \left(\frac{\mathrm{d}\boldsymbol{u}}{\mathrm{d}\boldsymbol{v}}\right)^\top \frac{\mathrm{d}f(\boldsymbol{x})}{\mathrm{d}\boldsymbol{u}}$$

où

- $\frac{\mathrm{d}\boldsymbol{v}}{\mathrm{d}\boldsymbol{x}}$ est la matrice jacobienne de $\boldsymbol{h}$
- $\frac{\mathrm{d}\boldsymbol{u}}{\mathrm{d}\boldsymbol{v}}$ est la matrice jacobienne de $\boldsymbol{g}$

Le nombre d'opérations dans le produit matriciel ci-dessous peut être optimisé en effectuant les calculs dans le bon ordre.

- De gauche à droite : $O(mn_1n_2)$ opérations pour $\boldsymbol{Z} = \left(\frac{\mathrm{d}\boldsymbol{v}}{\mathrm{d}\boldsymbol{x}}\right)^\top \left(\frac{\mathrm{d}\boldsymbol{u}}{\mathrm{d}\boldsymbol{v}}\right)^\top$, puis $O(mn_2)$ opérations pour $\boldsymbol{Z}\frac{\mathrm{d}f(\boldsymbol{x})}{\mathrm{d}\boldsymbol{u}}$.
- De droite à gauche : $O(n_1n_2)$ opéartions pour $\boldsymbol{z} = \left(\frac{\mathrm{d}\boldsymbol{u}}{\mathrm{d}\boldsymbol{v}}\right)^\top \frac{\mathrm{d}f(\boldsymbol{x})}{\mathrm{d}\boldsymbol{u}}$, puis $O(mn_1)$ opérations pour $\left(\frac{\mathrm{d}\boldsymbol{v}}{\mathrm{d}\boldsymbol{x}}\right)^\top \boldsymbol{z}$.

La **rétropropagation du gradient** repose sur l'idée d'effectuer les calculs de droite à gauche en mémorisant la plupart des termes : si $\boldsymbol{z} = \frac{\mathrm{d}f(\boldsymbol{x})}{\mathrm{d}\boldsymbol{v}}$ a déjà été calculé, uniquement $O(mn_1)$ opérations sont requises.

21.2.3.3 Rétropropagation du gradient dans un réseau de neurones

Graphe de calcul pour

$$f(\boldsymbol{x}) = \sigma_2(\ \boldsymbol{w}_2^\top \sigma_1(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1) + b_2)$$

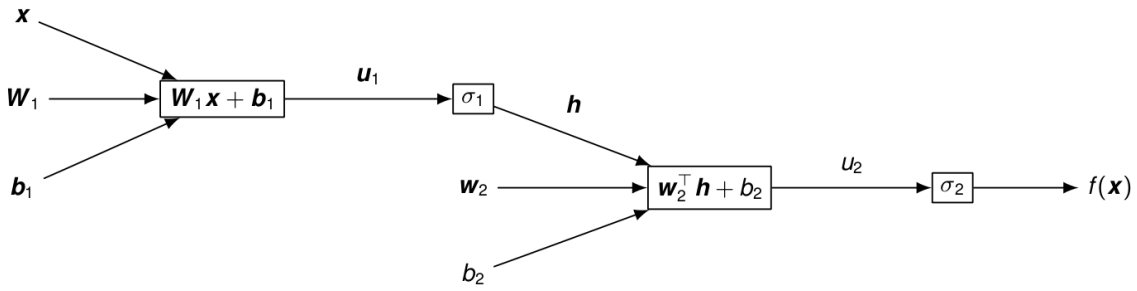


Figure 21.7: Graphe de calcul d'un MLP

$$\boldsymbol{u}_1 = \boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1 = \begin{bmatrix} \boldsymbol{w}_{1,1}^\top \boldsymbol{x} + b_{1,1} \\ \vdots \\ \boldsymbol{w}_{1,n}^\top \boldsymbol{x} + b_{1,n} \end{bmatrix}, \qquad \boldsymbol{h} = \sigma_1(\boldsymbol{u}_1) = \begin{bmatrix} \sigma_1(u_{1,1}) \\ \vdots \\ \sigma_1(u_{1,n}) \end{bmatrix}$$

Les dérivées par rapport aux paramètres de la couche de sortie sont

$$\frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{w}_2} = \frac{\partial f(\boldsymbol{x})}{\partial u_2}\frac{\partial u_2}{\partial \boldsymbol{w}_2} = \sigma_2'(u_2)\boldsymbol{h} \quad ; \quad \frac{\partial f(\boldsymbol{x})}{\partial b_2} = \frac{\partial f(\boldsymbol{x})}{\partial u_2}\frac{\partial u_2}{\partial b_2} = \sigma_2'(u_2)$$

Mais pour optimiser ces paramètres avec la descente de gradient stochastique, il faut en fait calculer le gradient de l'erreur en sortie $\ell(y, \hat{y})$, et donc considérer un graphe de calcul au bout duquel s'ajoute le calcul de l'erreur :

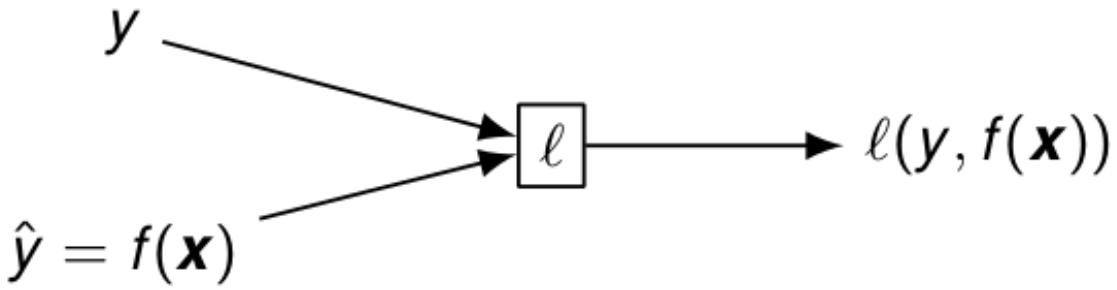


Figure 21.8: Graphe de calcul de l'erreur en sortie

Cela conduit à

$$\frac{\partial \ell(y,\hat{y})}{\partial \boldsymbol{w}_2} = \frac{\partial \ell(y,\hat{y})}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial \boldsymbol{w}_2} = \frac{\partial \ell(y,\hat{y})}{\partial \hat{y}}\sigma_2'(u_2)\boldsymbol{h}$$

$$\frac{\partial \ell(y,\hat{y})}{\partial b_2} = \frac{\partial \ell(y,\hat{y})}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial b_2} = \frac{\partial \ell(y,\hat{y})}{\partial \hat{y}} \sigma_2'(u_2)$$

avec des calculs redondants qui peuvent être économisés si $\frac{\partial \ell(y,\hat{y})}{\partial b_2}$ est calculé en premier par exemple.

Pour les gradients par rapport aux paramètres de la couche cachée :

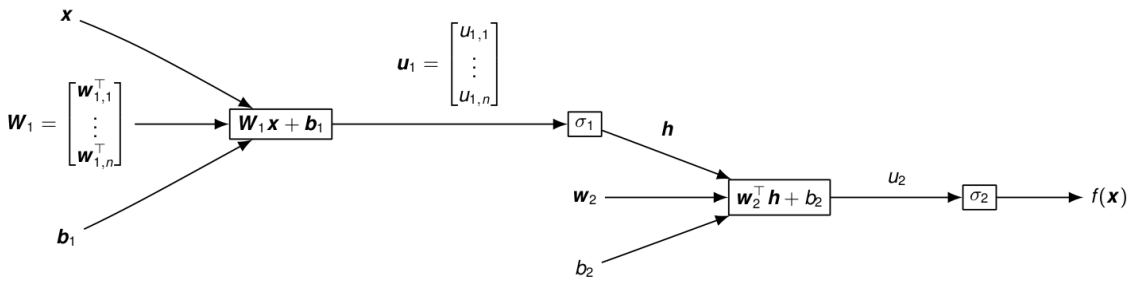


Figure 21.9: Graphe de calcul d'un MLP

le seul chemin de $\boldsymbol{w}_{1,k}$ à $f(\boldsymbol{x})$ passe uniquement par $u_{1,k}$ et h_k :

$$\frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{w}_{1,k}} = \frac{\partial f(\boldsymbol{x})}{\partial u_2} \frac{\partial u_2}{\partial h_k} \frac{\partial h_k}{\partial u_{1,k}} \frac{\partial u_{1,k}}{\partial \boldsymbol{w}_{1,k}} = \sigma_2'(u_2) w_{2,k} \sigma_1'(u_{1,k}) \boldsymbol{x}$$

$$\frac{\partial f(\boldsymbol{x})}{\partial b_{1,k}} = \frac{\partial f(\boldsymbol{x})}{\partial u_2} \frac{\partial u_2}{\partial h_k} \frac{\partial h_k}{\partial u_{1,k}} \frac{\partial u_{1,k}}{\partial b_{1,k}} = \sigma_2'(u_2) w_{2,k} \sigma_1'(u_{1,k})$$

Et pour l'erreur en sortie $\ell(y,\hat{y})$:

$$\frac{\partial \ell(y,\hat{y})}{\partial b_{1,k}} = \frac{\partial \ell(y,\hat{y})}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial b_{1,k}} = \frac{\partial \ell(y,\hat{y})}{\partial \hat{y}} \sigma_2'(u_2) w_{2,k} \sigma_1'(u_{1,k})$$

$$\frac{\partial \ell(y,\hat{y})}{\partial \boldsymbol{w}_{1,k}} = \frac{\partial \ell(y,\hat{y})}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \boldsymbol{w}_{1,k}} = \frac{\partial \ell(y,\hat{y})}{\partial \hat{y}} \sigma_2'(u_2) w_{2,k} \sigma_1'(u_{1,k}) \boldsymbol{x}$$

Ici encore, on peut remarquer que $\frac{\partial \ell(y,\hat{y})}{\partial \hat{y}} \sigma_2'(u_2)$ a déjà été calculé dans $\frac{\partial \ell(y,\hat{y})}{\partial b_2}$ au niveau de la couche supérieure. C'est toute l'idée de la **rétropropagation** qui consiste à calculer les gradients en propageant l'information de la sorite vers l'entrée du réseau.

Résumé de l'algorithme de rétropropagation du gradient

- On utilise la **descente de gradient stochastique** : de petites itérations qui minimisent l'erreur en sortie $\ell(y_i, f(\boldsymbol{x}_i))$ sur 1 exemple.
- Chaque itération demande le calcul du gradient de l'erreur par rapport aux paramètres.
- Les gradients peuvent être calculés en (retro)propageant l'information de la sortie vers l'entrée pour économiser énormément de calculs.

Donc, on boucle sur les étapes suivantes :

1. tirer un exemple $(\boldsymbol{x}_i, y_i)$;
2. calculer $f(\boldsymbol{x}_i)$ en mémorisant toutes les quantités intermédiaires ($\boldsymbol{u}_1$, $\boldsymbol{h}$, u_2, ...) *(forward pass)* ;
3. calculer récursivement les gradients $\frac{\mathrm{d}\ell(y_i, f(\boldsymbol{x}_i))}{\mathrm{d}\boldsymbol{w}}$ pour tous les paramètres $\boldsymbol{w}$ *(backward pass)* ;
4. mettre à jour $\boldsymbol{w} \leftarrow \boldsymbol{w} - \mu \frac{\mathrm{d}\ell(y_i, f(\boldsymbol{x}_i))}{\mathrm{d}\boldsymbol{w}}$.

Et l'algorithme fait plusieurs passages ainsi à travers la base d'apprentissage. Chaque passage sur l'ensemble de la base s'appelle une *époque (ou epoch)*.

21.2.4 Variantes de l'algorithme

En pratique, on utilise plutôt des « mini-batch » aléatoires $I \subset \{1, \dots m\}$ ($|I| \approx 10$ ou 100) pour optimiser chaque itération sur un petit ensemble de points au lieu d'un seul point :

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \mu \frac{1}{|I|} \sum_{i \in I} \frac{\mathrm{d}\ell(y_i, f(\boldsymbol{x}_i))}{\mathrm{d}\boldsymbol{w}}$$

La taux d'apprentissage μ peut aussi être modifié au cours de l'apprentissage, typiquement pour démarrer avec une grande valeur μ_0 qui décroît au cours du temps $0 \leq t \leq \tau$ jusqu'à μ_τ :

$$\mu \leftarrow (1 - \frac{t}{\tau})\mu_0 + \frac{t}{\tau}\mu_\tau$$

On peut aussi ajouter de l'inertie dans les mises à jour (du *« momentum »*) avec $\alpha \in [0, 1)$:

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \mu \frac{1}{|I|} \sum_{i \in I} \frac{\mathrm{d}\ell(y_i, f(\boldsymbol{x}_i))}{\mathrm{d}\boldsymbol{w}}$$

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \boldsymbol{v}$$

Cela permet de limiter les oscillations contraires entre les mises à jour successives.

En résumé, il existe beaucoup d'heuristiques différentes pour adapter μ ou implémenter l'inertie telles que AdaGrad, RMSprop, Adam...

21.3 Réseaux de neurones pour la classification

Pour la régression, l'algorithme ci-dessus s'applique directement à la fonction de perte quadratique. Mais pour la classification, la perte 0-1 n'est pas dérivable et ne peut donc pas être utilisée directement.

À la place, nous utilisons un encodage 0-1 des étiquettes : par exemple pour $C = 3$ catégories,

$$\boldsymbol{y} = \begin{bmatrix}1\\0\\0\end{bmatrix}, \ si\ y = 1, \quad \boldsymbol{y} = \begin{bmatrix}0\\1\\0\end{bmatrix}, \ si\ y = 2, \quad \boldsymbol{y} = \begin{bmatrix}0\\0\\1\end{bmatrix}, \ si\ y = 3$$

et nous construisons un réseau à C sorties, ou à sortie vectorielle

$$\hat{\boldsymbol{y}} = \boldsymbol{f}(\boldsymbol{x}) = \begin{bmatrix}\hat{y}_1\\\hat{y}_2\\\hat{y}_3\end{bmatrix} = \mathbf{softmax}(\boldsymbol{u})$$

avec $\boldsymbol{u} \in \mathbb{R}^C$ contenant un score réel pour chaque catégorie et

$$\hat{y}_k = \mathbf{softmax}_k(\boldsymbol{u}) = \frac{e^{u_k}}{\sum_{j=1}^{Q} e^{u_j}}$$

La fonction **softmax** permet de garantir que les sorties $\hat{y}_k$ sont dans $[0, 1]$ et normalisées telles que $\sum_{k=1}^{Q} \hat{y}_k = 1$. Ainsi, elles estiment l'encodage 0-1 $\boldsymbol{y}$ et peuvent être interprétées comme une loi de probabilité discrète sur les catégories : $\hat{y}_k = \widehat{P}(Y = k|X = x)$.

La **catégorie prédite** est ensuite donnée par la probabilité maximale :

$$\arg \max_{k \in \{1,\ldots,C\}} \hat{y}_k$$

Dans ce cas, l'**apprentissage** se fait typiquement en minimisant l'**entropie croisée** :

$$\ell(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\sum_{k=1}^{Q} y_k \log \hat{y}_k = -\log \hat{y}_y$$

sachant que minimiser $\ell(\boldsymbol{y}, \hat{\boldsymbol{y}})$ revient à maximiser $\hat{y}_y = \widehat{P}(Y = y|X = x)$.

Le calcul des dérivées s'effectue ainsi : avec

$$\ell(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\log \hat{y}_y = -u_y + \log \sum_{j=1}^{Q} e^{u_j}$$

on a

$$\frac{\partial \ell(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial u_k} = -\mathbf{1}(k = y) + \frac{e^{u_k}}{\sum_{j=1}^{Q} e^{u_j}} = \hat{y}_k - \mathbf{1}(k = y) = \hat{y}_k - y_k$$

et donc

$$\frac{\partial \ell(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial u_k} \begin{cases} \leq 0, & si\ k = y \\ \geq 0, & si\ k \neq y \\ = 0, & si\ \hat{y}_k = \mathbf{1}(k = y) \end{cases}$$

(où le dernier cas pour $\hat{y}_k = \mathbf{1}(k = y)$ ne peut jamais arriver).

partie III

Apprentissage non supervisé

L'apprentissage non supervisé se place dans le contexte où les données disponibles ne sont pas étiquetées et où l'objectif est en général d'extraire de l'information sur les données présentes dans la base d'apprentissage et non de construire un modèle de prédiction capable de généraliser comme dans le cas supervisé.

22 Bases de l'apprentissage non supervisé

L'apprentissage non supervisé se place dans le contexte où les données disponibles ne sont **pas étiquetées**.

À partir de telles données, il ne s'agit donc pas d'apprendre un modèle de prédiction capable de généraliser, c'est-à-dire de répondre correctement à de nouvelles questions sur la base d'exemples de bonnes réponses. Ici, il s'agit plutôt d'extraire de l'information sur les objets représentés par les données disponibles.

Différents problèmes d'apprentissage non supervisés peuvent se présenter :

- le clustering cherche à regrouper les données en un certain nombre de groupes homogènes, permettant d'identifier des catégories d'objets ;
- l'estimation de densité cherche à modéliser le processus qui génère les données, soit pour pouvoir en générer de nouvelles, soit pour pouvoir les expliquer, ou les résumer.

22.1 Données disponibles

Les données considérées correspondent à une représentation numérique des objets sur les lesquels on cherche à apprendre quelque chose, que ce soit des images, des sons, des textes, ou plus simplement des tableaux de nombres. De manière générale, toutes ces données peuvent être représentées par des vecteurs d'une certaine dimension d :

$$\boldsymbol{x} \in \mathcal{X} \subset \mathbb{R}^d$$

Le choix des composantes de $\boldsymbol{x}$ (les descripteurs) est très lié à l'application visée. Il est aussi possible d'utilier des techniques de *sélection de variables* ou de *réduction de dimension* pour construire des représentation $\boldsymbol{x}$ plus efficaces.

En apprentissage non supervisé, la **base d'apprentissage** contient des données non étiquetées, c'est-à-dire des exemples pour lesquels seul $\boldsymbol{x} \in \mathcal{X}$ est connu :

$$S = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m\}$$

Ici, m représente la taille de la base d'apprentissage, ou encore, nombre d'exemples disponibles. En apprentissage, la base d'apprentissage S contient (souvent) les seules informations disponibles pour résoudre le problème.

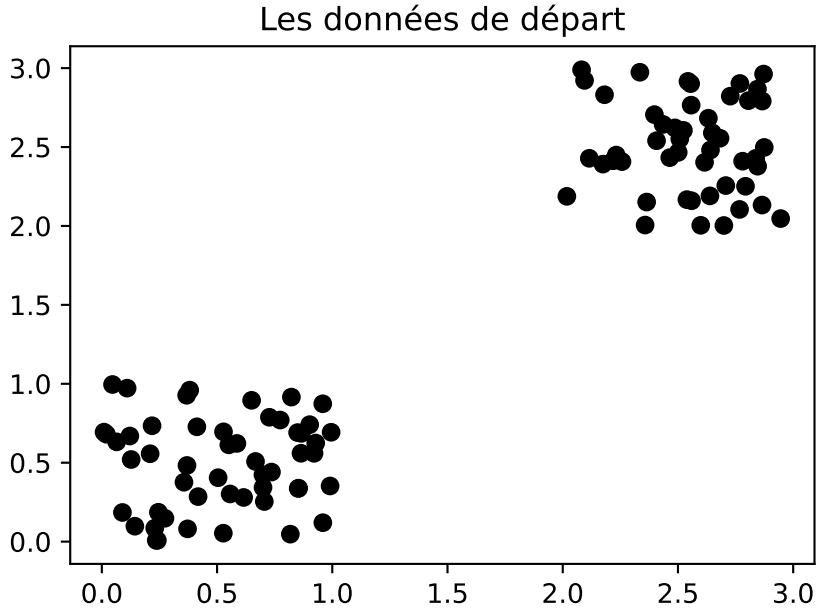
22.2 Clustering

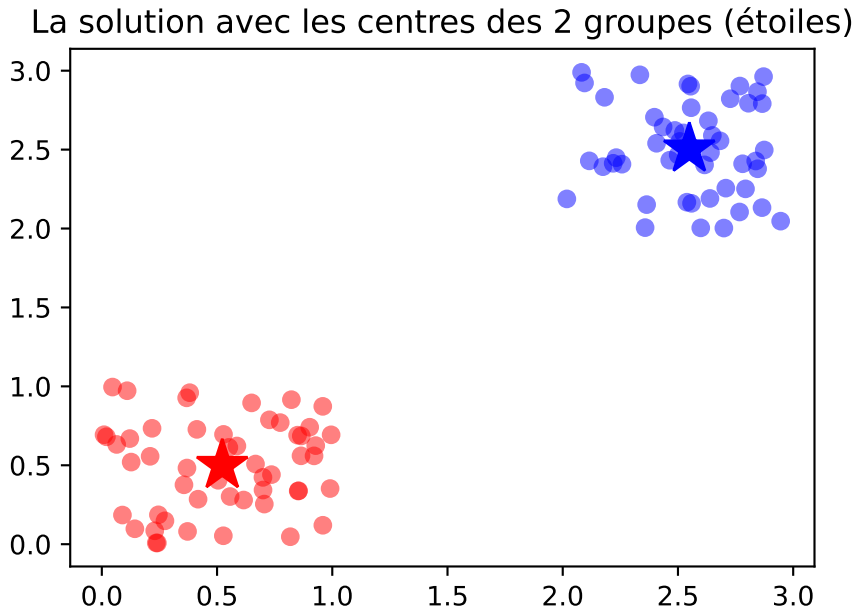
À partir d'un jeu de données S non étiquetées, un algorithme de clustering construit une partition de cet ensemble de données : une division de S en groupes G_k telle que $\bigcup_{k=1}^{K} G_k = S$. La difficulté ici est bien évidemment de savoir comment regrouper les exemples.

Une première approche consiste à chercher les K groupes les plus homogènes possibles, c'est-à-dire pour lesquels les données paraissent le plus semblable à l'intérieur de chaque groupe. L'algorithme K-means est une méthode simple qui suit cette approche.

Exemple en 2D

```
X = np.random.rand(100,2)
y = np.random.rand(100) > 0.5
X[y,:] += np.array([2,2])
plt.plot(X[:,0], X[:,1], "ok")
plt.title("Les données de départ")

plt.figure()
plt.plot(X[y,0], X[y,1], 'ob', alpha=0.5)
plt.plot(X[~y,0], X[~y,1], 'or', alpha=0.5)
mu1 = np.mean(X[y,:], axis=0)
mu2 = np.mean(X[~y,:], axis=0)
plt.plot(mu1[0], mu1[1], "*b", markersize=20)
plt.plot(mu2[0], mu2[1], "*r", markersize=20)
t=plt.title("La solution avec les centres des 2 groupes (étoiles)")
```



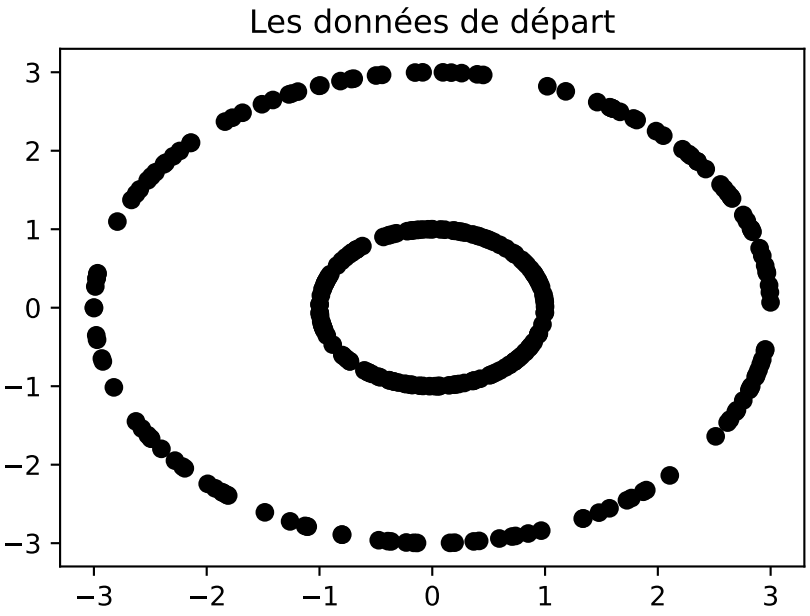


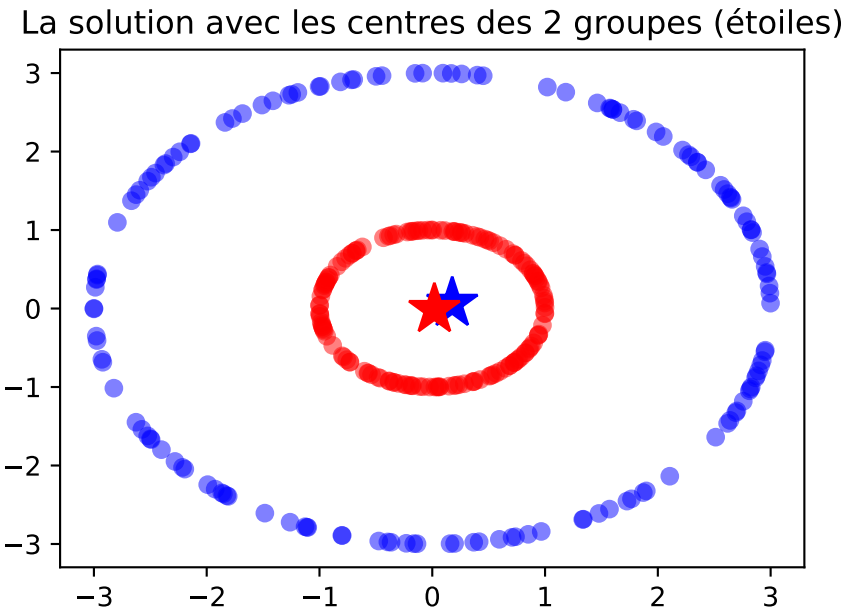
Ici les données de chaque groupe sont toutes proches les unes des autres, et donc aussi plus proches de leur centre que de celui de l'autre groupe.

Une autre approche consiste à cherches des groupes d'exemples fortement connectés. L'idée de similarité est ici toujours présente, mais deux exemples d'un même groupe peuvent être très éloignés s'il existe un « chemin d'exemples » qui permet d'aller d'un à l'autre en sautant de proche en proche. La méthode de clustering spectral permet de construire de tels groupes efficacement.

Exemple en 2D avec des groupes non homogènes

```
X = np.random.randn(300,2)
y = np.random.rand(300) > 0.5
X[y,:] *= np.tile(3 / np.linalg.norm(X[y,:], axis=1), (2, 1)).T
X[~y,:] *= np.tile(1 / np.linalg.norm(X[~y,:], axis=1), (2, 1)).T
plt.plot(X[:,0], X[:,1], "ok")
plt.title("Les données de départ")

plt.figure()
plt.plot(X[y,0], X[y,1], 'ob', alpha=0.5)
plt.plot(X[~y,0], X[~y,1], 'or', alpha=0.5)
mu1 = np.mean(X[y,:], axis=0)
mu2 = np.mean(X[~y,:], axis=0)
plt.plot(mu1[0], mu1[1], "*b", markersize=20)
plt.plot(mu2[0], mu2[1], "*r", markersize=20)
t=plt.title("La solution avec les centres des 2 groupes (étoiles)")
```





Il est évident ici que représenter les deux groupes avec leur centre ne permet pas de les différencier. De même, les deux groupes (en particulier le bleu) contiennent des données très différentes (assez éloignées dans l'espace). En revanche, les données sont assez bien connectées à l'intérieur de chaque groupe : il exste un chemin n'incluant que des petits écarts entre deux points de la même couleur.

22.3 Estimation de densité

L'estimation de densité cherche à apprendre la distribution des données dans l'espace de représentation $\mathcal{X}$.

23 Algorithme K-means

L'algortihme K-means est un algorithme de clustering. À partir d'un ensemble d'exemples non étiquetés

$$S = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m\}$$

il va trouver une répartition de ceux-ci **en** K **groupes** G_k, $k = 1, \ldots, K$, mais aussi les **prototypes** de ces groupes, c'est-à-dire des exemples (artificiels) et représentatifs des données de chaque groupe. Ces prototypes correspondent en fait simplement aux **centres des groupes** $\boldsymbol{c}_k$.

23.1 Détails de l'algorithme

1. Initialiser les K centres c_k des groupes (typiquement aléatoirement)
2. Classer les exemples avec

$$y_i = \arg \min_{k \in \{1,\ldots,K\}} \|\boldsymbol{x}_i - \boldsymbol{c}_k\| \tag{23.1}$$

$$G_k = \{i : y_i = k\}$$

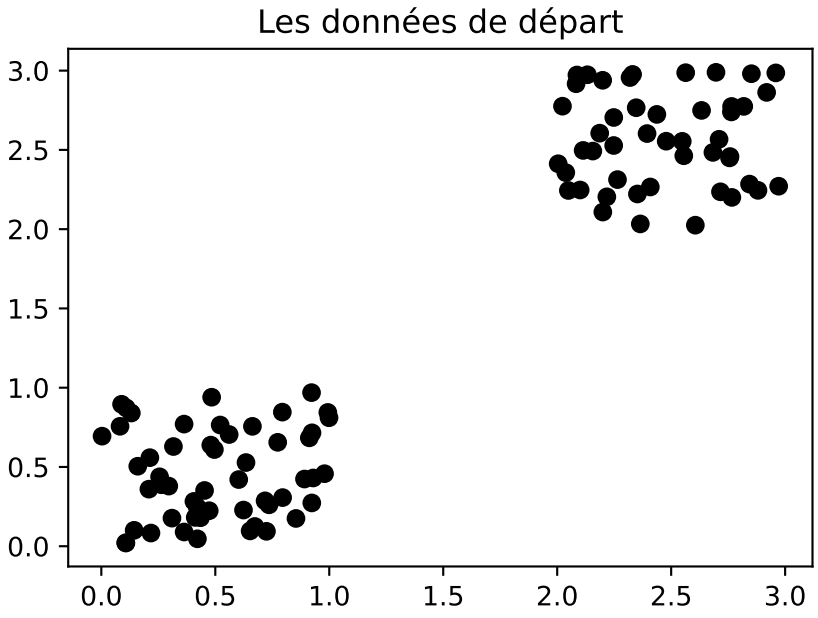
3. Mettre à jour les centres des groupes avec

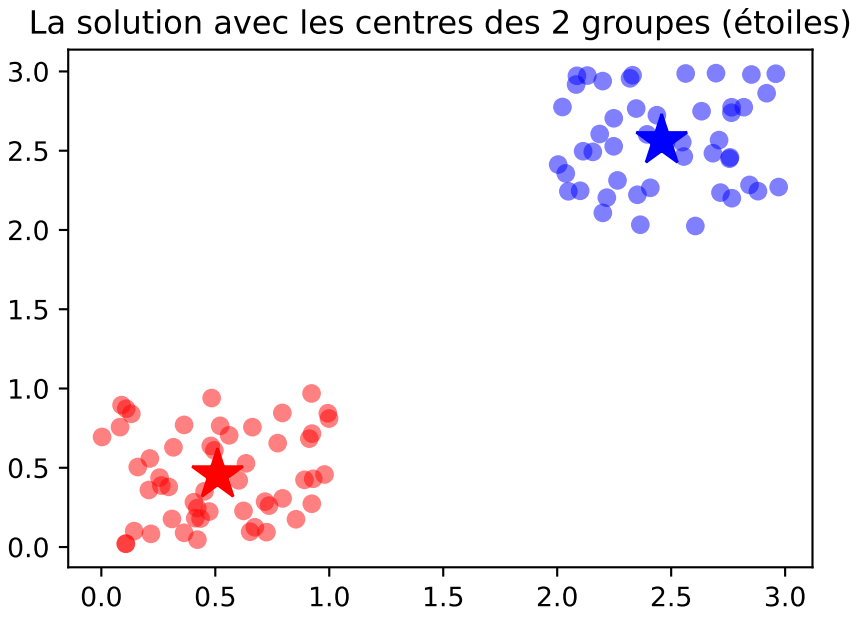
$$\boldsymbol{c}_k = \frac{1}{|G_k|} \sum_{\boldsymbol{x}_i \in G_k} \boldsymbol{x}_i \tag{23.2}$$

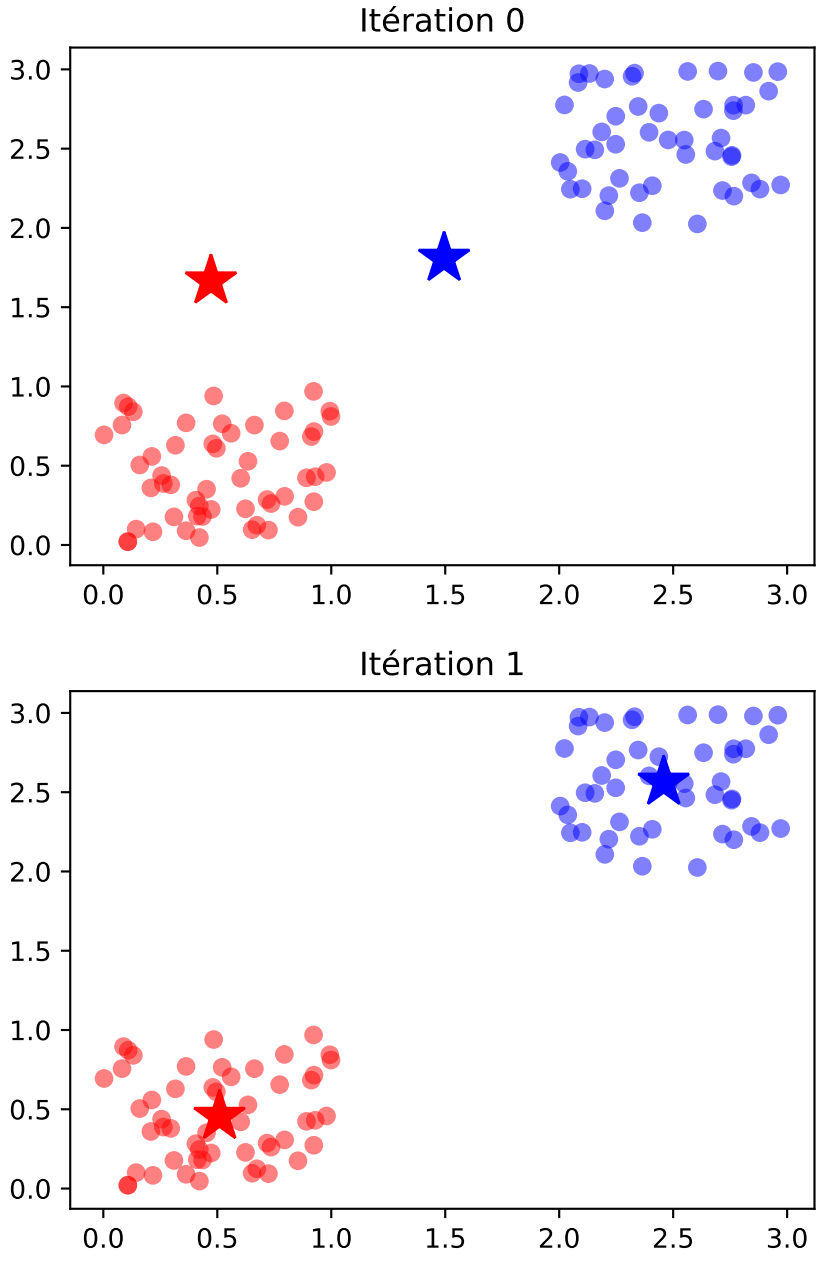
   où $|G_k|$ désigne le cardinal du groupe G_k (le nombre d'exemples affectés à ce groupe).
4. Répéter depuis 2. jusqu'à convergence.

En python

```
def kmeans(X, K):
    m, d = X.shape
    C = np.min(X) + (np.random.rand(K,d)  * (np.max(X)-np.min(X)))

    Cprecedent = np.zeros(C.shape)
    Distances = np.zeros((m,K))
    history = []
    while np.linalg.norm(C - Cprecedent) > 1e-6:
        Cprecedent = np.copy(C)

        # Classer les données:
        for k in range(K):
            Distances[:,k] = np.linalg.norm(X-C[k,:], axis=1)
        y = np.argmin(Distances, axis=1)

        history.append([np.copy(y),np.copy(C)])

        # Mettre à jour les centres
        for k in range(K):
            C[k,:] = np.mean(X[y==k,:], axis=0)
    return y, C, history

#### Test ###
X = np.random.rand(100,2)
y = np.random.rand(100) > 0.5
X[y,:] += np.array([2,2])
plt.plot(X[:,0], X[:,1], "ok")
plt.title("Les données de départ")

ypred, C, history = kmeans(X, 2)
plt.figure()
plt.plot(X[ypred==0,0], X[ypred==0,1], 'ob', alpha=0.5)
plt.plot(X[ypred==1,0], X[ypred==1,1], 'or', alpha=0.5)
plt.plot(C[0,0], C[0,1], "*b", markersize=20)
plt.plot(C[1,0], C[1,1], "*r", markersize=20)
plt.title("La solution avec les centres des 2 groupes (étoiles)")

for t in range(len(history)):
    ypred, C = history[t]
    plt.figure()
    plt.plot(X[ypred==0,0], X[ypred==0,1], 'ob', alpha=0.5)
    plt.plot(X[ypred==1,0], X[ypred==1,1], 'or', alpha=0.5)
    plt.plot(C[0,0], C[0,1], "*b", markersize=20)
    plt.plot(C[1,0], C[1,1], "*r", markersize=20)
    plt.title("Itération " + str(t))
```


Les données de départ
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0


La solution avec les centres des 2 groupes (étoiles)
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0



Pour être complet, l'algorithme devrait inclure la gestion des cas où aucun point n'est affecté à un groupe. Typiquement, il est possible de réinitialiser le centre du groupe aléatoirement ou de supprimer complétement le groupe du reste de la procédure.

23.2 Critère optimisé par K-means

L'algorithme K-means est une méthode d'optimisation alternée pour minimiser le critère

$$J((\boldsymbol{c}_k)_{k=1}^K, (G_k)_{k=1}^K) = \sum_{k=1}^{K} \sum_{i \in G_k} \|\boldsymbol{x}_i - \boldsymbol{c}_k\|^2$$

qui mesure la somme de toutes les distances des points $\boldsymbol{x}_i$ au centre $\boldsymbol{c}_k$ du groupe G_k auquel il est affecté.

L'algorithme alterne en fait entre

1. la minimisation de cette fonction par rapport aux groupes G_k (à centres fixés),
2. puis la minimisation par rapport aux centres $\boldsymbol{c}_k$ (à groupes G_k fixés).

Il est aisé de voir que, pour minimiser ce critère, il est nécessaire d'affecter chaque point $\boldsymbol{x}_i$ au centre $\boldsymbol{c}_k$ le plus proche, ce qui donne la formule de classification Équation 23.1.

Pour minimiser par rapport aux centres, on remarque que le critère est une somme de K termes, chacun n'impliquant qu'un centre $\boldsymbol{c}_k$ de manière indépendante. Il suffit donc de résoudre les K sous-problèmes d'optimisation

$$\min_{\boldsymbol{c}_k} J_k(\boldsymbol{c}_k), \qquad J_k(\boldsymbol{c}_k) = \sum_{i \in G_k} \|\boldsymbol{x}_i - \boldsymbol{c}_k\|^2$$

Le gradient de l'objectif $J_k(\boldsymbol{c}_k)$ est ici

$$\frac{\mathrm{d}J_k(\boldsymbol{c}_k)}{\mathrm{d}\boldsymbol{c}_k} = -2 \sum_{i \in G_k} (\boldsymbol{x}_i - \boldsymbol{c}_k)$$

et la solution est obtenue lorsque ce gradient est nul, ce qui donne $|G_k|\boldsymbol{c}_k = \sum_{i \in G_k} \boldsymbol{x}_i$ et la formule Équation 23.2 qui calcule $\boldsymbol{c}_k$ comme la moyenne des points.

On peut montrer qu'à chaque itération de l'algorithme, le critère $J((\boldsymbol{c}_k)_{k=1}^K, (G_k)_{k=1}^K)$ décroît de manière monotone. Cependant, aucune garantie n'est donnée sur l'optimalité de la solution, car la solution converge vers un point qui dépend fortement de l'initialisation.

Exemple difficile pour K-means

```
def J(C, y):
    cost = 0
    for k in range(len(C)):
        cost += np.sum( ( X[y==k,:] - C[k,:] )**2 )
    return cost

# Données en grande dimension
m = 200
d = 20
K = 5
X = np.random.randn(m,d)
y = np.random.randint(0,K,size=m)
Copt = np.zeros((K,d))
for k in range(K):
    X[y==k,:] += 3 * k * np.ones(d)/np.sqrt(d)
    Copt[k,:] = np.mean(X[y==k,:],0)

print("Coût optimal =", J(Copt,y))

ypred, C, _ = kmeans(X, K)
print("Coût de la solution K-means =", J(C,ypred))
```

```
Coût optimal = 3835.531980047208
Coût de la solution K-means = 4477.61804002767
```

23.3 Amélioration par multiples initialisations

Pour améliorer la qualité de la solution, une approche simple consiste à répéter l'algorithme N fois à partir d'initialisation aléatoires et différentes pour ne conserver au final que la solution associée à la plus petite valeur de $J((\boldsymbol{c}_k)_{k=1}^K, (G_k)_{k=1}^K)$.

Cependant, le nombre d'initialisations requis croît rapidement avec la dimension du problème, comme prédit par la malédiction de la dimension.

23.4 K-means++ : initialisation dispersée

L'algorithme K-means++, proposé par Arthur et Vassilvitskii (2007), fournit une initialisation particulièrement efficace pour K-means. Celle-ci repose sur l'idée que les résultats de K-means sont meilleurs lorsque les centres initiaux sont bien dispersés dans l'espace.

K-means++ choisi les centres initiaux parmi les m points $\boldsymbol{x}_i$ de la base d'apprentissage ainsi :

1. Tirer $\boldsymbol{c}_1$ aléatoirement de manière uniforme parmi tous les exemples $\boldsymbol{x}_i$, $i = 1, \ldots, m$.
2. Pour $k = 2, \ldots, K$:

- Pour chaque point $\boldsymbol{x}_i$, calculer la distance au centre le plus proche

$$D_i = \min_{1 \leq j < k} \|\boldsymbol{x}_i - \boldsymbol{c}_k\|$$

- Choisir $\boldsymbol{c}_k$ aléatoirement selon la loi de probabilité

$$P(\boldsymbol{c}_k = \boldsymbol{x}_i) = \frac{D_i^2}{\sum_i D_i^2}$$

24 Clustering spectral

Le clustering cherche à regrouper **en** K **groupes** G_k, $k = 1, \ldots, K$ des exemples non étiquetés

$$S = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m\}$$

Contrairement aux méthodes basées sur une représentation explicite des groupes, comme K-means avec des centres, les méthodes de clustering spectrales s'appuient sur la connectivité entre exemples d'un même groupe, indépendamment de la forme du groupe. Cette connectivité est représentée par un graphe.

24.1 Construction du graphe

Dans un premier temps, il s'agit de contruire un graphe dans lequel

- chaque sommet correspond à un exemple $\boldsymbol{x}_i$,
- chaque arête représente un lien de similarité entre les deux points $\boldsymbol{x}_i$ et $\boldsymbol{x}_j$ connectés et la pondération du l'arête mesure le niveau de ce lien.

Tous les liens peuvent être représentés au travers de la matrice d'adjacence $\boldsymbol{A} \in \mathbb{R}^{m \times m}$ où A_{ij} mesure l'affinité entre $\boldsymbol{x}_i$ et $\boldsymbol{x}_j$ et respecte $A_{ij} = A_{ji}$, $A_{ij} \geq 0$ et $A_{ii} = 0$.

Différents types de graphes peuvent être considérés avec différentes mesures d'affinité/similarité :

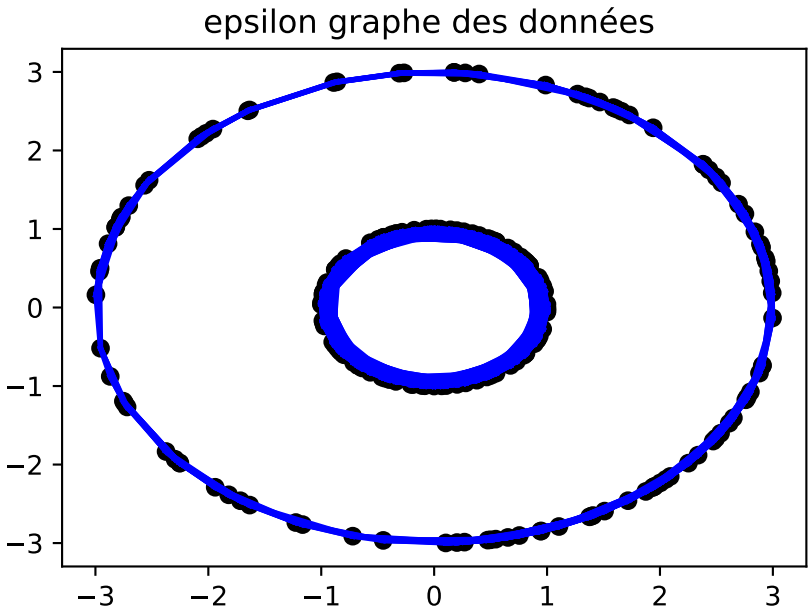
- *kppv* : $A_{ij} = 1$ si $\boldsymbol{x}_j$ est un des plus proches voisins de $\boldsymbol{x}_i$ ou l'inverse, 0 sinon ;
- *Kppv-bis* : $A_{ij} = 1$ si $\boldsymbol{x}_j$ est un des plus proches voisins de $\boldsymbol{x}_i$ *et* l'inverse, 0 sinon ;
- ϵ*-graphe* : $A_{ij} = 1$ si $\|\boldsymbol{x}_i - \boldsymbol{x}_j\| \leq \epsilon$, 0 sinon ;
- *RBF gaussien* : $A_{ij} = \exp\left(\frac{-\|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2}{2\sigma^2}\right)$.

Une fois le graphe construit, le but est de détecter les composantes connectées du graphe.

Exemple

```
def plot_graph(X, A, title):
    m = len(X)
    plt.figure()
    plt.plot(X[:,0], X[:,1], "ok")
    for i in range(m):
        for j in range(i):
            if A[i,j] > 0:
                plt.plot(X[[i,j],0], X[[i,j],1], "-b", linewidth=2*A[i,j])
    plt.title(title + " graphe des données ")
```

```
m=200
X = np.random.randn(m,2)
y = np.random.rand(m) > 0.5
X[y,:] *= np.tile(3 / np.linalg.norm(X[y,:], axis=1), (2, 1)).T
X[~y,:] *= np.tile(1 / np.linalg.norm(X[~y,:], axis=1), (2, 1)).T

eps = 1
A = np.zeros((m,m))
for i in range(m):
    for j in range(i):
        if np.linalg.norm(X[i,:]-X[j,:]) < eps:
            A[i,j] = 1
            A[j,i] = 1

plot_graph(X, A, "epsilon")
```



Dans ce graphe, on peut identifier 2 composantes connectées correspondant aux deux groupes de données recherchées. À noter que tous les points d'un groupe ne sont pas forcément tous liés directement, mais simplement connectés par un chemin qui ne parcourt que des points du même groupe.

24.2 Algorithme

Après la construction du graphe, l'algorithme de clustering spectral va projeter les données dans un nouvel espace de représentation de petite dimension où les groupes apparaîtront de manière évidente.

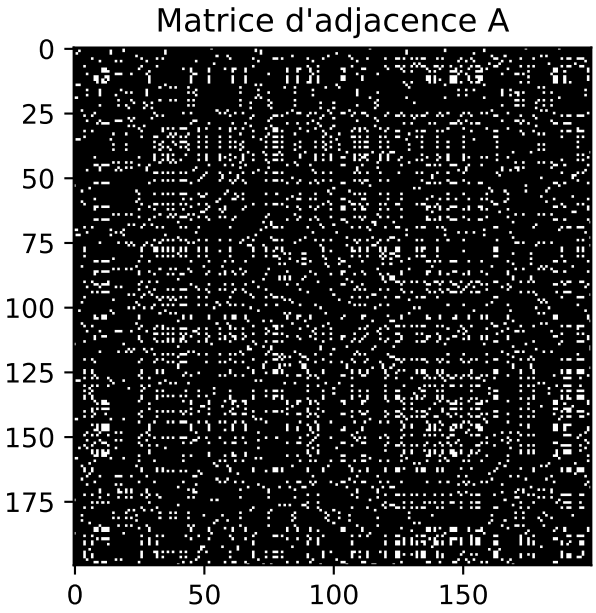
1. Construire la matrice de similarité $\boldsymbol{A}$ (matrice d'adjacence du graphe) (voir ci-dessus).
2. Calculer la matrice des degrés[1] : $\boldsymbol{D} = diag(\boldsymbol{A}\mathbf{1})$, $D_{ii} = \sum_{j=1}^{m} A_{ij}$.
3. Calculer le laplacien du graphe : $\boldsymbol{L} = \boldsymbol{D} - \boldsymbol{A}$.[2]
4. Calculer les K vecteurs propres $\boldsymbol{u}_k$ de $\boldsymbol{L}$ avec les plus petites valeurs propres.
5. Considérer les lignes $\boldsymbol{v}_i^T$ de la matrice $\boldsymbol{U} = [\boldsymbol{u}_1, \ldots, \boldsymbol{u}_K] = \begin{bmatrix} \boldsymbol{v}_1^T \\ \vdots \\ \boldsymbol{v}_m^T \end{bmatrix} \in \mathbb{R}^{m \times K}$
6. Appliquer K-means (par exemple) aux points $\boldsymbol{v}_i$ dans la nouvelle représentation.

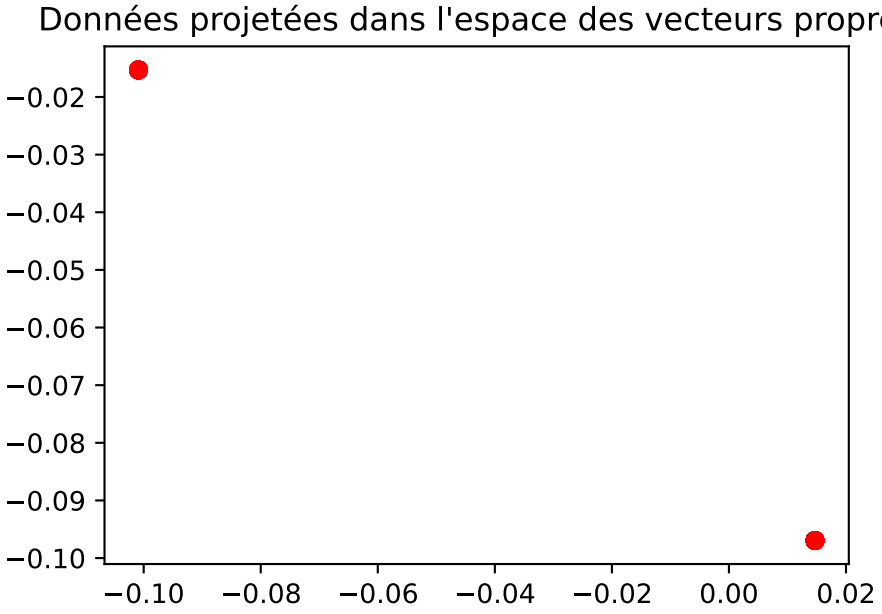
[1] Le degré d'un sommet du graphe est la somme des pondérations des arêtes connectées à ce sommet. La matrice des degrés est la matrice diagonale où chaque terme sur la diagonale donne le degré d'un sommet.

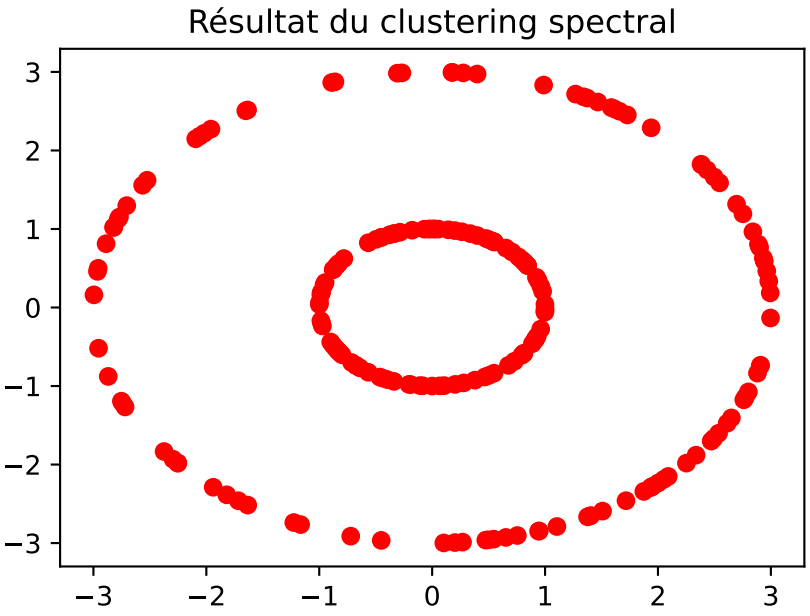
[2] Certaines variantes considèrent une version normalisée du laplacien : $\boldsymbol{L}_{rw} = \boldsymbol{D}^{-1}\boldsymbol{L}$ ou $\boldsymbol{L}_{sym} = \boldsymbol{D}^{-1/2}\boldsymbol{L}\boldsymbol{D}^{-1/2}$.

Exemple en python

```
def spectralclustering(X, A, K):
    m = len(X)
    D = np.diag(np.sum(A, axis=1))
    L = D - A
    lambdas, U = np.linalg.eigh(L)
    U = U[:,:K]
    y = kmeans(U, K)
    return y, U

# application aux données de l'exemple précédent
y, U = spectralclustering(X, A, 2)

plt.figure()
plt.imshow(A, cmap="gray")
plt.title("Matrice d'adjacence A")

plt.figure()
plt.plot(U[y==0,0], U[y==0,1], "ob")
plt.plot(U[y==1,0], U[y==1,1], "or")
plt.title("Données projetées dans l'espace des vecteurs propres")

plt.figure()
plt.plot(X[y==0,0], X[y==0,1], "ob")
plt.plot(X[y==1,0], X[y==1,1], "or")
t=plt.title("Résultat du clustering spectral")
```


Matrice d'adjacence A
0
25
50
75
100
125
150
175
0
50
100
150


Données projetées dans l'espace des vecteurs propres
−0.02
−0.03
−0.04
−0.05
−0.06
−0.07
−0.08
−0.09
−0.10
−0.10
−0.08
−0.06
−0.04
−0.02
0.00
0.02



24.3 Justification théorique

Pour comprendre pourquoi le clustering spectral fonctionne, plaçons-nous dans un cas idéal où (pour y_i l'indice du groupe de $\boldsymbol{x}_i$) :

- l'affinité de deux points est strictement nulle s'ils n'appartiennent pas au même groupe : $A_{ij} = 0$ si $y_i \neq y_j$;
- il existe un chemin entre chaque paire de points $\boldsymbol{x}_i$ et $\boldsymbol{x}_j$ appartenant au même groupe qui ne parcourt que des sommets du graphe affectés au même groupe : si $y_i = y_j$, alors il existe $n \geq 0$, $(\boldsymbol{x}_k)_{1 \leq k \leq n}$, tels que

$$y_k = y_i = y_j,\ \boldsymbol{x}_0 = \boldsymbol{x}_j,\ \boldsymbol{x}_n = \boldsymbol{x}_j,\ \textit{et}\ A_{k,k+1} > 0,\ k = 0, \dots, n-1$$

Dans ce cas, nous pouvons démontrer que $\lambda_1 = \cdots = \lambda_K = 0$ *(la multiplicité de la valeur propre 0 est égale au nombre de composantes connectées K du graphe), et les vecteurs propres correspondant sont les indicatrices des groupes G_k :* $(\boldsymbol{u}_k)_i = \mathbf{1}(\boldsymbol{x}_i \in G_k)$.

Ainsi, chaque vecteur $\boldsymbol{v}_i$ représentant $\boldsymbol{x}_i$ dans l'espace des vecteurs propres ne peut prendre qu'une valeur parmi K : le vecteur binaire avec un unique 1 en position y_i.[3] Et donc, tous les points du groupe G_k sont projetés sur le même point : $\boldsymbol{v}_i = \boldsymbol{v}_j$ si $y_i = y_j$ et le clustering devient évident.

Pour le démontrer, nous pouvons utiliser le résultat intermédiaire suivant :

[3] La valeur de $\boldsymbol{v}_i$ observée en pratique peut être différente de 1 car les vecteurs propres ne sont définis qu'à un facteur multiplicatif près.

Si un graphe est connecté (il existe un chemin entre toute paire de sommets), alors son laplacien $\boldsymbol{L}$ possède $\lambda = 0$ comme valeur propre de multiplicité 1 associée à un vecteur propre constant, $\boldsymbol{u} = \mathbf{1}$.

Preuve

Pour tout vecteur $\boldsymbol{u} \in \mathbb{R}^m$,

$$\boldsymbol{u}^\top \boldsymbol{L}\boldsymbol{u} = \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} A_{ij}(u_i - u_j)^2$$

car

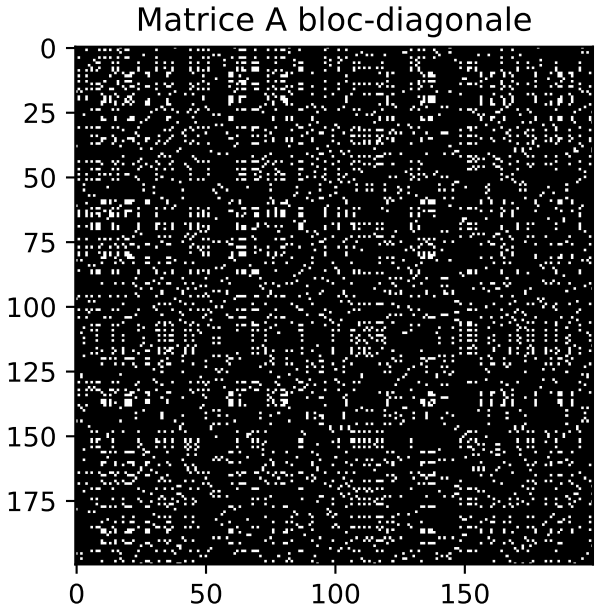
$$\begin{aligned}
\boldsymbol{u}^\top \boldsymbol{L}\boldsymbol{u} = \boldsymbol{u}^\top \boldsymbol{D}\boldsymbol{u} - \boldsymbol{u}^\top \boldsymbol{A}\boldsymbol{u} &= \sum_{i=1}^{m} D_{ii}u_i^2 - \sum_{i=1}^{m}\sum_{j=1}^{m} A_{ij}u_i u_j \qquad (car\ \boldsymbol{L} = \boldsymbol{D} - \boldsymbol{A}) \\
&= \frac{1}{2}\sum_{i=1}^{m} D_{ii}u_i^2 - \sum_{i=1}^{m}\sum_{j=1}^{m} A_{ij}u_i u_j + \frac{1}{2}\sum_{j=1}^{m} D_{jj}u_j^2 \\
&= \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} A_{ij}u_i^2 - \sum_{i=1}^{m}\sum_{j=1}^{m} A_{ij}u_i u_j + \frac{1}{2}\sum_{j=1}^{m}\sum_{i=1}^{m} A_{ji}u_j^2 \qquad (car\ D_{ii} = \sum_{j=1}^{m} A_{ij}) \\
&= \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} A_{ij}u_i^2 - \sum_{i=1}^{m}\sum_{j=1}^{m} A_{ij}u_i u_j + \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} A_{ij}u_j^2 \qquad (car\ A_{ji} = A_{ij}) \\
&= \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} A_{ij}(u_i^2 - 2u_i u_j + u_j^2) = \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} A_{ij}(u_i - u_j)^2
\end{aligned}$$

Donc, si $\boldsymbol{u}$ est un vecteur propre associé à la valeur propre $\lambda = 0$, alors

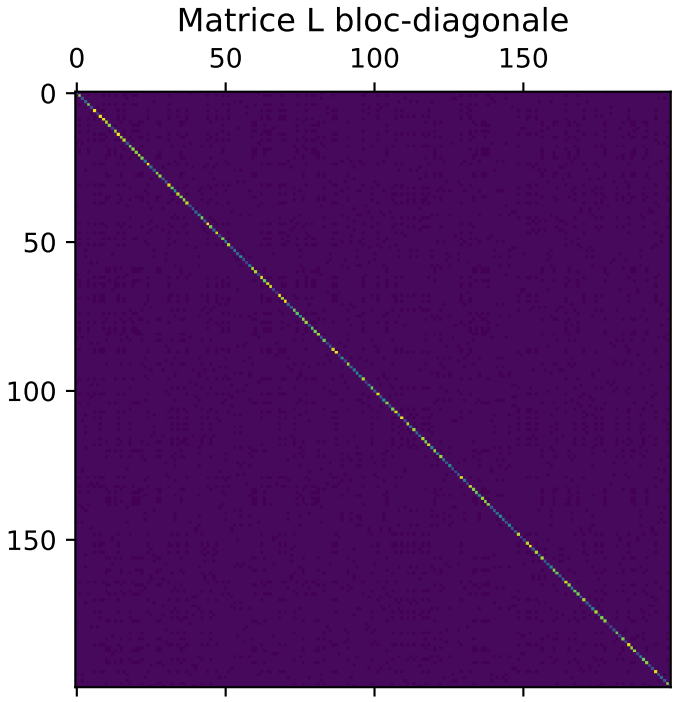
$$\boldsymbol{L}\boldsymbol{u} = \lambda\boldsymbol{u} = \mathbf{0} \qquad et \qquad \boldsymbol{u}^\top \boldsymbol{L}\boldsymbol{u} = \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} A_{ij}(u_i - u_j)^2 = \lambda\boldsymbol{u}^\top\boldsymbol{u} = 0$$

Et puisque $A_{ij} \geq 0$, on a $A_{ij} \neq 0 \Rightarrow u_i = u_j$.
Donc $u_i = u_j$ si un chemin existe entre $\boldsymbol{x}_i$ et $\boldsymbol{x}_j$, ce qui implique un vecteur propre constant pour $\lambda = 0$ pour le laplacien d'un graphe connecté.

La deuxième étape consiste à déduire ce qu'il se passe lorsque le graphe contient plusieurs composantes (sous-graphes) connectées. Dans ce cas, puisque $A_{ij} = 0$ si $y_i \neq y_j$, la matrice $\boldsymbol{A}$ est bloc-diagonale avec des zéros en dehors des blocs correspondant aux groupes recherchés :



Le laplacien $\boldsymbol{L}$ est donc lui aussi bloc-diagonal :



Il est alors possible d'utiliser le résultat suivant :

> Les valeurs propres d'une matrce bloc-diagonale sont l'ensemble des valeurs propres des blocs et les vecteurs propres sont les vecteurs propres des blocs étendus avec des zéros.

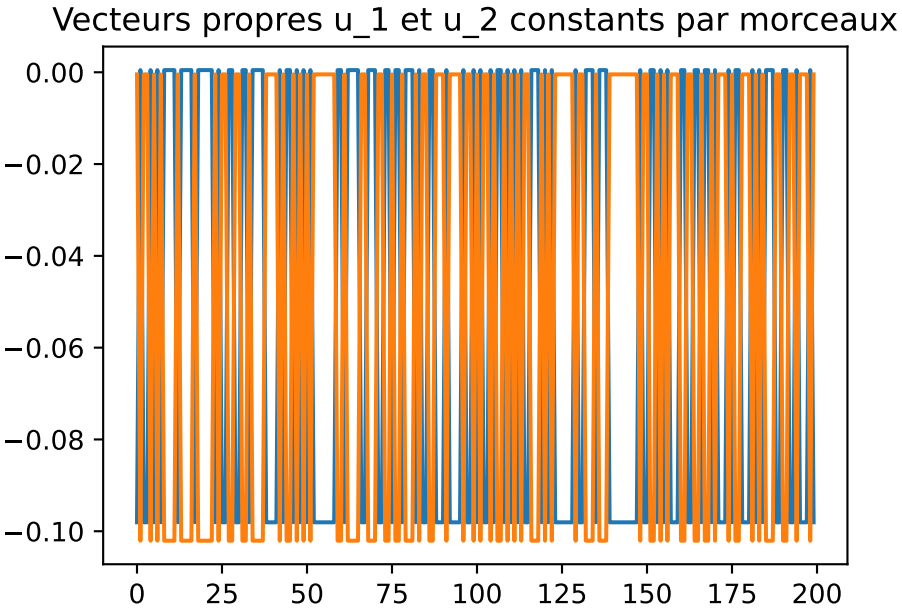
Preuve

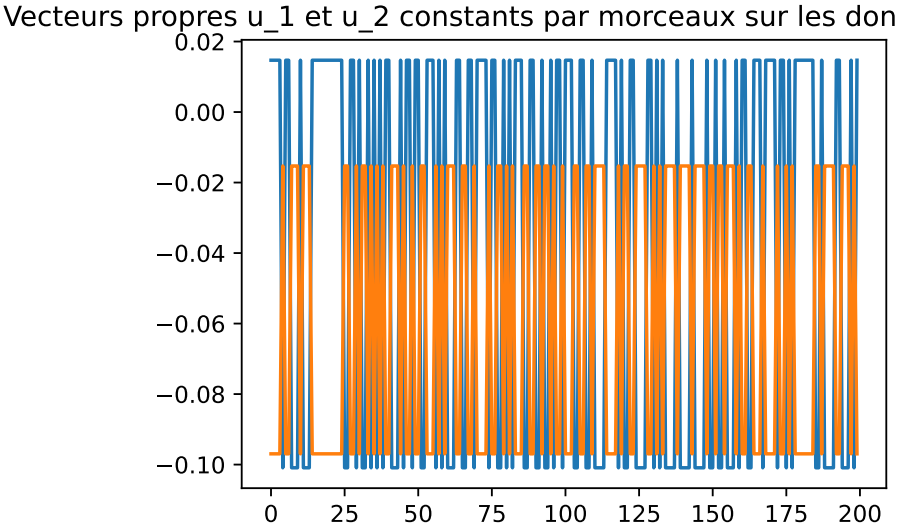
Si

$$\boldsymbol{L} = \begin{bmatrix} \boldsymbol{L}_1 & & \boldsymbol{0} \\ & \ddots & \\ \boldsymbol{0} & & \boldsymbol{L}_K \end{bmatrix}$$

et $\lambda_k \boldsymbol{u}_k = \boldsymbol{L}_k \boldsymbol{u}_k$, alors pour

$$\tilde{\boldsymbol{u}}_k = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ \boldsymbol{u}_k \\ 0 \\ \vdots \end{bmatrix} \quad \Rightarrow \quad \boldsymbol{L}\tilde{\boldsymbol{u}}_k = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ \boldsymbol{L}_k \boldsymbol{u}_k \\ 0 \\ \vdots \end{bmatrix} = \lambda_k \tilde{\boldsymbol{u}}_k$$





À noter qu'en pratique, suivant le réglage des paramètres de la construction du graphe, les matrices ne sont plus forcément bloc-diagonales (certaines connexions peuvent se créer entre les groupes), ce qui conduit à des résultats un peu moins nets que ceux-ci.

partie IV

Théorie de l'apprentissage

La théorie de l'apprentissage cherche à comprendre et modéliser les phénomènes qui entrent en jeu dans le processus d'apprentissage machine. La plupart du temps, cette théorie s'intéresse plus particulièrement à l'apprentissage supervisé.

Voici quelques exemples de questions que l'on peut se poser en théorie de l'apprentissage :

- Peut-on résoudre un problème raisonnablement par apprentissage ?
- Comment garantir les performances de l'apprentissage ?
- Comment construire des algorithmes pour optimiser ces garanties/performances ?
- Comment définir des bonnes pratiques pour l'apprentissage ?

Différentes théories ont été proposées, et nous nous concentrerons ici sur les **théories statistiques**, et en particulier sur l'apprentissage PAC et l'apprentissage PAC agnostique.

Nous verrons en partiulier comment dériver des garanties sur les performances en généralisation au travers de bornes sur l'erreur de généralisation exprimées en fonction de différentes mesures de capacité telles que le nombre de modèles, la VC dimension, ou la complexité de Rademacher.

25 Apprentissage PAC

L'apprentissage PAC pour **Probablement Approximativement Correct** a été proposé par Valiant (1984). On peut comprendre les définitions ci-dessous en faisant un parallèle avec le domaine de la complexité algorithmique, classiquement étudiée en informatique.

Dans ce domaine, on cherche à quantifier la complexité des algorithmes et des problèmes pour définir s'il est raisonnable ou non de tenter de les résoudre.

De même, nous chercherons ici à savoir s'il est raisonnable d'attaquer un « problème » par apprentissage.

En informatique, la complexité d'un problème est la plus petite complexité d'un algorithme qui résoud le problème.

En apprentissage, cette définition pose un souci : nous savons pertinemment qu'aucun algorithme ne résoud exactement un problème au sens où l'erreur de généralisation sera toujours non nulle.

La première étape consiste donc à définir ce que l'on entend par avoir un algorithme qui « résoud un problème », ce que nous traduirons par « admettre une borne PAC ».

Une autre différence concerne la définition du problème en lui-même. Ici, le « problème » que nous chercherons à résoudre sera celui de l'apprentissage au sens du choix d'un modèle parmi une classe de fonctions prédéfinie. Ainsi, différentes classes de fonctions auront des complexités différentes car il sera plus ou moins évident de choisir un bon modèle parmi elles.

Enfin, la complexité algorithmique se mesure (pour simplifier) en termes de « nombre d'opérations » minimal pour résoudre le problème. En apprentissage, le facteur limitant la qualité du résultat n'est pas la quantité de calculs mais le nombre d'exemples : un temps de calcul infini ne permet pas d'atteindre un risque nul si les données sont insufisantes. Nous mesurerons donc la « complexité en échantillon » qui compte le nombre d'exemples nécessaires pour « résoudre le problème ».

L'apprentissage PAC présenté ici se limite au cas déterministe : on suppose l'existence d'une fonction cible f^* telle que $Y = f^*(X)$. Autrement dit, le risque de Bayes $R(f_{Bayes})$ est nul. L'apprentissage PAC agnostique permet de traiter le cas plus général.

25.1 Borne PAC

Dans ce qui suit, $\mathcal{F}$ est une classe de fonctions (l'ensembe des modèles possibles) et S la base d'apprentissage.

Un algorithme $\mathcal{A} : (\mathcal{F}, S) \mapsto \hat{f}$ admet une borne PAC si pour toute fonction cible f^* et toute loi de probabilité P de (X, Y) telle que $Y = f^*(X)$, pour tout $\delta \in (0, 1)$,

$$P^m \left\{ R(\hat{f}) \leq \epsilon \right\} \geq 1 - \delta$$

où $P^m\{A\}$ fait référence à la probabilité de tirer un jeu de m données telles que A.

Cette borne garantie que l'on ne résoud le problème que de manière **probablement approximativement correcte** :

- le modèle est Approximativement Correct : son risque est faible, inférieur à ϵ, (car on ne peut pas s'attendre à un modèle qui donne des prédictions parfaites) ;
- la borne n'est que probablement valide, avec une probabilité d'au moins $1 - \delta$ (car on ne peut pas être certain à 100% de quoi que ce soit sur le risque qui reste impossible à calculer).

25.2 Complexité en échantillon (*sample complexity*)

La complexité en échantillon $m(\epsilon, \delta)$ d'un algorithme est **le plus petit m tel que la borne PAC ci-dessus soit valide**.

Cette complexité nous indique le nombre d'exemples nécessaire pour pouvoir garantir que l'algorithme résolve le problème au sens de la borne PAC ci-dessus.

25.3 Apprenabilité

Un ensemble de fonctions $\mathcal{F}$ est **apprenable** si il existe un algorithme $\mathcal{A} : (\mathcal{F}, S) \mapsto \hat{f}$ tel que **la complexité en échantillon $m(\epsilon, \delta)$ est polynomiale en $1/\epsilon$ et $1/\delta$**.

Autrement dit, un ensemble de modèles est considéré apprenable si le nombre d'exemples nécessaires à son apprentissage n'explose pas exponentiellement lorsque la performance désirée augmente ou lorsque que la confiance souhaitée dans les garanties théoriques augmente.

> Exemple de la classification dans le cas réalisable avec une classe finie
>
> Nous nous plaçons ici dans le cas réalisable où le modèle optimal appartient à l'ensemble des modèles possibles pour l'algorithme : $f^* \in \mathcal{F}$.
> Dans ce cas, la complexité en échantillon de l'algorithme ERM (qui minimise le risque empirique),
>
> $$\hat{f} = \arg\min_{f \in \mathcal{F}} R_{emp}(f)$$

est bornée par

$$m(\epsilon, \delta) = \frac{\log |\mathcal{F}| + \log \frac{1}{\delta}}{\epsilon}$$

où $|\mathcal{F}|$ est le nombre de fonctions composant $\mathcal{F}$.
Ainsi, dans le cas réalisable, $m(\epsilon, \delta)$ est linéaire en $1/\epsilon$ et logarithmique en $1/\delta$, et donc tout ensemble de classifieurs $\mathcal{F}$ fini (avec un nombre fini de modèles) est **apprenable**.
Pour démontrer cela, notons que dans le cas réalisable, $f^* \in \mathcal{F}$, et déterminite, $Y = f^*(X)$, le risque empirique du modèle optimal est $R_{emp}(f^*) = 0$. Pour l'algorithme ERM qui conduit à la plus petite erreur d'apprentissage, $R_{emp}(\hat{f}) \leq R_{emp}(f^*)$, et cela signifie que

$$R_{emp}(\hat{f}) = 0.$$

est toujours vérifiée. Ainsi,

$$P^m\left\{R(\hat{f}) \leq \epsilon\right\} = 1 - P^m\left\{R(\hat{f}) > \epsilon\right\} = 1 - P^m\left\{R_{emp}(\hat{f}) = 0 \; et \; R(\hat{f}) > \epsilon\right\}$$

En utilisant $(A \Rightarrow B) \Rightarrow P(A) \leq P(B)$ et $P(C, D) \leq P(C|D)$:

$$\begin{aligned} P^m\left\{R_{emp}(\hat{f}) = 0 \; et \; R(\hat{f}) > \epsilon\right\} &\leq P^m\left\{\exists f \in \mathcal{F}, \; R_{emp}(f) = 0 \; et \; R(f) > \epsilon\right\} \\ &= P^m\left\{\bigcup_{f \in \mathcal{F}} R_{emp}(f) = 0 \; et \; R(f) > \epsilon\right\} \\ &\leq \sum_{f \in \mathcal{F}} P^m\left\{R_{emp}(f) = 0 \; et \; R(f) > \epsilon\right\} \\ &\leq \sum_{f \in \mathcal{F}} P^m\left\{R_{emp}(f) = 0 \mid R(f) > \epsilon\right\} \end{aligned}$$

Si $R(f) = P(f(X) \neq Y) > \epsilon$, alors la probabilité que f classe correctement un exemple est $1 - R(f) \leq 1 - \epsilon$ et pour m exemples indépendants,

$$P^m\left\{R_{emp}(f) = 0 \mid R(f) > \epsilon\right\} = \prod_{i=1}^{m} P\left\{f(X_i) = Y_i \mid R(f) > \epsilon\right\} \leq (1 - \epsilon)^m \leq e^{-\epsilon m}$$

Donc, en posant $\delta = |\mathcal{F}| e^{-\epsilon m}$,

$$P^m\left\{R(\hat{f}) \leq \frac{1}{m}\left(\log |\mathcal{F}| + \log \frac{1}{\delta}\right)\right\} \geq 1 - \delta$$

et il suffit d'avoir $m \geq m(\epsilon, \delta)$ pour valider la borne avec un ϵ donné.

25.4 Apprenabilité efficace

Il est aussi possible de considérer la complexité algorithmique de l'apprentissage en plus de la complexité en échantillon pour définir l'*apprenabilité efficace*. Une classe de fonctions est donc apprenable efficacement, si, en plus d'être apprenable, la complexité algorithme est aussi polynomiale.

Cependant, il faut rester vigilant face à un certain nombre de détails pour pouvoir définir cela correctement. En particulier :

- il faut équilibrer le temps de calcul entre l'apprentissage et la prédiction, sans quoi la plupart des calculs pourraient être « cachés » dans la fonction de prédiction qui referait l'apprentissage, et le temps de l'apprentissage serait nul ;
- il faut prendre en compte que la complexité en temps ne peut pas être uniquement une fonction du nombre d'exemples m, car pour $m \geq m(\epsilon, \delta)$, l'algorithme peut ignorer tous les exemples supplémentaires (car les performances sont suffisantes pour la borne PAC) et calculer ainsi en temps constant par rapport à $m > m(\epsilon, \delta)$, mais exponentiel par rapport à $m(\epsilon, \delta)$.

26 Apprentissage PAC agnostic

L'apprentissage PAC pour **Probablement Approximativement Correct** proposé par Valiant (1984) est limité au cas déterministe et souvent réalisable, où l'on s'attend à ce que l'erreur de généralisation du modèle tende vers zéro lorsque le nombre d'exemples augmente.

Cependant, ces hypothèses son très rarement vérifiées en pratique, notamment dès lors que l'étiquette Y prédite dépend de variables non mesurées et non présentes dans $\boldsymbol{X}$.

Pour palier à ce défaut, Kearns (1994) a proposé une version agnostic du cadre PAC. L'idée principale consiste à ne plus s'intéresser directement à la valeur du risque, mais plutôt à la différence entre le risque du modèle $\hat{f}$ et le risque du meilleur modèle de la classe $\mathcal{F}$ des modèles possibles (ce dernier pouvant être différent du meilleur modèle dans l'absolu).

26.1 Borne PAC agnostique

Dans ce qui suit, $\mathcal{F}$ est une classe de fonctions (l'ensembe des modèles possibles) et S la base d'apprentissage.

Un algorithme $\mathcal{A} : (\mathcal{F}, S) \mapsto \hat{f}$ admet une borne PAC agnostique si pour toute loi jointe P du couple $(\boldsymbol{X}, Y) \in \mathcal{X} \times \mathcal{Y}$, pour tout $\delta \in (0, 1)$,

$$P^m \left\{ R(\hat{f}) - \min_{f \in \mathcal{F}} R(f) \leq \epsilon \right\} \geq 1 - \delta$$

Cette borne garantie que l'on ne résoud le problème que de manière **probablement approximativement correcte** :

- le modèle est Approximativement Correct : son risque est proche (à ϵ près) du meilleur risque atteignable dans $\mathcal{F}$;
- la borne n'est que probablement valide, avec une probabilité d'au moins $1 - \delta$ (comme pour la borne PAC classique).

26.2 Complexité en échantillon agnostique (*sample complexity*)

La complexité en échantillon agnostique $m(\epsilon, \delta)$ d'un algorithme est **le plus petit m tel que la borne PAC agnostique ci-dessus soit valide**.

Cette complexité nous indique le nombre d'exemples nécessaire pour pouvoir garantir que l'algorithme résolve le problème au sens où il se rapproche de la meilleure fonction qu'il aurait pu sélectionner parmi $\mathcal{F}$.

26.3 Apprenabilité

Un ensemble de fonctions $\mathcal{F}$ est **agnostiquement apprenable** si il existe un algorithme $\mathcal{A} : (\mathcal{F}, S) \mapsto \hat{f}$ tel que **la complexité en échantillon agnostique $m(\epsilon, \delta)$ est polynomiale en $1/\epsilon$ et $1/\delta$**.

Autrement dit, un ensemble de modèles est considéré apprenable si le nombre d'exemples nécessaires à son apprentissage n'explose pas exponentiellement lorsque la performance désirée augmente ou lorsque que la confiance souhaitée dans les garanties théoriques augmente.

26.4 Apprenabilité et bornes uniformes

Considérons le meilleur modèle de la classe $\mathcal{F}$:

$$f^* = \arg\min_{f \in \mathcal{F}} R(f)$$

L'erreur d'estimation est

$$R(\hat{f}) - R(f^*) = R(\hat{f}) - \min_{f \in \mathcal{F}} R(f)$$

Pour l'algorithme ERM (qui minimise le risque empirique),

$$\hat{f} = \arg\min_{f \in \mathcal{F}} R_{emp}(f),$$

on a

$$\begin{aligned} R(\hat{f}) - R(f^*) &= R(\hat{f}) - R_{emp}(\hat{f}) + R_{emp}(\hat{f}) - R(f^*) \\ &\leq R(\hat{f}) - R_{emp}(\hat{f}) + R_{emp}(f^*) - R(f^*) \\ &\leq 2 \sup_{f \in \mathcal{F}} |R(f) - R_{emp}(f)| \end{aligned}$$

Ainsi, une borne uniforme à ϵ près du type :

$$P^m \left\{ \forall f \in \mathcal{F}, \quad |R(f) - R_{emp}(f)| \leq \epsilon \right\} \geq 1 - \delta$$

borne aussi avec une forte probabilité l'erreur d'estimation à 2ϵ près pour l'ERM, et fournit donc aussi une borne PAC agnostique. Une borne uniforme du type ci-dessus suffit donc à démontrer l'apprenabilité et est plus simple à dériver. L'avantage ici est que le terme $|R(f) - R_{emp}(f)|$ est simplement la déviation entre la moyenne empirique et l'espérance de l'erreur d'une fonction et ne dépend pas de f^* qui est difficile à caractériser.

La borne uniforme ci-dessus est aussi appelée borne sur l'erreur de généralisation car elle permet de garantir les performances en généralisation à partir de l'erreur sur la base d'apprentissage.

27 Bornes sur l'erreur de généralisation

Une borne sur l'erreur de généralisation permet de garantir les performances d'un modèle à partir de son comportement sur la base d'apprentissage.

La première borne calculée à partir de l'erreur de test est en effet limitée car

- elle demande de mettre des données de côté ;
- elle ne dépend pas du modèle f, ni de la classe de modèles $\mathcal{F}$, ni de l'algorithme utilisé, et ne peut donc pas nous aider à développer de nouveaux algorithmes ou à comprendre le comportement des algorithmes existants.

Ainsi, nous souhaitons ici obtenir une borne de la forme générale :

Avec une forte probabilité, pour toute fonction $f \in \mathcal{F}$,

$$R(f) \leq R_{emp}(f) + \epsilon(m, \mathcal{F})$$

où

- $R(f) = \mathbb{E}\ell(f, \boldsymbol{X}, Y)$ est le risque ;
- $R_{emp}(f) = \frac{1}{m}\sum_{i=1}^{m} \ell(f, \boldsymbol{X}_i, Y_i)$ est le risque empirique ;
- $\epsilon(m, \mathcal{F})$ est un terme de contrôle ou « (semi)-intervalle » de confiance qui prend en compte la classe de modèles $\mathcal{F}$.

Nous présentons ici la borne la plus simple pour le cas avec un nombre fini de modèles possibles. Le cas des classes de fonctions infinies est traité ici.

27.1 Borne pour les classes de fonctions finies

Tout ce qui suit se place dans le cadre de la classification avec $\ell(f, \boldsymbol{X}, Y) = \mathbf{1}(f(\boldsymbol{X}) \neq Y)$.

Commençons par considérer un ensemble très simple ne contenant que 2 modèles : $\mathcal{F} = \{f_1, f_2\}$. Dans ce cas, l'inégalité de Hoeffding appliquée à f_1 donne

$$P^m\{R(f_1) - R_{emp}(f_1) > \epsilon\} \leq \delta$$

avec $\delta = \exp\left(-2m\epsilon^2\right)$.

De même, pour f_2,

$$P^m\{R(f_2) - R_{emp}(f_2) > \epsilon\} \leq \delta$$

La probabilité d'observer un jeu de données S conduisant à une grande erreur pour f_1 ou f_2 est donc, en utilisant la borne de l'union (Équation B.1) :

$$\begin{aligned} P^m\{R(f_1) - R_{emp}(f_1) > \epsilon \cup R(f_2) - R_{emp}(f_2) > \epsilon\} \\ &\leq P^m(R(f_1) - R_{emp}(f_1) > \epsilon) + P^m(R(f_2) - R_{emp}(f_2) > \epsilon) \\ &\leq 2\delta \end{aligned}$$

De manière générale, pour une classe de fonctions $\mathcal{F}$ finie avec $|\mathcal{F}| = N < +\infty$, le même raisonnement s'applique et

$$\begin{aligned} P^m\{\exists f_j \in \mathcal{F},\ R(f_j) - R_{emp}(f_j) > \epsilon\} &\leq \sum_{j=1}^{N} P^m\{R(f_j) - R_{emp}(f_j) > \epsilon\} \\ &\leq N\delta \end{aligned}$$

Ainsi, si l'on redéfinit $\delta = N \exp\left(-2m\epsilon^2\right)$, et que l'on résoud cette équation pour obtenir ϵ, on obtient le théorème suivant.

Théorème 27.1. *Pour tout $\delta > 0$, avec une probabilité égale ou supérieure à $1 - \delta$,*

$$\forall f \in \mathcal{F}, \quad R(f) \leq R_{emp}(f) + \sqrt{\frac{\ln N + \ln \frac{1}{\delta}}{2m}}$$

> **Note**
>
> Cette borne est
>
> - valide **simultanément pour toute fonction de** $\mathcal{F}$ et donc en particulier pour le classifieur $\hat{f} \in \mathcal{F}$ sélectionné par un algorithme d'apprentissage.
> - **indépendante de la distribution des données** P
> - **indépendante de l'algorithme** utilisé pour choisir $\hat{f} \in \mathcal{F}$

27.1.1 Interprétation de la borne

La borne du Théorème 27.1 illustre le compromis entre la taille de la classe de fonctions et le risque empirique.

Soit deux classes de fonctions telles que $\mathcal{F}_1 \subset \mathcal{F}_2$, et donc avec des nombres de fonctions $N_2 > N_1$. En passant de $\mathcal{F}_1$ à $\mathcal{F}_2$:

- l'erreur d'apprentissage du classifieur retenu par l'algorithme ERM diminue (une meilleure approximation des données peut être trouvée) car

$$\min_{f \in \mathcal{F}_2} R_{emp}(f) \leq \min_{f \in \mathcal{F}_1} R_{emp}(f)$$

- mais la confiance dans cet algorithme diminue aussi.

Dis autrement, plus la taille de la classe de fonctions est grande, plus le nombre de données d'apprentissage doit être grand pour garantir que le risque reste proche du risque empirique et ainsi éviter le surapprentissage.

27.1.2 Complexité en échantillon

Le Théorème 27.1 permet de déterminer le nombre d'exemples nécessaires pour apprendre correctement, c'est-à-dire garantir (avec une probabilité d'au moins $1 - \delta$) un risque $R(f) \leq R_{emp}(f) + \epsilon$ pour N, ϵ et δ fixés :

$$m \geq \frac{\ln N + \ln \frac{1}{\delta}}{2\epsilon^2} = O\left(\frac{\ln \frac{N}{\delta}}{\epsilon^2}\right)$$

Pour une certitude de 95%, cela donne les nombres ci-dessous.

Table 27.1: Nombre d'exemples nécessaires pour $\delta = 0.05$

Nombre de modèles N	avec $\epsilon = 0.02$	avec $\epsilon = 0.01$
10	6623	26492
100	9502	38005
1000	12380	49518
100000	18136	72544

Ce tableau montre l'intérêt d'une complexité en échantillon logarithmique par rapport à la taille de $\mathcal{F}$. Cependant celle-ci a une limite... pour N infini, la borne devient inutile.

Pour traiter ce cas, il faudra trouver d'autres moyens de mesurer la complexité de la classe $\mathcal{F}$, comme par exemple avec la VC dimension.

28 Bornes pour les classes de fonctions infinies

Une borne sur l'erreur de généralisation permet de garantir les performances d'un modèle à partir de son comportement sur la base d'apprentissage.

La première borne de ce type est inutile pour un nombre de modèles possibles $|\mathcal{F}|$ infini, ce qui est gênant car

- beacoup de classifieurs implémentent une infinité de modèles ;
- un des classifieurs les plus simple, le classifieur linéaire, est déjà dans ce cas : avec

$$\mathcal{F} = \{f : f(\boldsymbol{x}) = \text{signe}(\boldsymbol{w}^T\boldsymbol{x} + b),\ \boldsymbol{w} \in \mathbb{R}^d,\ b \in \mathbb{R}\}$$

  on a $|\mathcal{F}| = \infty$.

Pour pouvoir garantir les performances dans de tels cas, nous allons utiliser d'autres manières de mesurer la complexité de $\mathcal{F}$, plus fines que simplement compter le nombre de classifieurs.

28.1 Fonction de croissance

L'astuce est la suivante : même avec une infinité de fonctions, le nombre de manières de classer (étiqueter) des exemples est fini.

Ainsi, au lieu de compter le nombre de classifieurs $N = |\mathcal{F}|$, nous compterons le nombre de classifications différentes que peuvent générer ces classifieurs.

Définissons tout d'abord la **projection d'une classe de fonctions $\mathcal{F}$ sur un jeu de données** $S = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m)$:

$$\mathcal{F}_S = \{(f(\boldsymbol{x}_1), \ldots, f(\boldsymbol{x}_m)) \mid f \in \mathcal{F}\} \subseteq \{-1, +1\}^m$$

qui correspond à l'ensemble de tous les étiquetages possibles de S fournis par les fonctions $f \in \mathcal{F}$.

Il est évident que le nombre d'éléments dans cet ensemble est borné par le nombre maximum de classifications binaires de m points :

$$|\mathcal{F}_S| \leq 2^m$$

Si $|\mathcal{F}_S| = 2^m$, $\mathcal{F}$ peut réaliser n'importe quelle classification de S : on dit alors que S **est « pulvérisé » par** $\mathcal{F}$ ou que $\mathcal{F}$ « pulvérise » S.

La **fonction de croissance** d'une classe de fonctions $\mathcal{F}$ est le nombre maximum de manières d'étiqueter un ensemble de m points de $\mathcal{X}$ par $\mathcal{F}$:

$$\Pi_{\mathcal{F}}(m) = \max_{S \in \mathcal{X}^m} |\mathcal{F}_S|$$

et il est clair que

$$\Pi_{\mathcal{F}}(m) \leq 2^m$$

Remarquons que la fonction de croissance donne une **mesure au pire cas** de la complexité (ou la capacité) de $\mathcal{F}$: son calcul dépend du pire jeu de m points qui conduit au plus grand nombre de classifications.

Ainsi, si $\Pi_{\mathcal{F}}(m) = 2^m$, alors $\mathcal{F}$ pulvérise au moins un ensemble S de taille m, mais pas nécessairement tous les ensembles de m.

28.1.1 Borne sur l'erreur de généralisation

Théorème 28.1. *Pour tout $\delta > 0$, avec une probabilité d'au moins $1 - \delta$,*

$$\forall f \in \mathcal{F}, \quad R(f) \leq R_{emp}(f) + 2\sqrt{2\frac{\ln \Pi_{\mathcal{F}}(2m) + \ln \frac{2}{\delta}}{m}}$$

Cette borne dépend de la fonction de croissance de $\mathcal{F}$ de telle sorte que **le nombre de classifications possibles remplace le nombre de classifieurs** par rapport à la borne pour les classes finies.

Pour permettre un apprentissage correct par minimisation du risque empirique (ERM), cette borne suggère donc qu'il est nécessaire d'avoir

$$\lim_{m \to +\infty} \frac{\ln \Pi_{\mathcal{F}}(2m)}{m} = 0$$

Cette condition permet de s'assurer que le risque empirique tend bien vers l'erreur de généralisation lorsque le nombre d'exemples m tend vers l'infini.

On peut remarquer que cette condition n'est pas vérifiée si $\mathcal{F}$ peut pulvériser des jeux de données de taille arbitraire. En effet, tant que $\Pi_{\mathcal{F}}(2m) = 2^{2m}$,

$$\lim_{m \to +\infty} \frac{\ln \Pi_{\mathcal{F}}(2m)}{m} = 2 \log 2$$

et l'écart entre les deux risques ne peut pas converger vers zéro.

L'interprétation est la suivante : si $\mathcal{F}$ peut générer n'importe quelle classification de $2m$ points, alors il est possible de choisir un classifieur $f \in \mathcal{F}$ qui donne des prédictions parfaites sur les données d'apprentissage, tout en étant capable de générer n'importe quelle classification d'un autre jeu de m points (comme une base de test par exemple).

Dit autrement : *on ne peut pas garantir les performances en généralisation d'un classifieur capable de faire n'importe quoi.*

28.1.2 Symétrisation

Pour démontrer le Théorème 28.1, il faut passer par l'introduction d'un **échantillon fantôme**

$$S' = ((X'_i, Y'_i))_{1 \leq i \leq m}$$

contenant des copies indépendantes de (X, Y) au même titre que l'échantillon d'apprentissage $S = ((X_i, Y_i))_{1 \leq i \leq m}$, et sur lequel on calcule

$$R'_{emp}(f) = \frac{1}{m} \sum_{i=1}^{m} \mathbf{1}(f(X'_i) \neq Y'_i)$$

de la même manière que $R_{emp}(f)$.

Attention : ces X'_i, Y'_i ne servent qu'à la démonstration et ne correspondent à aucune donnée réelle que l'on mettrait de côté.

À partir de ces données fantômes, nous pouvons appliquer la symétrisation.

Lemme 28.1 (Symétrisation)**.** *Pour* $m\epsilon^2 \geq 2$*,*

$$P\left\{\sup_{f \in \mathcal{F}}(R(f) - R_{emp}(f)) \geq \epsilon\right\} \leq 2P\left\{\sup_{f \in \mathcal{F}}(R'_{emp}(f) - R_{emp}(f)) \geq \frac{\epsilon}{2}\right\} = 2\boldsymbol{P}_\epsilon$$

Preuve ici.

L'intérêt de la symétrisation est de pouvoir se passer d'étudier les fonctions f sur un ensemble infini de points $\boldsymbol{x} \in \mathcal{X}$:

- le terme $\boldsymbol{P}_\epsilon$ de droite ne fait intervenir que des risques empiriques, et donc uniquement les valeurs de f sur les échantillons S et S' ;
- il ne dépend donc que de la projection $\mathcal{F}_{SS'}$ avec $SS' = (\boldsymbol{X}_1, \ldots, \boldsymbol{X}_m, \boldsymbol{X}'_1, \ldots, \boldsymbol{X}'_m)$ la concaténation des deux échantillons.

28.1.3 Reste de la preuve du Théorème 28.1

Introduisons des variables aléatoires σ_i uniforme dans $\{-1, +1\}$ (telles que $P(\sigma_i = 1) = P(\sigma_i = -1) = 1/2$) et indépendantes de toutes les autres variables. Alors,

$$D_i(f) = \mathbf{1}(f(\boldsymbol{X}'_i) \neq Y'_i) - \mathbf{1}(f(\boldsymbol{X}_i) \neq Y_i) \quad \text{est distribuée comme} \quad \sigma_i D_i(f)$$

ce qui signifie que les variables aléatoires $D_i(f)$ et $\sigma_i D_i(f)$ ont la même loi de probabilité.

Preuve

Premièrement, $D_i(f) \in \{-1, 0, +1\}$ est **symétrique** :

$$\begin{aligned}
P(D_i(f) = 1) &= P(f(\boldsymbol{X}'_i) \neq Y'_i, f(\boldsymbol{X}_i) = Y_i) \\
&= P(f(\boldsymbol{X}'_i) \neq Y'_i) P(f(\boldsymbol{X}_i) = Y_i) && \textit{car } (\boldsymbol{X}'_i, Y'_i) \text{ indép. } (\boldsymbol{X}_i, Y_i) \\
&= P(f(\boldsymbol{X}_i) \neq Y_i) P(f(\boldsymbol{X}'_i) = Y'_i) && \textit{car } (\boldsymbol{X}'_i, Y'_i) \textit{ id. distrib. } (\boldsymbol{X}_i, Y_i) \\
&= P(f(\boldsymbol{X}_i) \neq Y_i, f(\boldsymbol{X}'_i) = Y'_i) \\
&= P(D_i(f) = -1)
\end{aligned}$$

et donc

$$P(\sigma_i D_i(f) = 1) = \begin{cases} P(D_i(f) = 1), & \text{si } \sigma_i = 1 \\ P(D_i(f) = -1), & \text{si } \sigma_i = -1 \end{cases} = P(D_i(f) = 1)$$

et de même $P(\sigma_i D_i(f) = -1) = P(D_i(f) = -1)$, et bien sûr $P(\sigma_i D_i(f) = 0) = P(D_i(f) = 0)$

Il est donc possible de remplacer $D_i(f)$ par $\sigma_i D_i(f)$ sans changer la probabilité :

$$\boldsymbol{P}_\epsilon = P\left\{\sup_{f \in \mathcal{F}} \frac{1}{m} \sum_{i=1}^{m} D_i(f) \geq \frac{\epsilon}{2}\right\} = P\left\{\sup_{f \in \mathcal{F}} \frac{1}{m} \sum_{i=1}^{m} \sigma_i D_i(f) \geq \frac{\epsilon}{2}\right\}$$

En utilisant l'espérance de l'indicatrice et le théorème de l'espérance totale, on a de manière générale $P(A) = \mathbb{E}\mathbf{1}(A) = \mathbb{E}_B \mathbb{E}[\mathbf{1}(A)|B] = \mathbb{E}_B P\{A|B\}$, et donc

$$\boldsymbol{P}_\epsilon = \mathbb{E}_{SS'} P\left\{\sup_{f \in \mathcal{F}} \frac{1}{m} \sum_{i=1}^{m} \sigma_i D_i(f) \geq \frac{\epsilon}{2} \,\middle|\, SS'\right\}$$

Définissons le vecteur des classifications produites par f :

$$\boldsymbol{f} = (f(\boldsymbol{X}_1), \dots f(\boldsymbol{X}_m), f(\boldsymbol{X}'_1), \dots, f(\boldsymbol{X}'_m)) \in \mathcal{F}_{SS'} \subset \{-1, +1\}^{2m}$$

qui permet de supprimer la dépendance de $D_i(f)$ à f en dehors des points $\boldsymbol{X}_i$ et $\boldsymbol{X}'_i$:

$$D_i(f) = D_i(\boldsymbol{f}) = \mathbf{1}(\boldsymbol{f}_{m+i} \neq Y'_i) - \mathbf{1}(\boldsymbol{f}_i \neq Y_i)$$

et donc de transformer $\sup_{f \in \mathcal{F}}$ en $\exists \boldsymbol{f} \in \mathcal{F}_{SS'}$ pour avoir

$$\boldsymbol{P}_\epsilon = \mathbb{E}_{SS'} P\left\{\exists \boldsymbol{f} \in \mathcal{F}_{SS'}, \frac{1}{m} \sum_{i=1}^{m} \sigma_i D_i(\boldsymbol{f}) \geq \frac{\epsilon}{2} \,\middle|\, SS'\right\}$$

Ainsi, en appliquant la borne de l'union (Équation B.1), nous obtenons

$$\begin{aligned}
\boldsymbol{P}_\epsilon &\leq \mathbb{E}_{SS'} \sum_{\boldsymbol{f} \in \mathcal{F}_{SS'}} P\left\{\frac{1}{m} \sum_{i=1}^{m} \sigma_i D_i(\boldsymbol{f}) \geq \frac{\epsilon}{2} \,\middle|\, SS'\right\} \\
&\leq \mathbb{E}_{SS'} \sum_{\boldsymbol{f} \in \mathcal{F}_{SS'}} \exp\left(\frac{-2m(\epsilon/2)^2}{2^2}\right)
\end{aligned}$$

où la dernière ligne vient de l'application de l'inégalité de Hoeffding avec $Z_i = \sigma_i D_i(\boldsymbol{f}) \in [-1, 1]$ et $\mathbb{E}[Z_i|SS'] = D_i(\boldsymbol{f})\mathbb{E}\sigma_i = 0$[1].

Ensuite, le terme dans la somme ne dépend plus de $\boldsymbol{f}$ et nous obtenons

$$\boldsymbol{P}_\epsilon \leq \mathbb{E}_{SS'}|\mathcal{F}_{SS'}| \exp\left(\frac{-m\epsilon^2}{8}\right) \leq \Pi_{\mathcal{F}}(2m) \exp\left(\frac{-m\epsilon^2}{8}\right)$$

car

$$\mathbb{E}_{SS'}|\mathcal{F}_{SS'}| \leq \max_{SS' \in \mathcal{X}^{2m}} |\mathcal{F}_{SS'}| = \Pi_{\mathcal{F}}(2m)$$

Pour finir, il suffit de poser

$$\delta = \Pi_{\mathcal{F}}(2m) \exp\left(\frac{-m\epsilon^2}{8}\right)$$

et de résoudre cette équation par rapport à ϵ pour obtenir le Théorème 28.1.

28.2 Dimension de Vapnik-Chervonenkis

La fonction de croissance permet de mesurer la complexité des classes de fonctions infinies, mais elle n'est pas simple à manipuler. En particulier, c'est une fonction du nombre de points m, et sa valeur doit donc être calculée/estimée pour tout m.

La dimension de Vapnik-Chervonenkis ou **VC dimension** permet de résumer le comportement de la fonction de croissance avec un seul nombre, plus simple à estimer. Elle est définie ainsi :

La VC dimension d'une classe de fonctions $\mathcal{F}$, notée d_{VC}, est le plus grand nombre de points m pulvérisables par $\mathcal{F}$, c'est-à-dire tel que

$$\Pi_{\mathcal{F}}(m) = 2^m$$

28.2.1 Borne à base de VC dimension

Le lien entre la fonction de croissance et la VC dimension est donné par le Lemme suivant qui indique que la fonction de croissance augmente exponentiellement avec m jusque $m = d_{VC}$, puis seulement polynomialement pour $m > d_{VC}$.

Lemme 28.2 (de Sauer–Shelah)**.** *Pour tout $m > d_{VC}$,*

$$\Pi_{\mathcal{F}}(m) \leq \sum_{k=0}^{d_{VC}} \binom{m}{k} \leq \left(\frac{em}{d_{VC}}\right)^{d_{VC}}$$

[1] $\mathbb{E}\sigma_i = +1P(\sigma_i = +1) - 1P(\sigma_i = -1) = 0.5 - 0.5 = 0$

où $\mathrm{e} = \exp(1)$ *et les coefficients du binômes sont donnés par*

$$\binom{n}{k} = \begin{cases} \frac{n!}{k!(n-k)!}, & \textit{si } 0 \leq k \leq n \\ 0, & \textit{sinon.} \end{cases}$$

Preuve du Lemme 28.2

La preuve de la première inégalité se fait par une double induction sur la VC dimension d'une part et la taille de la base d'apprentissage m d'autre part selon le schéma suivant :

1. cas de base pour $m = 0$ et tout $d_{VC} \geq 0$;
2. cas de base pour $d_{VC} = 0$ et tout $m \geq 1$;
3. induction du cas m, d_{VC} à partir de la validité pour $m - 1$ points et toute VC-dimension $< d_{VC}$.

La seconde inégalité utilise simplement des calculs sur les coefficients du binôme.
Cas de base pour $m = 0$ **:** sans aucun point, il n'y a aucune manière de les classer et on a $\Pi_{\mathcal{F}}(0) = 0$, ce qui est bien inférieur à $\sum_{k=0}^{d_{VC}} \binom{m}{k}$ pour toute d_{VC}.
Cas de base pour $d_{VC} = 0$ **:** aucun point ne peut être pulvérisé par $\mathcal{F}$ et la prédiciton $f(\boldsymbol{x})$ est donc constante pour tout $f \in \mathcal{F}$ quel que soit $\boldsymbol{x} \in \mathcal{X}$. Donc, pour tout m, il n'y a qu'une seule classification possible et $\Pi_{\mathcal{F}}(0) = 1$ qui est bien inférieur ou égal à la borne $\sum_{k=0}^{0} \binom{m}{k} = \binom{m}{0} = 1$.
Induction : si le Lemme est valide pour $m - 1$ points et toute VC-dimension $< d_{VC}$, alors pour tout $F' \subset F \subset \mathcal{F}$,

$$\Pi_{F'}(m-1) \leq \Pi_{\mathcal{F}}(m-1) \leq \sum_{k=0}^{d_{VC}} \binom{m-1}{k}$$

et si $d_{VC}(F \setminus F') \leq d_{VC} - 1$,

$$\Pi_{F \setminus F'}(m-1) \leq \sum_{k=0}^{d_{VC}(F \setminus F')} \binom{m-1}{k} \leq \sum_{k=0}^{d_{VC}-1} \binom{m-1}{k} = \sum_{k=1}^{d_{VC}} \binom{m-1}{k-1}$$

Dans ce cas,

$$\Pi_{F'}(m-1) + \Pi_{F \setminus F'}(m-1) \leq 1 + \sum_{k=1}^{d_{VC}} \binom{m-1}{k} + \binom{m-1}{k-1} = 1 + \sum_{k=1}^{d_{VC}} \binom{m}{k} = \sum_{k=0}^{d_{VC}} \binom{m}{k}$$

car $\binom{m}{k} = \binom{m-1}{k} + \binom{m-1}{k-1}$ (cette identité caractérise le comptage du nombre de manières de choisir k éléments parmi m en deux étapes. Dans la prmeière étape, on laisse de côté le premier élément et on compte le nombre de manières de choisir k éléments parmi

$m-1$. Dans la seconde étape, on choisit le premier élément et on compte le nombre de manières de choisir les $k-1$ restant parmi $n-1$. Le nombre total de choisir k éléments parmi m, $\binom{m}{k}$ est la somme des deux comptes.)
Choisissons un ensemble de points $S = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m\}$ tel que les $\Pi_{\mathcal{F}}(m)$ classifications de ces points soient toutes obtenues avec des classifieurs $f \in \mathcal{F}$ différents (cela implique aussi que S est choisi de telle sorte qu'un nombre maximal de classifications différentes soient obtenues). Notons $F \subset \mathcal{F}$ l'ensemble de ces classifieurs. Définissons $S' = S \setminus \{\boldsymbol{x}_1\}$ et $F' \subset F$ le plus petit sous-ensemble de F qui peut classer S' du plus grand nombre de manières possibles (ainsi, F' ne peut pas contenir deux classifieurs qui produisent la même classification de S').
Pour chaque classification de S', deux classifieurs de F produisent cette classification et diffèrent sur $\boldsymbol{x}_1$, sachant que l'un est dans F' et l'autre dans $F \setminus F'$. Considérons la partition de F : $F = F' \cup (F \setminus F')$ avec la relation $|F| = |F'| + |F \setminus F'|$.
Puisque F a été construit avec une seule fonction pour chaque classification de S, $|F| = \Pi_{\mathcal{F}}(m)$ et $|F'|$ et $|F \setminus F'|$ représentent aussi le nombre de classifications de S' produites par F' et $F \setminus F'$. Ainsi :

$$|F'| \leq \Pi_{F'}(m-1) \qquad \text{et} \qquad |F \setminus F'| \leq \Pi_{F \setminus F'}(m-1)$$

Nous allons maintenant nous intéresser à la VC dimension de ces ensembles. Puisque $F' \subset F \subset \mathcal{F}$, alors leur VC-dimension respecte

$$d_{VC}(F') \leq d_{VC}(F) \leq d_{VC}(\mathcal{F})$$

S'il existe un ensemble de points $T \subset S'$ pulvérisé par $F \setminus F'$, alors F' pulvérise aussi T car F' a été choisi pour produire la plus grand nombre de classifications de S' et donc aussi de ses sous-ensembles.
Ainsi, pour chaque classification de T, il existe $f \in F \setminus F'$ et $f' \in F'$ qui produisent cette classification. Mais puisque F ne contient qu'une fonction pour chaque classification de S, cela implique que f et f' doivent être en désaccord sur $S \setminus T$. Et puisque F' produit le plus grand nombre de classifications de S', quelle que soit la classification de S' donnée par f, il existe un $f' \in F'$ qui la produit aussi. Le désaccord entre f et f' doit donc être observé sur $S \setminus S' = \{\boldsymbol{x}_1\}$ et $f(\boldsymbol{x}_1) \neq f'(\boldsymbol{x}_1)$. Puisque toutes les paires f et f' construites ainsi appartiennet à F, l'ensemble de points $T \cup \{\boldsymbol{x}_1\}$ est donc pulvérisé par F (car toutes les classifications de $\boldsymbol{x}_1$ sont possibles en plus de celles de T).
D'un autre côté, $F \setminus F'$ ne peut pas pulvériser $T \cup \{\boldsymbol{x}_1\}$ et donc $d_{VC}(F \setminus F') \leq d_{VC}(F) - 1 \leq d_{VC}(\mathcal{F}) - 1$.
Au final,

$$\Pi_{\mathcal{F}}(m) = |F| = |F'| + |F \setminus F'| \leq \Pi_{F'}(m-1) + \Pi_{F \setminus F'}(m-1) \leq \sum_{k=0}^{d_{VC}} \binom{m}{k}$$

Borne polynomiale : pour obtenir la dernière borne polynomiale, nous utilisons, pour $m > d_{VC}$,

$$\sum_{k=0}^{d_{VC}} \binom{m}{k} \leq \left(\frac{em}{d_{VC}}\right)^{d_{VC}}$$

que l'on peut prouver ainsi.
Pour $d < n$, $n/d > 1$ et pour $k \leq d$, $(n/d)^{d-k} \geq 1$. Ainsi,

$$\begin{aligned}
\sum_{k=0}^{d_{VC}} \binom{m}{k} &\leq \sum_{k=0}^{d_{VC}} \binom{m}{k} \left(\frac{m}{d_{VC}}\right)^{d_{VC}-k} \\
&= \left(\frac{m}{d_{VC}}\right)^{d_{VC}} \sum_{k=0}^{d_{VC}} \binom{m}{k} \left(\frac{d_{VC}}{m}\right)^{k} \\
&\leq \left(\frac{m}{d_{VC}}\right)^{d_{VC}} \sum_{k=0}^{m} \binom{m}{k} \left(\frac{d_{VC}}{m}\right)^{k} \\
&= \left(\frac{m}{d_{VC}}\right)^{d_{VC}} \left(1 + \frac{d_{VC}}{m}\right)^{m} \\
&\leq \left(\frac{m}{d_{VC}}\right)^{d_{VC}} \left(e^{d_{VC}/m}\right)^{m} \\
&= \left(\frac{m}{d_{VC}}\right)^{d_{VC}} e^{d_{VC}} \\
&= \left(\frac{m}{e d_{VC}}\right)^{d_{VC}}
\end{aligned}$$

La croissance polynomiale donnée par le Lemme 28.2 permet de garantir que la borne du Théorème 28.1 converge bien vers le risque empirique en $O\left(\sqrt{\frac{\ln m}{m}}\right)$:

Théorème 28.2 (Borne de Vapnik-Chervonenkis)**.** *Pour tout $m > d_{VC}/2$ et $\delta > 0$, avec une probabilité d'au moins $1 - \delta$,*

$$\forall f \in \mathcal{F}, \quad R(f) \leq R_{emp}(f) + 2\sqrt{2\frac{d_{VC} \ln \frac{2em}{d_{VC}} + \ln \frac{2}{\delta}}{m}}$$

Ce théorème permet en particulier de garantir les performances en généralisation des classifieurs linéaires pour lesquelles la VC dimension peut être calculée exactement.

28.2.2 VC dimension des classifieurs linéaires

Pour l'ensemble des classifieurs linéaires de $\mathcal{X} = \mathbb{R}^d$:

$$\mathcal{F} = \{f \in \mathcal{Y}^{\mathcal{X}} | f(x) = \text{signe}(w^T x + b),\ w \in \mathbb{R}^d,\ b \in \mathbb{R}\}$$

la VC dimension vaut

$$d_{VC} = d + 1$$

28.2.3 VC dimension infinie

La VC dimension peut être infinie si l'ensemble des classifieurs $\mathcal{F}$ pulvérise un nombre arbitraire de points. Dans ce cas, le Théorème 28.2 devient trivial/inutile et il est nécessaire de considérer d'autres mesures de capacité comme la complexité de Rademacher.

VC dimension infinie du classifieur du plus proche voisin

Pour la méthode des K-plus proches voisins avec $K = 1$: $d_{VC} = +\infty$.
En effet, avec $K = 1$, le classifieur Kppv donne toujours des prédictions parfaites sur la base d'apprentissage, car chaque point de cette base est lui-même son plus proche voisin.
Ainsi, pour produire une certaine classification de m points avec cet algorithme, il suffit de lui fournir une base d'apprentissage avec ces m points et la classification voulue, quel que soit m.

VC dimension infinie des SVM à noyau gaussien

Pour les SVM à noyau gaussien (Équation 19.1), $d_{VC} = +\infty$.
En effet, le noyau gaussien projette implicitement les données dans un espace de dimension infinie, où une classification linéaire est appliquée. La VC dimension est donc celle d'un classifieur linéaire en dimension infinie.
Les performances en généralisation des classifieurs à marge comme les SVM s'expliquent donc plutôt en passant par d'autres mesures de capacité comme la complexité de Rademacher.

La VC dimension des classifieurs linéaires pourrait laisser supposer que la VC dimension est directement liée au nombre de paramètres. Mais l'exemple des classifieurs sinusoïdaux montre que c'est loin d'être le cas.

VC dimension infinie des classifieurs sinusoïdaux

Pour les classifieurs « sinusoïdaux » de $\mathcal{X} = [0, 2\pi]$ qui ne possèdent qu'un seul paramètre ω :

$$\mathcal{F} = \{f | f(x) = \text{signe}\left(sin(\omega x)\right),\ \omega \geq 0\}$$

$$d_{VC} = +\infty$$

Preuve

La VC dimension est une mesure de capacité au pire cas, ce qui signifie que nous pouvons choisir la position des m points :

$$x_i = 2\pi 10^{-i}$$

Pour chaque classification $\boldsymbol{y} \in \{-1, 1\}^m$ nous fixons la pulsation du sinus à

$$\omega = \frac{1}{2}\left(1 + \sum_{i=1}^{m} \frac{1 - y_i}{2} 10^i\right).$$

Les étiquettes négatives sont correctement prédites :
On peut réécrire le paramètre comme

$$\omega = \frac{1}{2}\left(1 + \sum_{\{i\,:\,y_i=-1\}} 10^i\right)$$

où, pour tout point $x_j = 2\pi 10^{-j}$ dans notre jeu de données tel que $y_j = -1$, le terme 10^j apparaît dans la somme. Cela donne

$$\begin{aligned}
\omega x_j &= \pi 10^{-j}\left(1 + \sum_{\{i\,:\,y_i=-1\}} 10^i\right) \\
&= \pi 10^{-j}\left(1 + 10^j + \sum_{\{i\,:\,y_i=-1,\ i\neq j\}} 10^i\right) \\
&= \pi\left(10^{-j} + 1 + \sum_{\{i\,:\,y_i=-1,\ i\neq j\}} 10^{i-j}\right) \\
&= \pi\left(10^{-j} + 1 + \sum_{\{i\,:\,y_i=-1,\ i>j\}} 10^{i-j} + \sum_{\{i\,:\,y_i=-1,\ i<j\}} 10^{i-j}\right)
\end{aligned}$$

Pour tout $i > j$, les termes 10^{i-j} sont des puissances positives de 10 et donc des nombres pairs qui peuvent s'écrire $2k_i$ pour un certain $k_i \in \mathbb{N}$. Ainsi,

$$\sum_{\{i\,:\,y_i=-1,\ i>j\}} 10^{i-j} = \sum_{\{i\,:\,y_i=-1,\ i>j\}} 2k_i = 2k$$

pour un certain $k \in \mathbb{N}$, ce qui donne

$$\omega x_j = \pi\left(10^{-j} + 1 + \sum_{\{i\,:\,y_i=-1,\ i<j\}} 10^{i-j}\right) + 2k\pi.$$

Pour le reste de la somme, on a

$$\sum_{\{i\,:\,y_i=-1,\ i<j\}} 10^{i-j} < \sum_{i=1}^{+\infty} 10^{-i} = \sum_{i=0}^{+\infty} 10^{-i} - 1$$

La série géométrique $S_m = \sum_{i=0}^{m-1} 10^{-i}$, de premier terme 1 et de raison 0.1, converge vers

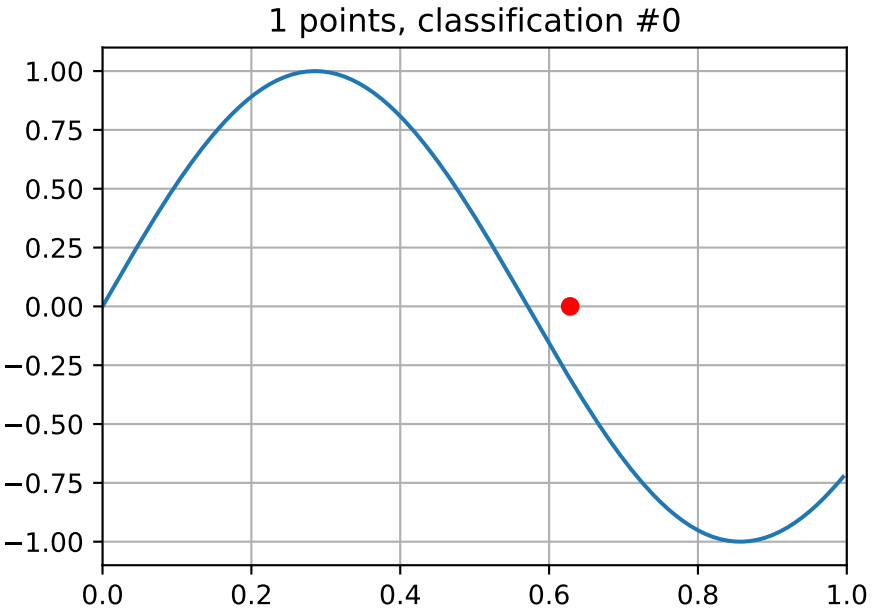
$$\sum_{i=0}^{+\infty} 10^{-i} = \frac{1}{1 - 0.1},$$

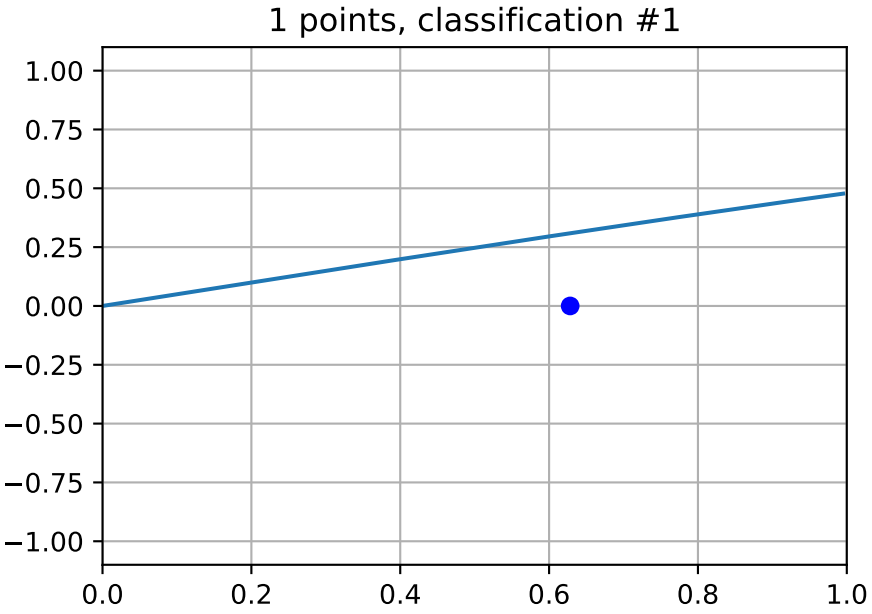
ce qui donne

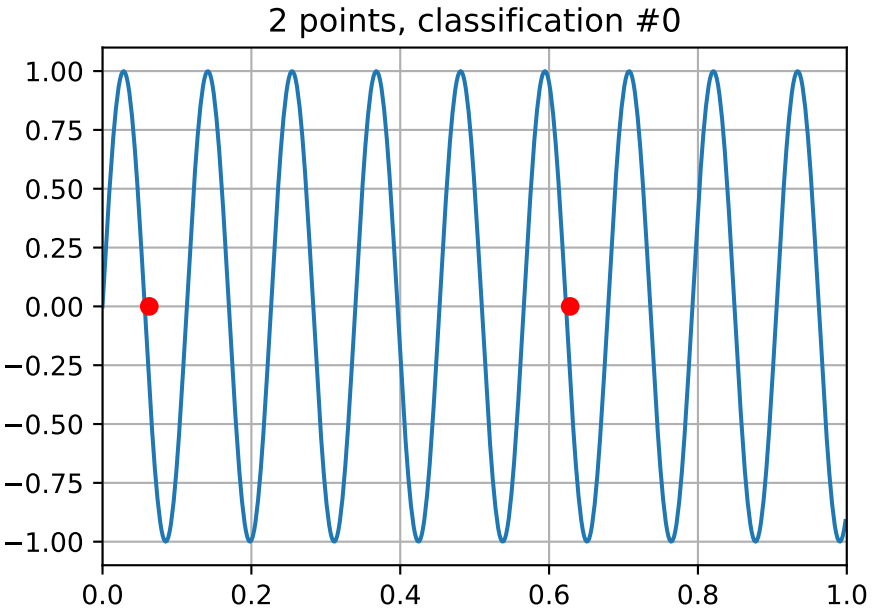
$$\sum 10^{i-j} < \frac{1}{1 - 0.1} - 1 = \frac{1}{9}.$$

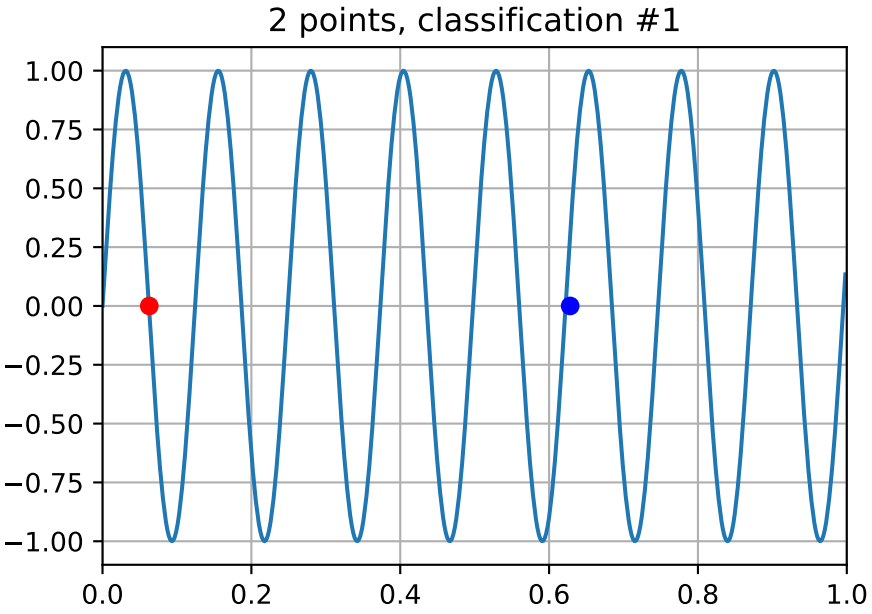
En d'autres termes, il est toujours possible d'obtenir n'importe quel slalome autour de m points avec un sinus en choisissant correctement la position des points et la fréquence du sinus.

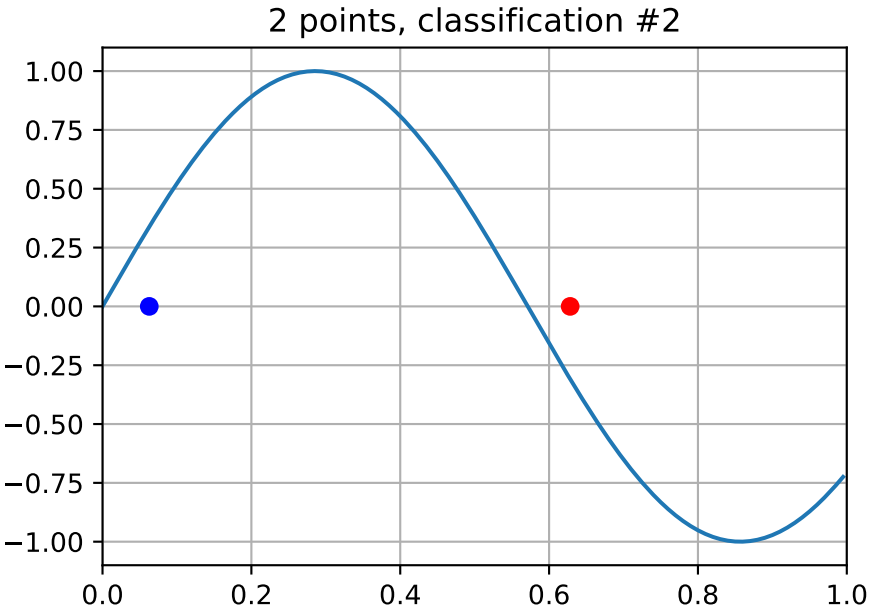
```
def f(x):
    return np.sign(omega * x)

def omega(y):
    tentoi = 10
    w = 1.0
    for i in range(len(y)):
        w += tentoi * (1-y[i])/2
        tentoi *= 10
    w /= 2
    return w
def generateBinaryLabels ( num_labeling, m):
    # Generate one (the num_labeling's) of the 2^m binary labelings
    y = np.zeros(m)

    div = 1
    y[0] =  1 + num_labeling % 2
    for i in range(1, m):
        div *= 2
        y[i] = 1 +  int(np.floor( num_labeling / div ) % 2)
    return 2*(y-1)-1

def generatePoints(m):
    x = 2 * np.pi * 0.1;
    X=np.zeros(m)
    for i in range(m):
        X[i] = x
        x *= 0.1
    return X


for m in range(1,4):
    X = generatePoints(m)
    for k in range(2**m):
        y = generateBinaryLabels(k, m)
        w = omega(y)
        Nsamples = max(10 *  w, 200)
        plt.figure()
        x = np.arange(0,1,1/ Nsamples)
        plt.plot(x,np.sin(w*x))
        plt.plot(X[y==1],np.zeros(np.sum(y==1)), "ob")
        plt.plot(X[y!=1],np.zeros(np.sum(y!=1)), "or")
        plt.grid()
        plt.axis([0,1, -1.1, 1.1])
        t=plt.title(str(m) + " points, classification #" + str(k))
```

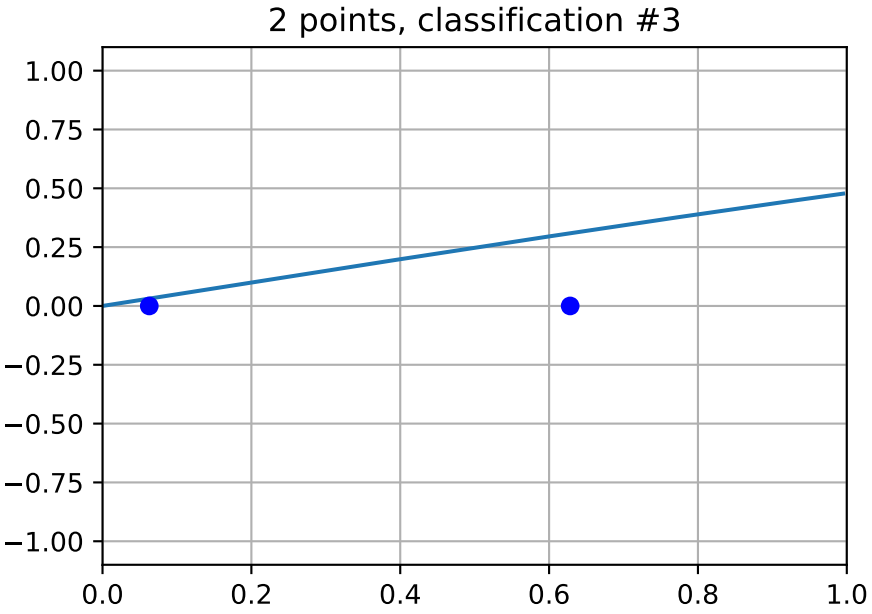

1 points, classification #0
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0.0
0.2
0.4
0.6
0.8
1.0

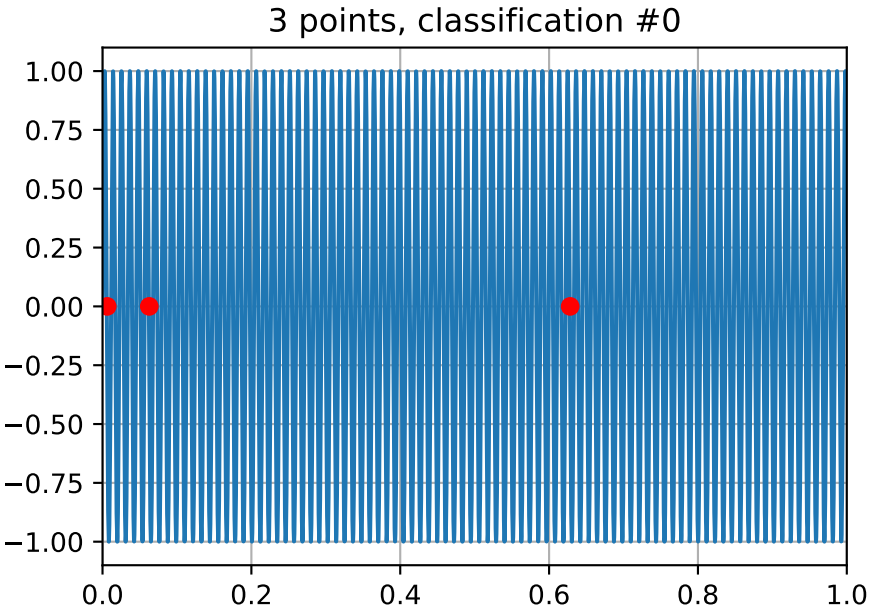

1 points, classification #1
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0.0
0.2
0.4
0.6
0.8
1.0

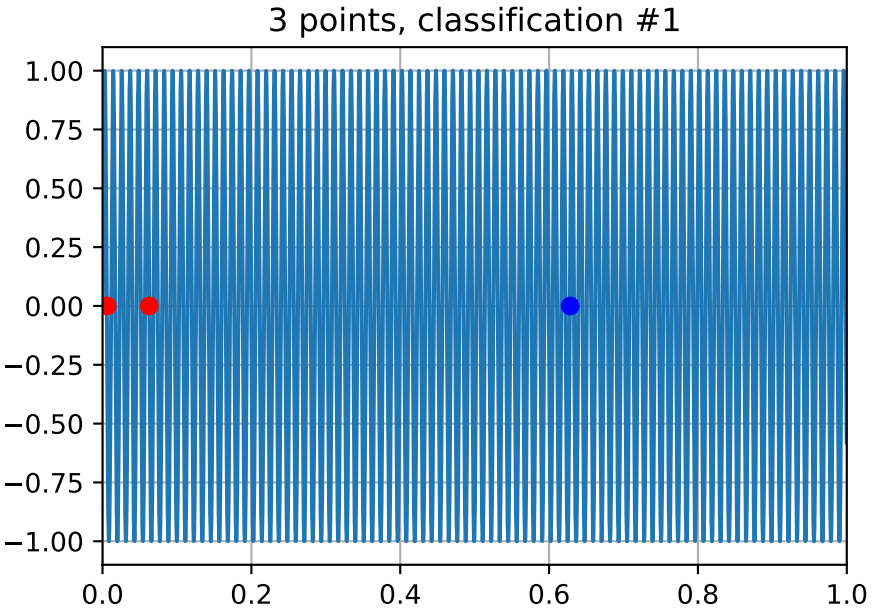

2 points, classification #0
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0.0
0.2
0.4
0.6
0.8
1.0

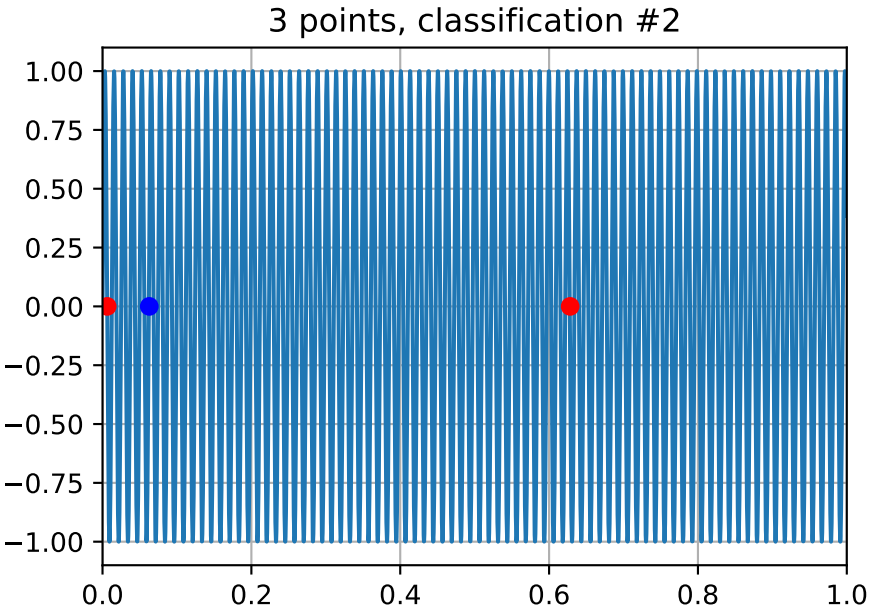

2 points, classification #1
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0.0
0.2
0.4
0.6
0.8
1.0

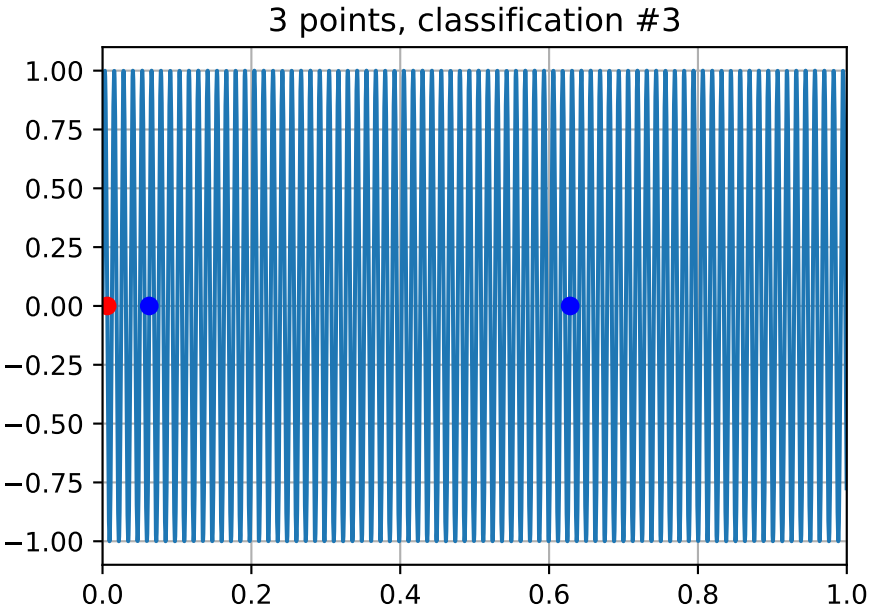

2 points, classification #2
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0.0
0.2
0.4
0.6
0.8
1.0

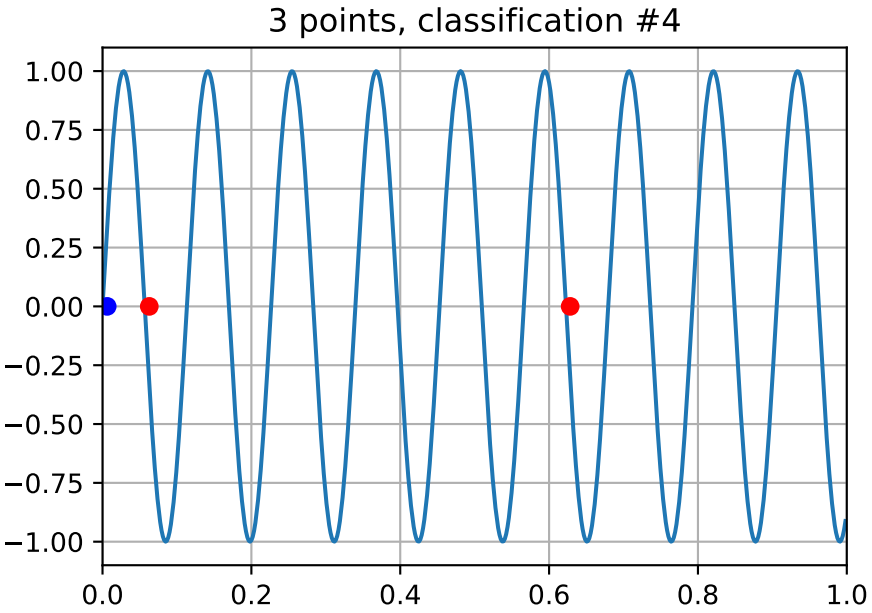

2 points, classification #3
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0.0
0.2
0.4
0.6
0.8
1.0

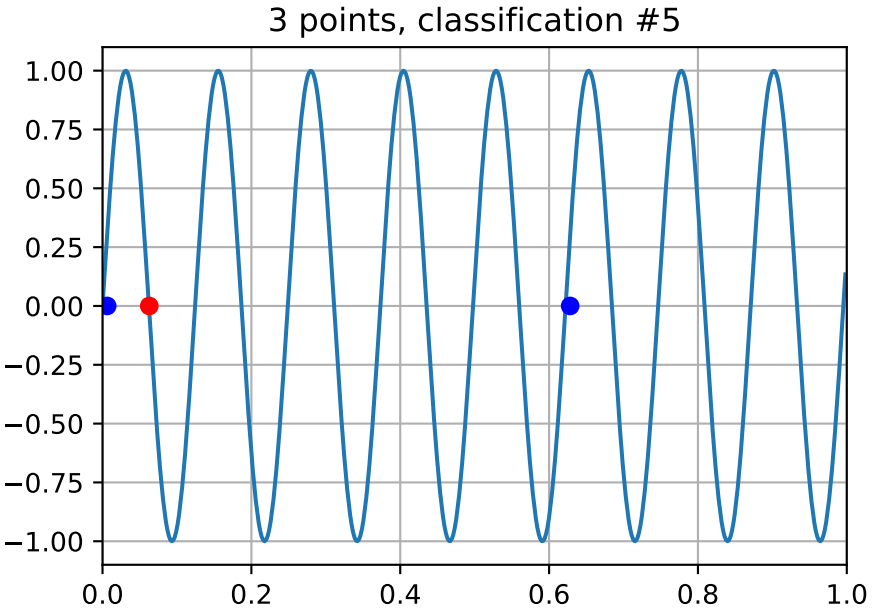

3 points, classification #0
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0.0
0.2
0.4
0.6
0.8
1.0

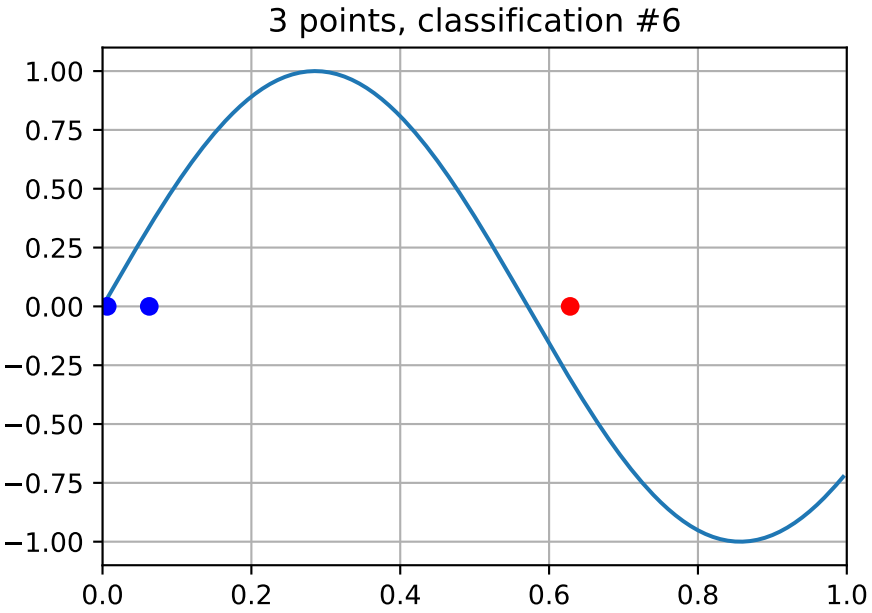

3 points, classification #1
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0.0
0.2
0.4
0.6
0.8
1.0

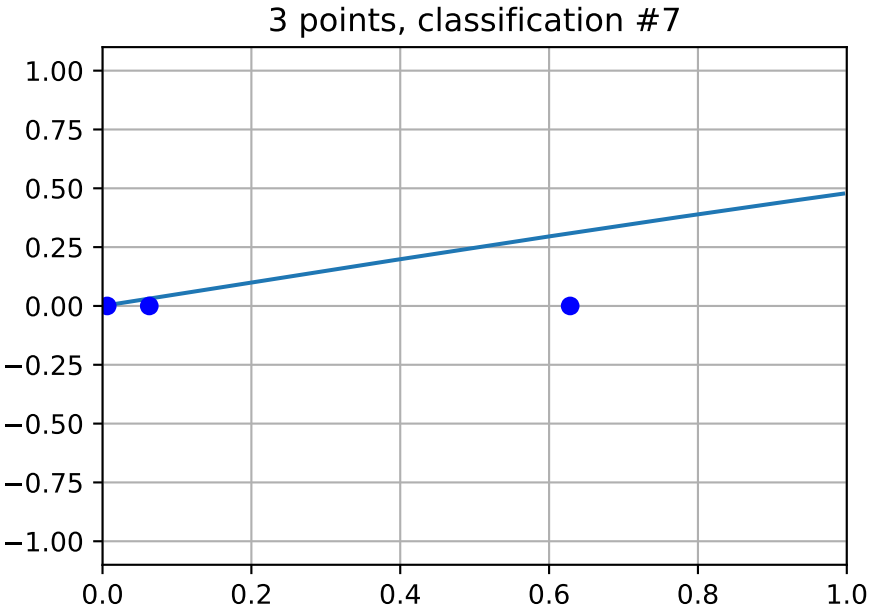

3 points, classification #2
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0.0
0.2
0.4
0.6
0.8
1.0


3 points, classification #3
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0.0
0.2
0.4
0.6
0.8
1.0


3 points, classification #4
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0.0
0.2
0.4
0.6
0.8
1.0


3 points, classification #5
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0.0
0.2
0.4
0.6
0.8
1.0


3 points, classification #6
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0.0
0.2
0.4
0.6
0.8
1.0


3 points, classification #7
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0.0
0.2
0.4
0.6
0.8
1.0

29 Complexité de Rademacher

Une borne sur l'erreur de généralisation permet de garantir les performances d'un modèle à partir de son comportement sur la base d'apprentissage. Celle-ci implique généralement une mesure de la complexité ou de la capacité de la classe de modèles $\mathcal{F}$ considérée. En particulier :

- la borne pour les classes finies se contente de compter le nombre de modèles inclut dans $\mathcal{F}$;
- la borne de Vapnik-Chervonenkis traite les classes de fonctions infinies en comptant plutôt le nombre de classifications que peuvent produire les modèles (soit directement par la fonction de croissance, soit de manière résumée par la VC dimension).

La complexité de Rademacher offre une alternative qui mesure plus finement la capacité des modèles lorsque ceux-ci sont basés sur une fonction à valeurs réelles, comme les classifieurs à marge.

La **complexité de Rademacher empirique** d'une classe de fonctions $\mathcal{F}$ de $\mathcal{X} \to \mathbb{R}$ relative à un ensemble $\mathbf{x}_n = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) \in \mathcal{X}^m$ fixé est

$$\hat{\mathcal{R}}_n(\mathcal{F}) = \mathbb{E} \sup_{f \in \mathcal{F}} \frac{1}{n} \sum_{i=1}^{n} \sigma_i f(\boldsymbol{x}_i)$$

où l'espérance est calculée par rapport à $\sigma_n = (\sigma_i)_{1 \leq i \leq n}$ qui est une séquence de n variables aléatoires indépendantes correspondant à des signes aléatoires : $P(\sigma_i = 1) = P(\sigma_i = -1) = 1/2$.

La **complexité de Rademacher** de $\mathcal{F}$ correspond à l'espérance de la complexité empirique par rapport à l'ensemble de points $\mathbf{X}_n$:

$$\mathcal{R}_n(\mathcal{F}) = \mathbb{E}\hat{\mathcal{R}}_n(\mathcal{F}) = \mathbb{E} \sup_{f \in \mathcal{F}} \frac{1}{n} \sum_{i=1}^{n} \sigma_i f(\boldsymbol{X}_i)$$

Cette complexité peut s'interpréter ainsi : pour une séquence de signes σ_n, le maximum de la somme est obtenu avec une fonction $f \in \mathcal{F}$ qui donne les mêmes signes, $\text{signe}(f(\boldsymbol{x}_i)) = \sigma_i$, avec la plus grande amplitude possible. Si pour chaque séquence de signes σ_n une telle fonction est trouvée, alors la complexité de Rademacher de $\mathcal{F}$ sera grande, car l'ensemble $\mathcal{F}$ peut s'aligner avec n'importe quelle séquence de signes aléatoires. En revanche, si $\mathcal{F}$ ne peut pas produire certaines séquences de signe, ou uniquement avec de petites amplitudes, alors la moyenne sur les séquences (l'espérance ci-dessus) sera plus faible.

29.1 Borne à base de complexité de Rademacher

Soit $\mathcal{L}$ une classe de fonctions de $\mathcal{Z}$ dans $[0,1]$. Soit $Z \in \mathcal{Z}$ une variable aléatoire et $(Z_i)_{1\leq i\leq n}$ un échantillon de n copies indépendantes de Z. Alors, avec une probabilité d'au moins $1-\delta$,

$$\forall \ell \in \mathcal{L}, \quad \mathbb{E}\ell(Z) \leq \frac{1}{n}\sum_{i=1}^{n}\ell(Z_i) + 2\mathcal{R}_n(\mathcal{L}) + \sqrt{\frac{\ln\frac{1}{\delta}}{2n}}$$

$$\forall \ell \in \mathcal{L}, \quad \mathbb{E}\ell(Z) \leq \frac{1}{n}\sum_{i=1}^{n}\ell(Z_i) + 2\hat{\mathcal{R}}_n(\mathcal{L}) + 3\sqrt{\frac{\ln\frac{2}{\delta}}{2n}}$$

La seconde borne basée sur la complexité *empirique* dépend des données et non de la distribution générale de Z. Elle permet de calculer des bornes plus serrées en pratique en évitant une analyse au pire cas, car en général la complexité de Rademacher est simplement bornée par

$$\mathcal{R}_n(\mathcal{L}) = \mathbb{E}_{\mathbf{Z}_n}\hat{\mathcal{R}}_n(\mathcal{L}) \leq \sup_{\mathbf{z}_n\in\mathcal{Z}^n}\hat{\mathcal{R}}_n(\mathcal{L})$$

Preuve

Nous allons montrer une forme plus générale pour une classe $\mathcal{L}$ de fonctions à valeur dans $[0,M]$: avec une probabilité d'au moins $1-\delta$,

$$\forall \ell \in \mathcal{L}, \quad \mathbb{E}\ell(Z) \leq \frac{1}{n}\sum_{i=1}^{n}\ell(Z_i) + 2\mathcal{R}_n(\mathcal{L}) + M\sqrt{\frac{\ln\frac{1}{\delta}}{2n}}$$

ce qui est équivalent à

$$\sup_{\ell\in\mathcal{L}}\left(\mathbb{E}\ell(Z) - \frac{1}{n}\sum_{i=1}^{n}\ell(Z_i)\right) \leq 2\mathcal{R}_n(\mathcal{L}) + M\sqrt{\frac{\ln\frac{1}{\delta}}{2n}}$$

Etape 1 (concentration) : montrons que

$$\sup_{\ell\in\mathcal{L}}\left(\mathbb{E}\ell(Z) - \frac{1}{n}\sum_{i=1}^{n}\ell(Z_i)\right) \leq \mathbb{E}\sup_{\ell\in\mathcal{L}}\left(\mathbb{E}\ell(Z) - \frac{1}{n}\sum_{i=1}^{n}\ell(Z_i)\right) + M\sqrt{\frac{\ln\frac{1}{\delta}}{2n}}$$

Pour cela, nous définissons

$$g(Z_1,\ldots,Z_n) = \sup_{\ell\in\mathcal{L}}\left(\mathbb{E}\ell(Z) - \frac{1}{n}\sum_{i=1}^{n}\ell(Z_i)\right)$$

et notons que, pour des fonctions ℓ à valeur dans $[0,M]$, remplacer une seule variable Z_i par Z_i' ne peut pas changer la valeur de g de plus de M/n. Ainsi, il est possible d'appliquer l'inégalité des différences bornées à g avec des constantes $c_i = M/n$:

$$P\left\{g(Z_1,\ldots,Z_n) - \mathbb{E}g(Z_1,\ldots,Z_n) \geq \epsilon\right\} \leq \exp\left(\frac{-2n\epsilon^2}{M^2}\right).$$

Posons $\delta = \exp\left(\frac{-2n\epsilon^2}{M^2}\right)$, alors nous avons

$$\epsilon = \sqrt{\frac{M^2 \ln \frac{1}{\delta}}{2n}} = M\sqrt{\frac{\ln \frac{1}{\delta}}{2n}}$$

et une probabilité d'au moins $1-\delta$ que $g(Z_1, \ldots, Z_n) \leq \mathbb{E}g(Z_1, \ldots, Z_n) + \epsilon$.
Etape 2 (symétrisation) : nous allons introduire la symétrisation avec un échantillon fantôme, $\mathbf{Z}'_n = (Z'_i)_{1\leq i\leq n}$ indépendant de $\mathbf{Z}_n = (Z_i)_{1\leq i\leq n}$ et distribué exactement comme celui-ci, c'est-à-dire avec des Z'_i qui sont des copies indépendantes de Z. Ainsi, par linéarité de l'espérance :

$$\mathbb{E}\left[\frac{1}{n}\sum_{i=1}^{n}\ell(Z'_i)\right] = \frac{1}{n}\sum_{i=1}^{n}\mathbb{E}\ell(Z'_i) = \frac{1}{n}\sum_{i=1}^{n}\mathbb{E}\ell(Z) = \mathbb{E}\ell(Z)$$

et

$$\begin{aligned}
\mathbb{E}\sup_{\ell\in\mathcal{L}}\left(\mathbb{E}\ell(Z) - \frac{1}{n}\sum_{i=1}^{n}\ell(Z_i)\right) &= \mathbb{E}\sup_{\ell\in\mathcal{L}}\left(\mathbb{E}\left[\frac{1}{n}\sum_{i=1}^{n}\ell(Z'_i)\right] - \frac{1}{n}\sum_{i=1}^{n}\ell(Z_i)\right) \\
&= \mathbb{E}\sup_{\ell\in\mathcal{L}}\left(\mathbb{E}\left[\frac{1}{n}\sum_{i=1}^{n}\ell(Z'_i) - \frac{1}{n}\sum_{i=1}^{n}\ell(Z_i) \;\middle|\; \mathbf{Z}_n\right]\right) \\
&= \mathbb{E}\sup_{\ell\in\mathcal{L}}\left(\mathbb{E}\left[\frac{1}{n}\sum_{i=1}^{n}(\ell(Z'_i) - \ell(Z_i)) \;\middle|\; \mathbf{Z}_n\right]\right) \\
&\leq \mathbb{E}\mathbb{E}\left[\sup_{\ell\in\mathcal{L}}\left(\frac{1}{n}\sum_{i=1}^{n}(\ell(Z'_i) - \ell(Z_i))\right) \;\middle|\; \mathbf{Z}_n\right] \\
&= \mathbb{E}\sup_{\ell\in\mathcal{L}}\left(\frac{1}{n}\sum_{i=1}^{n}(\ell(Z'_i) - \ell(Z_i))\right)
\end{aligned}$$

où l'avant dernière ligne utilise l'inégalité de Jensen et la dernière le théorème de l'espérance totale.
Etape 3 : introduction des variables de Rademcaher σ_i.
Puisque Z_i et Z'_i sont indépendant et identiquement distribuées, la variable aléatoire $D_i = \ell(Z'_i) - \ell(Z_i)$ est symétrique, c'est-à-dire que $-D_i$ a exactement la même loi de probabilité. De même, pour $\sigma_i \in \{-1, +1\}$, $\sigma_i D_i = \pm D_i$ possède aussi la même loi et il est possible de remplacer D_i par $\sigma_i D_i$ dans le calcul de l'espérance sans changer le résultat. Ainsi, calculer la moyenne sur toutes les séquences σ_n de signes possibles ne

change pas non plus le résutlat et

$$\begin{aligned}
&\mathbb{E}\sup_{\ell\in\mathcal{L}}\left(\frac{1}{n}\sum_{i=1}^{n}(\ell(Z_i')-\ell(Z_i))\right)\\
&=\mathbb{E}\left[\sup_{\ell\in\mathcal{L}}\left(\frac{1}{n}\sum_{i=1}^{n}\sigma_i(\ell(Z_i')-\ell(Z_i))\right)\right]\\
&\leq\mathbb{E}\left[\sup_{\ell\in\mathcal{L}}\frac{1}{n}\sum_{i=1}^{n}\sigma_i\ell(Z_i')+\sup_{f\in\mathcal{F}}\frac{1}{n}\sum_{i=1}^{n}-\sigma_i\ell(Z_i)\right]\\
&\leq\mathbb{E}\left[\sup_{\ell\in\mathcal{L}}\frac{1}{n}\sum_{i=1}^{n}\sigma_i\ell(Z_i')\right]+\mathbb{E}_{\boldsymbol{X}_n\boldsymbol{X}_n'\boldsymbol{\sigma}_n}\left[\sup_{\ell\in\mathcal{L}}\frac{1}{n}\sum_{i=1}^{n}-\sigma_i\ell(Z_i)\right]\\
&\leq\mathbb{E}\left[\sup_{f\in\mathcal{F}}\frac{1}{n}\sum_{i=1}^{n}\sigma_i\ell(Z_i)\right]+\mathbb{E}\left[\sup_{\ell\in\mathcal{L}}\frac{1}{n}\sum_{i=1}^{n}\sigma_i\ell(Z_i)\right]\\
&\leq 2\mathbb{E}\sup_{\ell\in\mathcal{L}}\frac{1}{n}\sum_{i=1}^{n}\sigma_i\ell(Z_i)=2\mathcal{R}_n(\mathcal{F})
\end{aligned}$$

où nous avons utilisé le fait que $\sup_{a,b}(a+b)\leq\sup_a a+\sup_b b$, et que $\sigma_i\ell(Z_i')$ et $-\sigma_i\ell(Z_i)$ ont la même loi que $\sigma_i\ell(Z_i)$.

Preuve de la version empirique

Nous prouvons ici la deuxième borne ci-dessus, dans laquelle la complexité de Rademacher est remplacée par sa version empirique.
En utilisant l'inégalité des différences bornées à la complexité de Rademacher empirique avec des constantes $c_i=M/n$, nous savons que celle-ci se concentre autour de son espérance qui n'est autre que la complexité de Rademacher : avec une probabilité d'au moins $1-\delta_2$,

$$\mathcal{R}_n(\mathcal{F})=\mathbb{E}\hat{\mathcal{R}}_n(\mathcal{F})\leq\hat{\mathcal{R}}_n(\mathcal{F})+M\sqrt{\frac{\ln\frac{1}{\delta_2}}{2n}}$$

Cependant, avant de pouvoir insérer ce résultat dans la première borne, il faut s'assurer que les deux soient satisfaits simultanément avec une forte probabilité. Pour cela, on applique la première borne avec $\delta_1=\delta/2$ et le résultat ci-dessus avec $\delta_2=\delta/2$. Alors, la probabilité qu'ils soient tous les deux valides simultanément est d'au moins

$$(1-\delta_1)(1-\delta_2)=1-\delta_1-\delta_2+\delta_1\delta_2=1-\delta+\frac{\delta^2}{4}\geq 1-\delta$$

29.2 Borne pour la classification (sans marge)

Pour appliquer la borne générique ci-dessus, on l'instancie avec $Z=(\boldsymbol{X},Y)\in\mathcal{X}\times\mathcal{Y}$, $n=m$ et la classe de fonctions de perte. Pour la classification, l'ensemble des classifieurs $\mathcal{F}$ contient

des fonctions de $\mathcal{X}$ dans $\mathcal{Y} = \{-1, +1\}$ et la classe de fonctions de perte est

$$\mathcal{L} = \left\{ \ell \in [0,1]^{\mathcal{X}\times\mathcal{Y}} : \ell(\boldsymbol{x}, y) = \mathbf{1}(f(\boldsymbol{x}) \neq y),\ f \in \mathcal{F} \right\}$$

Dans ce cas, le calcul de la complexité de Rademacher (empirique) donne

$$\mathcal{R}_n(\mathcal{L}) = \frac{1}{2}\mathcal{R}_n(\mathcal{F})$$

Preuve

$$\begin{aligned}
\hat{\mathcal{R}}_n(\mathcal{L}) &= \mathbb{E}_{\sigma_n} \sup_{f\in\mathcal{F}} \frac{1}{n}\sum_{i=1}^{n} \sigma_i \mathbf{1}(f(x_i) \neq y_i) = \mathbb{E}_{\sigma_n} \sup_{f\in\mathcal{F}} \frac{1}{n}\sum_{i=1}^{n} \sigma_i \frac{1 - y_i f(x_i)}{2} \\
&= \mathbb{E}_{\sigma_n} \sup_{f\in\mathcal{F}} \left(\frac{1}{2n}\sum_{i=1}^{n} \sigma_i + \frac{1}{2n}\sum_{i=1}^{n} -\sigma_i y_i f(x_i) \right) \\
&= \underbrace{\mathbb{E}_{\sigma_n} \frac{1}{2n}\sum_{i=1}^{n} \sigma_i}_{= \frac{1}{2n}\sum_{i=1}^{n} \mathbb{E}_{\sigma_i}\sigma_i = 0} + \mathbb{E}_{\sigma_n} \sup_{f\in\mathcal{F}} \frac{1}{2n}\sum_{i=1}^{n} -\sigma_i y_i f(x_i) = \mathbb{E}_{\sigma_n} \sup_{f\in\mathcal{F}} \frac{1}{2n}\sum_{i=1}^{n} -\sigma_i y_i f(x_i)
\end{aligned}$$

On montre facilement que $-\sigma_i y_i$ est distribué comme σ_i (ce sont donc des variables aléatoires identiques) :

$$P(-\sigma_i y_i = 1) = \begin{cases} P(\sigma_i = 1), & si\ y_i = -1 \\ P(\sigma_i = -1), & si\ y_i = 1 \end{cases} = 1/2 \qquad et \quad P(-\sigma_i y_i = -1) = 1/2$$

Donc

$$\hat{\mathcal{R}}_n(\mathcal{L}) = \mathbb{E}_{\sigma_n} \sup_{f\in\mathcal{F}} \frac{1}{2n}\sum_{i=1}^{n} \sigma_i f(x_i) = \frac{1}{2}\hat{\mathcal{R}}_n(\mathcal{F})$$

Cela conduit à la **borne sur le risque de classification binaire** suivante :

Avec une probabilité d'au moins $1 - \delta$,

$$\forall f \in \mathcal{F}, \quad R(f) \leq R_{emp}(f) + \mathcal{R}_m(\mathcal{F}) + \sqrt{\frac{\ln\frac{1}{\delta}}{2m}}$$

$$\forall f \in \mathcal{F}, \quad R(f) \leq R_{emp}(f) + \hat{\mathcal{R}}_m(\mathcal{F}) + 3\sqrt{\frac{\ln\frac{2}{\delta}}{2m}}$$

- Pour les classes $\mathcal{F}$ finies, $|\mathcal{F}| = N < \infty$, le Lemme de Massart donne

$$\hat{\mathcal{R}}_m(\mathcal{F}) \leq \sqrt{\frac{2\ln N}{m}}$$

et la borne ressemble beaucoup à notre première borne.

- Pour les classes $\mathcal{F}$ à VC-dimension finie, le nombre de classifications $|\mathcal{F}_S|$ remplace $|\mathcal{F}|$ et donc

$$\hat{\mathcal{R}}_m(\mathcal{F}) \leq 2\sqrt{\frac{\ln \Pi_{\mathcal{F}}(m)}{m}} \leq 2\sqrt{\frac{d_{VC} \ln \frac{em}{d_{VC}}}{m}}$$

  ce qui fournit une borne similaire à celle de Vapnik–Chervonenkis.

Lemme de Massart

Le Lemme de Massart s'exprime pour tout ensemble fini $\mathcal{A} = \{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_N\}$ de N vecteurs $\boldsymbol{a} \in \mathbb{R}^n$ bornés par $\forall \boldsymbol{a} \in \mathcal{A},\ \|\boldsymbol{a}\| \leq R$ comme

$$\hat{\mathcal{R}}_n(\mathcal{A}) = \mathbb{E}_{\sigma_n} \sup_{\boldsymbol{a} \in \mathcal{A}} \frac{1}{n} \sum_{i=1}^{n} \sigma_i a_i \leq \frac{R\sqrt{2 \log N}}{n}$$

Pour N classifieurs $\mathcal{F}$ à sorties binaires $f(\boldsymbol{x}_i) \in \{-1, +1\}$, le vecteur des sorties $\boldsymbol{a} = (f(\boldsymbol{x}_1), \ldots, f(\boldsymbol{x}_n))$ a une norme $\|\boldsymbol{a}\| = \sqrt{n} = R$. Ainsi,

$$\hat{\mathcal{R}}_n(\mathcal{F}) \leq \sqrt{\frac{2 \log N}{n}}$$

Mais si les classifieurs de $\mathcal{F}$ ne peuvent produire en réalité que $\Pi_{\mathcal{F}}(m) < |\mathcal{F}|$ classifications différentes, alors il n'existe que $N = \Pi_{\mathcal{F}}(m)$ vecteurs de sorties.

Preuve :

Soit $S_n = n\hat{\mathcal{R}}_n(\mathcal{A})$, alors pour tout $\lambda > 0$,

$$e^{\lambda S_n} = \exp\left(\mathbb{E}_{\sigma_n} \lambda \sup_{\boldsymbol{a} \in \mathcal{A}} \sum_{i=1}^{n} \sigma_i a_i\right)$$

et, puisque l'exponentielle est convexe, l'inégalité de Jensen donne

$$\begin{aligned} e^{\lambda S_n} &\leq \mathbb{E}_{\sigma_n} \exp\left(\lambda \sup_{\boldsymbol{a} \in \mathcal{A}} \sum_{i=1}^{n} \sigma_i a_i\right) \\ &= \mathbb{E}_{\sigma_n} \sup_{\boldsymbol{a} \in \mathcal{A}} \exp\left(\lambda \sum_{i=1}^{n} \sigma_i a_i\right) \end{aligned}$$

Puisque le maximum sur tous les $\boldsymbol{a}$ est plus petit que la somme sur tous les $\boldsymbol{a}$:

$$\begin{aligned} e^{\lambda S_n} &\leq \mathbb{E}_{\sigma_n} \sum_{\boldsymbol{a} \in \mathcal{A}} \exp\left(\lambda \sum_{i=1}^{n} \sigma_i a_i\right) \\ &= \sum_{\boldsymbol{a} \in \mathcal{A}} \mathbb{E}_{\sigma_n} \exp\left(\lambda \sum_{i=1}^{n} \sigma_i a_i\right) \\ &= \sum_{\boldsymbol{a} \in \mathcal{A}} \mathbb{E}_{\sigma_n} \prod_{i=1}^{n} e^{\lambda \sigma_i a_i} \end{aligned}$$

Les σ_i étant indépendants, l'espérance du produit est le produit des espérances :

$$\begin{aligned} e^{\lambda S_n} &\leq \sum_{\boldsymbol{a}\in\mathcal{A}} \prod_{i=1}^{n} \mathbb{E}_{\sigma_i} e^{\lambda \sigma_i a_i} \\ &= \sum_{\boldsymbol{a}\in\mathcal{A}} \prod_{i=1}^{n} \frac{e^{\lambda a_i} + e^{-\lambda a_i}}{2} \end{aligned}$$

où l'espérance par rapport à chaque $\sigma_i \in \{-1, +1\}$ a pu se calculer explicitement. Ensuite, nous utilisons l'inégalité $(e^x + e^{-x})/2 \leq e^{x^2/2}$ pour obtenir

$$\begin{aligned} e^{\lambda S_n} &\leq \sum_{\boldsymbol{a}\in\mathcal{A}} \prod_{i=1}^{n} e^{\lambda^2 a_i^2/2} \\ &= \sum_{\boldsymbol{a}\in\mathcal{A}} e^{\lambda^2 \sum_{i=1}^{n} a_i^2/2} \\ &= \sum_{\boldsymbol{a}\in\mathcal{A}} e^{\lambda^2 \|\boldsymbol{a}\|^2/2} \\ &\leq N e^{\lambda^2 R^2/2} \end{aligned}$$

Donc, en prenant le log,

$$\lambda S_n \leq \log N + \frac{\lambda^2 R^2}{2} \quad \Rightarrow \quad S_n \leq \frac{\log N}{\lambda} + \frac{\lambda R^2}{2}$$

et en choisissant $\lambda = \frac{\sqrt{2\log N}}{R}$, cela donne

$$S_n \leq R\sqrt{\frac{\log N}{2}} + R\sqrt{\frac{\log N}{2}} = R\sqrt{2\log N}$$

et donc $\hat{\mathcal{R}}_n(\mathcal{A}) \leq R\sqrt{2\log N}/n$.

29.3 Classifieurs à marge

Un classifieur à marge est un classifieur basé sur une ou plusieurs fonctions de score à valeur réelle.

- En classification binaire, un classifieur à marge retourne le signe d'une fonction à valeur réelle
$$f(x) = \text{signe}(g(x))$$
- En classification multi-classe, un classifieur à marge est basé sur C fonctions à valeurs réelles qui retournent un score par catégorie :
$$f(x) = \arg \max_{k\in\{1,\ldots,C\}} g_k(x)$$

Nous traiterons ici uniquement de la classification binaire.

29.3.1 Fonction de perte et risque à marge

Pour une marge $\gamma \geq 0$, la **fonction de perte à marge** est donnée par

$$\ell_\gamma(f,x,y) = \varphi_\gamma(yg(x)), \qquad \varphi_\gamma(u) = \begin{cases} 1, & \text{si } u \leq 0 \\ 1 - \frac{u}{\gamma}, & \text{si } u \in (0,\gamma) \\ 0, & \text{si } u \geq \gamma \end{cases}$$

et elle fournit une majoration de la perte classique de classification 0-1 $\ell(f,x,y)$:

$$\forall (f,x,y), \quad \ell_\gamma(f,x,y) \geq \ell(f,x,y)$$

car $\ell(f,x,y) = \mathbf{1}(f(x) \neq y) \leq \mathbf{1}(yg(x) \leq 0)$.

```
u = np.arange(-1,2,0.1)
l = 1.0 * ( u < 0 )
lg = (1 - u) * (1-u >= 0) * (1-u < 1) + (1-u >= 1)
plt.plot(u,l)
plt.plot(u,lg,"--")
plt.grid()
plt.xlabel("y g(x)")
plt.legend(["perte 0-1", "perte à marge"])
t=plt.title("Fonction de perte à marge")
```

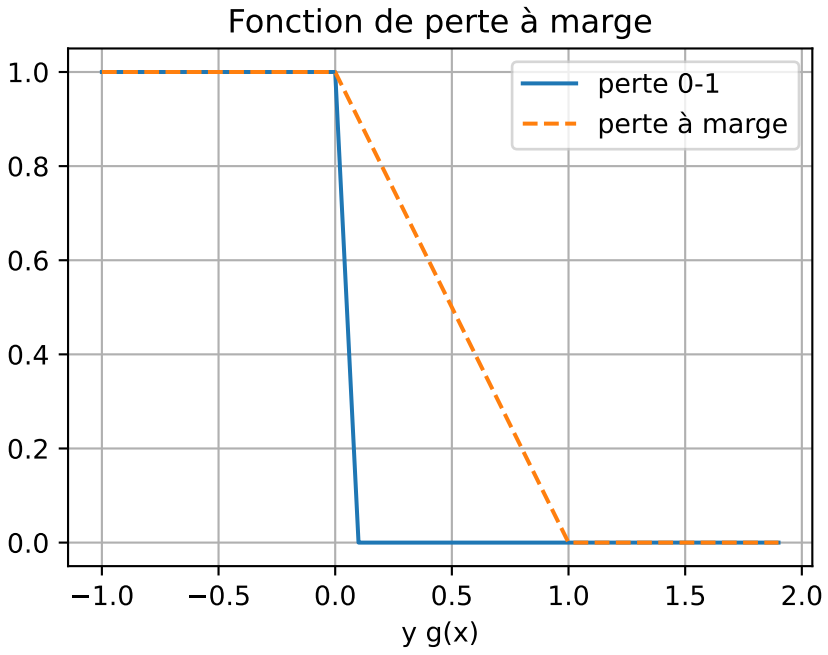


Figure 29.1

Ainsi, le **risque à marge** $R_\gamma(f)$ borne le risque de classification $R(f)$:

$$R_\gamma(f) = \mathbb{E}\ell_\gamma(f, X, Y) \geq \mathbb{E}\ell(f, X, Y) = R(f)$$

On définit aussi le risque empirique à marge :

$$R_{emp,\gamma}(f) = \frac{1}{m}\sum_{i=1}^{m} \ell_\gamma(f, X_i, Y_i)$$

et la classe de fonctions de perte à marge :

$$\mathcal{L}_\gamma = \left\{\ell \in [0,1]^{\mathcal{X}\times\mathcal{Y}} : \ell(x,y) = \ell_\gamma(f,x,y),\ f \in \mathcal{F}\right\}$$

29.3.2 Borne sur le risque à marge

L'analyse ci-dessus associée à la borne générique nous donne : avec une probabilité d'au moins $1-\delta$,

$$\forall f \in \mathcal{F}, \quad R(f) \leq R_\gamma(f) \leq R_{emp,\gamma}(f) + 2\mathcal{R}_m(\mathcal{L}_\gamma) + \sqrt{\frac{\ln\frac{1}{\delta}}{2m}}$$

et il ne reste plus qu'à relier la complexité de la classe de fonctions de perte à marge à celle des fonctions de score g sur lesquelles s'appuient les classifieurs de $\mathcal{F}$ que l'on peut écrire comme

$$\mathcal{F} = \{f : f(x) = \text{signe}(g(x)),\ g \in \mathcal{G}\}$$

Cela se fait grâce au principe de contraction suivant.

Lemme 29.1 (Principe de contraction)**.** *Soit une classe de fonctions $\mathcal{L} = \varphi \circ \mathcal{U} = \{\ell : \ell(z) = \varphi(u(z)),\ u \in \mathcal{U}\}$, définie par la composition avec une fonction $\varphi : \mathbb{R} \to \mathbb{R}$ lipschitzienne ($|\varphi(t) - \varphi(t')| \leq L|t - t'|$), alors*

$$\hat{\mathcal{R}}_m(\mathcal{L}) = \hat{\mathcal{R}}_m(\varphi \circ \mathcal{U}) \leq L\ \hat{\mathcal{R}}_m(\mathcal{U})$$

Pour la classe de fonctions de perte à marge γ, il est aisé de voir sur la Figure 29.1 que φ_γ est lipschitzienne de constante $L = 1/\gamma$.

Ainsi, avec $u(z_i) = y_i g(x_i)$,

$$\hat{\mathcal{R}}_m(\mathcal{L}_\gamma) \leq \frac{1}{\gamma}\mathbb{E}_{\sigma_n}\sup_{g\in\mathcal{G}}\frac{1}{n}\sum_{i=1}^{n}\sigma_i y_i g(x_i) = \frac{1}{\gamma}\mathbb{E}_{\sigma_n}\sup_{g\in\mathcal{G}}\frac{1}{n}\sum_{i=1}^{n}\sigma_i g(x_i) = \frac{1}{\gamma}\hat{\mathcal{R}}_m(\mathcal{G})$$

car $\sigma_i y_i$ est distribuée comme σ_i (multiplier σ_i par ± 1 ne modifie pas ses probabilités).

Donc, avec une probabilité d'au moins $1-\delta$,

$$\forall f \in \mathcal{F}, \quad R(f) \leq R_\gamma(f) \leq R_{emp,\gamma}(f) + \frac{2}{\gamma}\mathcal{R}_m(\mathcal{G}) + \sqrt{\frac{\ln\frac{1}{\delta}}{2m}}$$

Cette borne permet de justifier théoriquement l'intérêt de la maximisation de la marge γ : s'il est possible de trouver un classifieur qui conduit à un petit risque empirique à marge avec une grande marge, alors il devrait avoir de bonnes performances en généralisation. Cependant, il faut faire attention : dans l'analyse ci-dessus, la valeur de γ était supposée constante et fixée à l'avance, donc indépendante des données d'apprentissage. Pour pouvoir réellement justifier de l'utilisation d'un algorithme qui maximise la marge en fonction des données disponibles, il faut « uniformiser » la borne par rapport à ce paramètre, comme montré ci-dessous.

29.3.3 Version uniforme en γ

La version suivante s'applique quel que soit γ, et donc potentiellement pour une valeur de γ choisie après avoir vu les données de la base d'apprentissage.

Avec une probabilité d'au moins $1-\delta$,

$$\forall \gamma \in (0,1),\ \forall f \in \mathcal{F}, \quad R(f) \leq R_{emp,\gamma}(f) + \frac{4}{\gamma}\mathcal{R}_m(\mathcal{G}) + \sqrt{\frac{\ln \frac{2}{\delta}}{2m}} + \sqrt{\frac{\ln \log_2 \frac{2}{\gamma}}{m}}$$

On peut voir ici que le coût de l'uniformisation en γ est minime : le terme ajouté est petit devant le reste de la borne.

29.4 Borne pour les classifieurs linéaires régularisés

Pour les classifieurs linéaires,

$$\mathcal{G} = \{g : g(\boldsymbol{x}) = \langle \boldsymbol{w}, \boldsymbol{x} \rangle\ ,\ \|\boldsymbol{w}\| \leq \Lambda\}$$

où Λ est un hyperparamètre de régularisation qui contrôle l'amplitude de vecteur de paramètres $\boldsymbol{w}$.

Dans ce cas, la complexité de Rademacher est bornée par

$$\hat{\mathcal{R}}_m(\mathcal{G}) \leq \frac{\Lambda}{m}\sqrt{\sum_{i=1}^{m} \|x_i\|^2}$$

ou encore, si $\Lambda_x = \sup_{\boldsymbol{x} \in \mathcal{X}} \|\boldsymbol{x}\|$,

$$\hat{\mathcal{R}}_m(\mathcal{G}) \leq \frac{\Lambda \Lambda_x}{\sqrt{m}}$$

ce qui conduit à une borne sur le risque $R(f)$ en $O\left(\frac{1}{\sqrt{m}}\right)$.

Preuve

$$\begin{aligned}
\hat{\mathcal{R}}_m(\mathcal{G}) &= \mathbb{E}_{\sigma_m} \sup_{g\in\mathcal{G}} \frac{1}{m}\sum_{i=1}^m \sigma_i g(\boldsymbol{x}_i) \\
&= \mathbb{E}_{\sigma_m} \sup_{\|\boldsymbol{w}\|\leq\Lambda} \frac{1}{m}\sum_{i=1}^m \sigma_i \langle \boldsymbol{w}, \boldsymbol{x}_i\rangle \\
&= \mathbb{E}_{\sigma_m} \sup_{\|\boldsymbol{w}\|\leq\Lambda} \frac{1}{m}\left\langle \boldsymbol{w}, \sum_{i=1}^m \sigma_i \boldsymbol{x}_i\right\rangle \\
&\leq \mathbb{E}_{\sigma_m} \sup_{\|\boldsymbol{w}\|\leq\Lambda} \frac{1}{m}\|\boldsymbol{w}\| \left\|\sum_{i=1}^m \sigma_i \boldsymbol{x}_i\right\| \\
&\leq \frac{\Lambda}{m}\mathbb{E}_{\sigma_m} \left\|\sum_{i=1}^m \sigma_i \boldsymbol{x}_i\right\|
\end{aligned}$$

où nous avons utilisé l'inégalité de Cauchy-Schwarz dans l'avant dernière ligne.
La fonction racine étant concave, l'inégalité de Jensen donne

$$\mathbb{E}_{\sigma_m} \left\|\sum_{i=1}^m \sigma_i \boldsymbol{x}_i\right\| = \mathbb{E}_{\sigma_m} \sqrt{\left\|\sum_{i=1}^m \sigma_i \boldsymbol{x}_i\right\|^2} \leq \sqrt{\mathbb{E}_{\sigma_m} \left\|\sum_{i=1}^m \sigma_i \boldsymbol{x}_i\right\|^2}$$

où

$$\begin{aligned}
\mathbb{E}_{\sigma_m} \left\|\sum_{i=1}^m \sigma_i \boldsymbol{x}_i\right\|^2 &= \mathbb{E}_{\sigma_m} \left\langle \sum_{i=1}^m \sigma_i \boldsymbol{x}_i, \sum_{j=1}^m \sigma_j \boldsymbol{x}_j\right\rangle = \mathbb{E}_{\sigma_m} \sum_{i=1}^m\sum_{j=1}^m \sigma_i\sigma_j \langle \boldsymbol{x}_i, \boldsymbol{x}_j\rangle \\
&= \mathbb{E}_{\sigma_m} \sum_{i=1}^m \left[\sigma_i^2 \langle \boldsymbol{x}_i, \boldsymbol{x}_i\rangle + \sum_{j\neq i} \sigma_i\sigma_j \langle \boldsymbol{x}_i, \boldsymbol{x}_j\rangle\right] \\
&= \sum_{i=1}^m \left[\langle \boldsymbol{x}_i, \boldsymbol{x}_i\rangle + \sum_{j\neq i} \langle \boldsymbol{x}_i, \boldsymbol{x}_j\rangle \mathbb{E}_{\sigma_i,\sigma_j} \sigma_i\sigma_j\right]
\end{aligned}$$

Pour $j \neq i$, σ_i est indépendant de σ_j et donc l'espérance se décompose en produit : $\mathbb{E}_{\sigma_i,\sigma_j}\sigma_i\sigma_j = (\mathbb{E}_{\sigma_i}\sigma_i)(\mathbb{E}_{\sigma_j}\sigma_j) = 0$.
Donc

$$\mathbb{E}_{\sigma_m} \left\|\sum_{i=1}^m \sigma_i \boldsymbol{x}_i\right\|^2 = \sum_{i=1}^m \langle \boldsymbol{x}_i, \boldsymbol{x}_i\rangle = \sum_{i=1}^m \|\boldsymbol{x}_i\|^2 \quad \Rightarrow \quad \hat{\mathcal{R}}_m(\mathcal{G}) \leq \frac{\Lambda}{m}\sqrt{\sum_{i=1}^m \|x_i\|^2}$$

Contrairement à la borne de Vapnik-Chervonenkis, cette borne est **indépendante de la dimension** d de $\boldsymbol{x}$. Elle est donc applicable pour la classification non linéaire avec une projection $\phi : \boldsymbol{x} \mapsto \phi(\boldsymbol{x})$ en grande dimension, ou en dimension infinie comme avec les SVM à noyau gaussien ; à condition bien sûr de contrôler la complexité par $\Lambda \geq \|\boldsymbol{w}\|$.

Ainsi, cette étude théorique permet de justifier les techniques de régularisation basées sur la minimisation de $\|\boldsymbol{w}\|$.

partie V

Réduction de dimension

30 Analyse en composantes principales (ACP)

L'analyse en composantes principales (ACP ou *Principal Component Analysis, PCA*) est une technique de réduction de dimension qui vise à créer une nouvelle représentation des données en plus petite dimension.

30.1 Analyse statistique

On suppose ici que le vecteur $\boldsymbol{X} \in \mathbb{R}^d$ contient d variables aléatoires centrées, c'est-à-dire de moyenne nulle : $\mathbb{E}X_k = 0,\ k = 1, \dots, d$.

Le but est de trouver un nouveau vecteur aléatoire $\boldsymbol{Z} \in \mathbb{R}^p$ représentant $\boldsymbol{X} \in \mathbb{R}^d$ avec $p < d$ composantes et tel que

- $\boldsymbol{Z}$ est une transformation linéaire de $\boldsymbol{X}$:

$$\boldsymbol{Z} = \boldsymbol{U}\boldsymbol{X} \quad ou \quad \begin{bmatrix} Z_1 \\ \vdots \\ Z_p \end{bmatrix} = \begin{bmatrix} — \ \boldsymbol{u}_1^\top \ — \\ \vdots \\ — \ \boldsymbol{u}_p^\top \ — \end{bmatrix} \begin{bmatrix} X_1 \\ \vdots \\ X_d \end{bmatrix}$$

- les composantes Z_k sont décorrélées : $\mathbb{E}[Z_j Z_k] = 0,\ \forall j \neq k$;
- la variance des Z_k est maximisée et $Var(Z_1) \geq Var(Z_2) \geq \cdots \geq Var(Z_p)$;
- les lignes $\boldsymbol{u}_k^\top$ de $\boldsymbol{U}$ sont normalisées telles que $\|\boldsymbol{u}_k\| = 1$.

Cela s'obtient simplement en choisissant les vecteurs $\boldsymbol{u}_k$ comme les p vecteurs propres de la matrice de covariance

$$\boldsymbol{\Sigma} = \mathbb{E}[\boldsymbol{X}\boldsymbol{X}^\top]$$

associés aux p plus grandes valeurs propres λ_k.

De plus, les valeurs propres indiquent la variance de chaque composante : $\lambda_k = Var(Z_k)$.

30.1.1 Obtention de la première composante principale

Trouver $Z_1 = \boldsymbol{u}_1^\top \boldsymbol{X}$ avec $Var(Z_1) = \mathbb{E}[(Z_1 - \mathbb{E}Z_1)^2]$ maximale et un $\boldsymbol{u}_1$ normé revient à résoudre le problème d'optimisation

$$\boldsymbol{u}_1 = \arg\max_{\boldsymbol{u} \in \mathbb{R}^d} Var(\boldsymbol{u}^\top \boldsymbol{X}), \qquad s.c. \quad \boldsymbol{u}^\top \boldsymbol{u} = 1$$

La fonction objectif de ce problème peut se reformuler à partir de la matrice de covariance : pour X_j centrée, la moyenne de Z_1 est

$$\mathbb{E}[\boldsymbol{u}^\top \boldsymbol{X}] = \mathbb{E}\left[\sum_{j=1}^{d} u_j X_j\right] = \sum_{j=1}^{d} u_j \mathbb{E}[X_j] = 0$$

et donc

$$Var(\boldsymbol{u}^\top \boldsymbol{X}) = \mathbb{E}[(\boldsymbol{u}^\top \boldsymbol{X})^2] = \mathbb{E}[(\boldsymbol{u}^\top \boldsymbol{X})(\boldsymbol{X}^\top \boldsymbol{u})] = \boldsymbol{u}^\top \mathbb{E}[\boldsymbol{X}\boldsymbol{X}^\top]\boldsymbol{u} = \boldsymbol{u}^\top \boldsymbol{\Sigma} \boldsymbol{u}$$

La problème d'optimisation peut être résolu par dualité lagrangienne. Le lagrangien du problème est (avec λ une variable duale) :

$$L(\boldsymbol{u}, \lambda) = \boldsymbol{u}^\top \boldsymbol{\Sigma} \boldsymbol{u} + \lambda(1 - \boldsymbol{u}^\top \boldsymbol{u})$$

et sa maximisation est obtenue en un point où sa dérivée par rapport à $\boldsymbol{u}$ est nulle :

$$\frac{\partial L(\boldsymbol{u}, \lambda)}{\partial \boldsymbol{u}} = 2\boldsymbol{\Sigma}\boldsymbol{u} - 2\lambda \boldsymbol{u} = 0 \quad \Rightarrow \quad \boldsymbol{\Sigma}\boldsymbol{u} = \lambda \boldsymbol{u}$$

Ainsi, $\boldsymbol{u}_1$ est bien un vecteur propre de $\boldsymbol{\Sigma}$ associé à la valeur propre λ, et avec la contrainte $\boldsymbol{u}^\top \boldsymbol{u} = 1$:

$$Var(Z_1) = Var(\boldsymbol{u}^\top \boldsymbol{X}) = \boldsymbol{u}^\top \boldsymbol{\Sigma} \boldsymbol{u} = \lambda \boldsymbol{u}^\top \boldsymbol{u} = \lambda$$

Donc $\boldsymbol{u}_1$ est le vecteur propre associé à la plus grande valeur propre λ_1.

30.1.2 Obtention des composantes principales suivantes

Pour trouver $Z_2 = \boldsymbol{u}_2^\top \boldsymbol{X}$, on peut vérifier que le vecteur $\boldsymbol{u}_2$ de $\boldsymbol{\Sigma}$ avec la seconde plus grande valeur propre $\lambda_2 < \lambda_1$ satisfait aux contraintes :

- Comme pour Z_1, $Var(Z_2) = \lambda_2$ sera ainsi maximisée ;
- $\boldsymbol{u}_2$ sera automatiquement normalisé avec $\|\boldsymbol{u}_2\| = 1$;
- Pour ce qui est de la nouvelle contrainte « Z_1 et Z_2 décorrélées $\Leftrightarrow \mathbb{E}[Z_1 Z_2] = 0$ », nous pouvons écrire

  $$\mathbb{E}[Z_1 Z_2] = \mathbb{E}[(\boldsymbol{u}_1^\top \boldsymbol{X})(\boldsymbol{u}_2^\top \boldsymbol{X})] = \mathbb{E}[(\boldsymbol{u}_1^\top \boldsymbol{X})(\boldsymbol{X}^\top \boldsymbol{u}_2)] = \boldsymbol{u}_1^\top \boldsymbol{\Sigma} \boldsymbol{u}_2$$

  Puisque $\boldsymbol{u}_1$ vecteur propre, $\lambda_1 \boldsymbol{u}_1^\top = \boldsymbol{u}_1^\top \boldsymbol{\Sigma}^\top = \boldsymbol{u}_1^\top \boldsymbol{\Sigma}$ (car la matrice de covariance est symétrique) et donc

  $$\mathbb{E}[Z_1 Z_2] = u_1^\top \boldsymbol{\Sigma} u_2 = \lambda_1 u_1^\top u_2$$

  Ainsi, si $\boldsymbol{u}_2$ est aussi vecteur propre, $\boldsymbol{\Sigma}\boldsymbol{u}_2 = \lambda_2 \boldsymbol{u}_2$, et

  $$\mathbb{E}[Z_1 Z_2] = u_1^\top \boldsymbol{\Sigma} u_2 = \lambda_2 u_1^\top u_2$$
  $$\Rightarrow \lambda_1 \boldsymbol{u}_1^\top \boldsymbol{u}_2 = \lambda_2 \boldsymbol{u}_1^\top \boldsymbol{u}_2 \;\Rightarrow\; \boldsymbol{u}_1^\top \boldsymbol{u}_2 = 0 \quad (si\ \lambda_1 \neq \lambda_2)$$

  et donc $\mathbb{E}[Z_1 Z_2] = 0$.

En résumé, *les vecteurs propres d'une matrice $\boldsymbol{\Sigma}$ symétrique (et réelle) sont orthogonaux et donnent des projections décorrélées.*

Il suffit ensuite d'itérer ce raisonnement pour montrer que les composantes suivantes sont les vecteurs propres associés aux plus grandes valeurs propres restantes.

30.2 En pratique

En pratique, nous ne travaillons pas à partir d'une variable aléatoire, mais à partir d'un jeu de données composé de n exemples $\boldsymbol{x}_i \in \mathbb{R}^d$ (tirages du vecteur aléatoire) stockés dans un matrice

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x}_1^\top \\ \vdots \\ \boldsymbol{x}_n^\top \end{bmatrix} \in \mathbb{R}^{n \times d}$$

Plusieurs questions peuvent se poser :

1. Les données ne sont pas nécessairement centrées. Dans ce cas, il faut centrer les données (en soustrayant la moyenne) avant d'appliquer la méthode :

$$\boldsymbol{X} \leftarrow \boldsymbol{X} - \boldsymbol{1}\boldsymbol{1}^\top \boldsymbol{X}$$

2. La distribution de $\boldsymbol{X}$ et la matrice de covariance $\boldsymbol{\Sigma}$ sont inconnues. Il faut donc appliquer la méthode à partir d'une estimation de $\boldsymbol{\Sigma}$: la matrice de covariance empirique

$$\boldsymbol{C} = \frac{1}{n}\boldsymbol{X}^\top \boldsymbol{X} = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{x}_i \boldsymbol{x}_i^\top$$

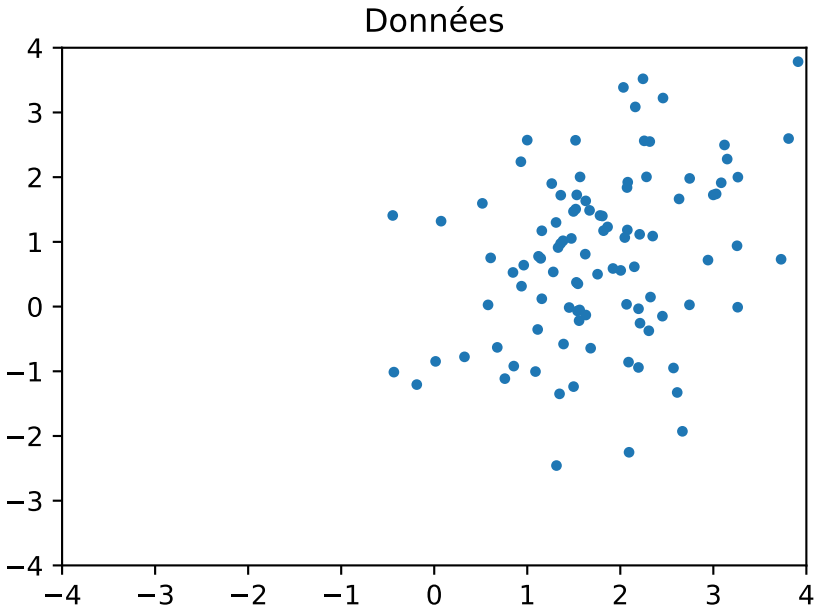
   Les vecteurs $\boldsymbol{u}_k$ sont donc les vecteurs propres de $\boldsymbol{C}$ au lieu de $\boldsymbol{\Sigma}$ (ils peuvent aussi être calculés directement à partir de $\boldsymbol{X}^\top \boldsymbol{X}$ qui possède les mêmes vecteurs propres que $\boldsymbol{C}$ et dont les valeurs propres sont juste multipliées par n).
3. La charge de calcul peut devenir importante en grande dimension. Dans ce cas, si $n < d$, il est préférable de passer par la SVD fine ou tronquée de $\boldsymbol{X}$ pour récupérer les vecteurs propres $\boldsymbol{u}_k$ de $\boldsymbol{X}^\top \boldsymbol{X}$ comme les p premiers vecteurs singuliers à droite de $\boldsymbol{X}$.
4. Il faut fare attention au sens des matrices pour obtenir la nouvelle représentation $\boldsymbol{Z} \in \mathbb{R}^{n \times p}$ du jeu de données $\boldsymbol{X} \in \mathbb{R}^{n \times d}$:

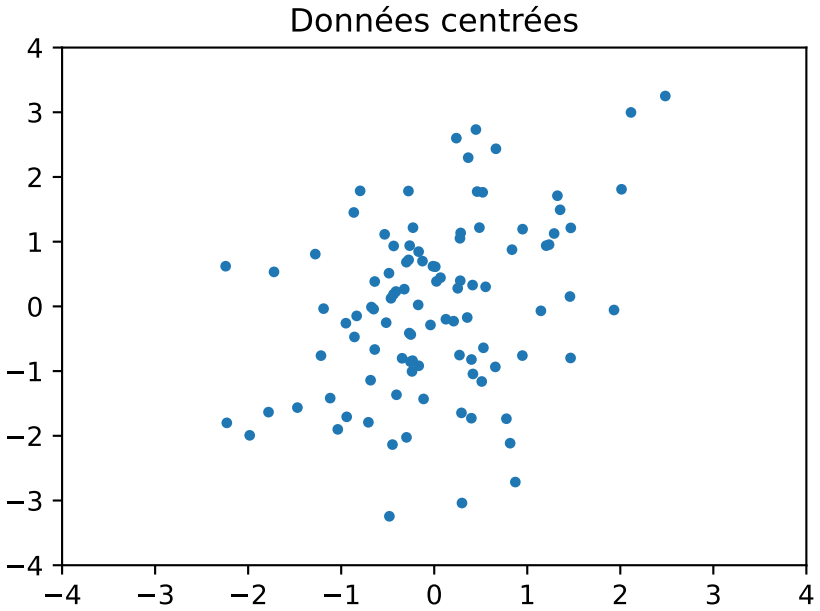
$$\boldsymbol{Z} = \boldsymbol{X}\boldsymbol{U}^\top \qquad ou \qquad \boldsymbol{Z}^\top = \boldsymbol{U}\boldsymbol{X}^\top$$

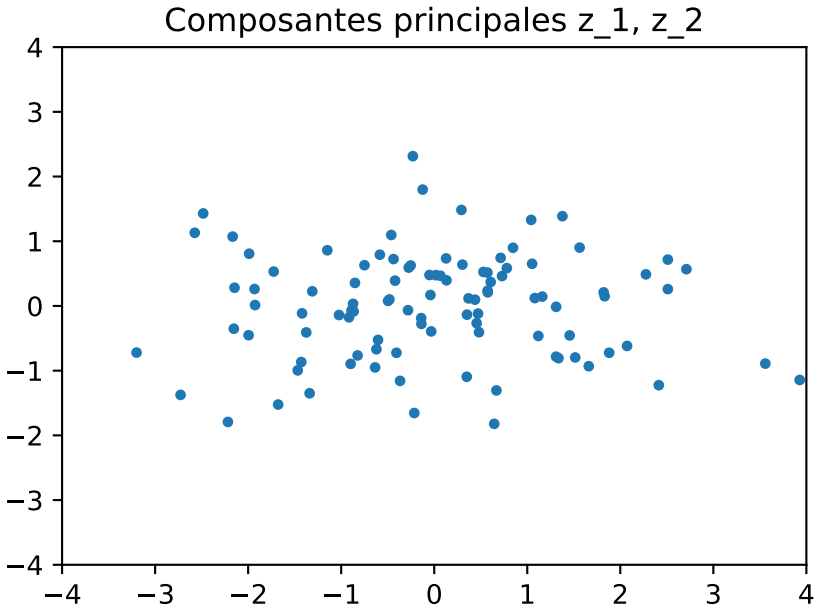
$$\begin{bmatrix} \boldsymbol{z}_1^\top \\ \vdots \\ \boldsymbol{z}_n^\top \end{bmatrix} = \begin{bmatrix} \boldsymbol{x}_1^\top \\ \vdots \\ \boldsymbol{x}_n^\top \end{bmatrix} \begin{bmatrix} \boldsymbol{u}_1 & \dots & \boldsymbol{u}_p \end{bmatrix} \qquad ou \qquad \begin{bmatrix} \boldsymbol{z}_1 & \dots & \boldsymbol{z}_n \end{bmatrix} = \begin{bmatrix} \boldsymbol{u}_1^\top \\ \vdots \\ \boldsymbol{u}_p^\top \end{bmatrix} \begin{bmatrix} \boldsymbol{x}_1 & \dots & \boldsymbol{x}_n \end{bmatrix}$$

Exemple en 2D

```
import numpy as np
X = np.random.multivariate_normal((2,1), [[1, 0.5],[0.5, 2]], size=100)

plt.plot(X[:,0], X[:,1], ".")
plt.axis([-4, 4, -4, 4])
plt.title("Données")

mu = np.mean(X, axis=0)
X = X - mu
plt.figure()
plt.plot(X[:,0], X[:,1], ".")
plt.axis([-4, 4, -4, 4])
plt.title("Données centrées")

Lambdas,U = np.linalg.eigh(X.T @ X)
# réordonner car eigh trie par ordre croissant des valeurs propres:
# et transposer pour avoir les u_k en lignes dans U:
U = U[:,-1:-3:-1].T
Z = X @ U.T

plt.figure()
plt.plot(Z[:,0], Z[:,1], ".")
plt.axis([-4, 4, -4, 4])
t=plt.title("Composantes principales z_1, z_2")
```




Données centrées
4
3
2
1
0
−1
−2
−3
−4
−4
−3
−2
−1
0
1
2
3
4


Composantes principales z_1, z_2
4
3
2
1
0
−1
−2
−3
−4
−4
−3
−2
−1
0
1
2
3
4

31 Analyse en composantes principales kernelisée (Kernel PCA)

L'analyse en composantes principales est une technique de réduction de dimension qui vise à créer une nouvelle représentation des données en plus petite dimension par une transformation simple et linéaire. La version kernelisée permet de représenter des jeux de données plus complexes de manière plus compacte en appliquant une transformation non linéaire.

Cette transformation non linéaire est en fait construite à partir de la transformation linéaire classique de la PCA appliquée aux données préalablement projetées de manière non linéaire dans un nouvel espace de représentation. La *kernelisation* permet de bénéficier des techniques des méthodes à noyaux pour effectuer les calculs dans ce nouvel espace potentiellement en très grande dimension de manière efficace.

La méthode procède donc ainsi :

1. Projeter les données dans un nouvel espace de représentation :

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x}_1^\top \\ \vdots \\ \boldsymbol{x}_n^\top \end{bmatrix} \in \mathbb{R}^{n \times d} \qquad \mapsto \qquad \boldsymbol{\Phi} = \begin{bmatrix} \text{—} \; \phi(\boldsymbol{x}_1)^\top \; \text{—} \\ \vdots \\ \text{—} \; \phi(\boldsymbol{x}_n)^\top \; \text{—} \end{bmatrix} \in \mathbb{R}^{n \times d_\phi}$$

2. Appliquer la PCA dans le nouvel espace :

$$\boldsymbol{Z} = \boldsymbol{\Phi}\boldsymbol{U}^\top \qquad ou \qquad \boldsymbol{z}_i = \boldsymbol{U}\phi(\boldsymbol{x}_i),\ i = 1, \dots, n$$

   avec les lignes de $\boldsymbol{U}$ contenant les vecteurs propres de $\boldsymbol{\Phi}^\top\boldsymbol{\Phi}$ associés aux plus grandes valeurs propres.

Cela crée néanmoins deux problèmes :

- Même si les $\boldsymbol{x}_i$ sont centrés, les $\phi(\boldsymbol{x}_i)$ ne le sont pas forcément dans le nouvel espace. Il faut donc appliquer la PCA sur les $\phi(\boldsymbol{x}_i)$ centrés :

$$\boldsymbol{z}_i = \boldsymbol{U}\overline{\phi}(\boldsymbol{x}_i) \qquad avec\ \boldsymbol{\Phi} = \begin{bmatrix} \overline{\phi}(\boldsymbol{x}_1)^\top \\ \vdots \\ \overline{\phi}(\boldsymbol{x}_n)^\top \end{bmatrix}, \quad \overline{\phi}(\boldsymbol{x}_i) = \phi(\boldsymbol{x}_i) - \phi_0, \quad \boldsymbol{\phi}_0 = \frac{1}{n}\sum_{i=1}^{n}\phi(\boldsymbol{x}_i)$$

- Le nouvel espace peut être en très grande dimension et la complexité $O(d_\phi^3)$ de la PCA devient problématique. Mais si $n \ll d_\phi$, la version duale de la PCA permet de s'en sortir.

- Le complexité peut encore être trop importante pour le calcul explcite des projections $\phi(\boldsymbol{x}_i)$ et de leurs produits scalaires. Pour cela, les méthodes à noyaux fournissent une alternative efficace par calcul implicite.

Au final, comme démontré et détaillé ci-dessous, l'**algorithme Kernel PCA** s'implémente avec les grandes étapes suivantes :

1. Choisir une fonction de noyau K et calculer la matrice de noyau $\boldsymbol{K}$.
2. Calculer la matrice de noyau centrée $\overline{\boldsymbol{K}}$.
3. Calculer les vecteurs propres $\boldsymbol{w}_k$ de $\overline{\boldsymbol{K}}$ associés aux p plus grandes valeurs propres λ_k.
4. Normaliser les vecteurs propres :

$$\boldsymbol{W} = \begin{bmatrix} \boldsymbol{w}_1^\top / \sqrt{\lambda_1} \\ \vdots \\ \boldsymbol{w}_p^\top / \sqrt{\lambda_p} \end{bmatrix}$$

5. Calculer la nouvelle représentation des données :

$$\boldsymbol{z}_i = \boldsymbol{U}\overline{\phi}(\boldsymbol{x}_i) = \boldsymbol{W}\boldsymbol{\Phi}\overline{\phi}(\boldsymbol{x}_i) = \boldsymbol{W} \begin{bmatrix} \overline{K}(\boldsymbol{x}_1, \boldsymbol{x}_i) \\ \vdots \\ \overline{K}(\boldsymbol{x}_n, \boldsymbol{x}_i) \end{bmatrix}$$

   Pour toutes les données d'un coup :

$$\boldsymbol{Z}^\top = \boldsymbol{U}\boldsymbol{\Phi}^\top = \boldsymbol{W}\boldsymbol{\Phi}\boldsymbol{\Phi}^\top = \boldsymbol{W}\overline{\boldsymbol{K}}$$

31.1 Version duale de la PCA

La version duale de la PCA calcule les nouvelles composantes comme

$$\boldsymbol{z}_i = \boldsymbol{W}\boldsymbol{\Phi}\overline{\phi}(\boldsymbol{x}_i), \qquad \textit{avec} \quad \boldsymbol{W} = \begin{bmatrix} \boldsymbol{w}_1^\top / \sqrt{\lambda_1} \\ \vdots \\ \boldsymbol{w}_p^\top / \sqrt{\lambda_p} \end{bmatrix} \in \mathbb{R}^{p \times n}$$

en calculant les vecteurs propres $\boldsymbol{w}_k$ de $\boldsymbol{\Phi}\boldsymbol{\Phi}^\top \in \mathbb{R}^{n \times n}$ au lieu des vecteurs propres $\boldsymbol{u}_k$ de $\boldsymbol{\Phi}^\top\boldsymbol{\Phi} \in \mathbb{R}^{d_\phi \times d_\phi}$.

Cela est possible grâce au raisonnement suivant, qui montre que $\boldsymbol{z}_i = \boldsymbol{W}\boldsymbol{\Phi}\overline{\phi}(\boldsymbol{x}_i)$ est équivalent à $\boldsymbol{z}_i = \boldsymbol{U}\overline{\phi}(\boldsymbol{x}_i)$.

Pour tout vecteur propre $\boldsymbol{u}_k$ de $\boldsymbol{\Phi}^\top\boldsymbol{\Phi}$:

$$\boldsymbol{\Phi}^\top\boldsymbol{\Phi}\boldsymbol{u}_k = \lambda_k\boldsymbol{u}_k$$

et, en posant

$$\boldsymbol{v}_k = \frac{1}{\lambda_k}\boldsymbol{\Phi}\boldsymbol{u}_k \in \mathbb{R}^n$$

cela donne

$$\boldsymbol{u}_k = \frac{1}{\lambda_k}\boldsymbol{\Phi}^\top\boldsymbol{\Phi}\boldsymbol{u}_k = \boldsymbol{\Phi}^\top\left(\frac{1}{\lambda_k}\boldsymbol{\Phi}\boldsymbol{u}_k\right) = \boldsymbol{\Phi}^\top\boldsymbol{v}_k$$

et

$$\boldsymbol{\Phi}\boldsymbol{\Phi}^\top\boldsymbol{v}_k = \frac{1}{\lambda_k}\boldsymbol{\Phi}\boldsymbol{\Phi}^\top\boldsymbol{\Phi}\boldsymbol{u}_k = \frac{1}{\lambda_k}\boldsymbol{\Phi}(\lambda_k\boldsymbol{u}_k) = \boldsymbol{\Phi}\boldsymbol{u}_k = \lambda_k\boldsymbol{v}_k$$

(les deux dernières égalités sont dues au fait que $\boldsymbol{u}_k$ est vecteur propre de $\boldsymbol{\Phi}^\top\boldsymbol{\Phi}$, et à la définition de $\boldsymbol{v}_k$).

Donc $\boldsymbol{v}_k$ *est bien un vecteur propre de* $\boldsymbol{\Phi}\boldsymbol{\Phi}^\top$ *associé à la même valeur propre* λ_k *que* u_k.

Cependant, si l'on calcule les vecteurs propres de $\boldsymbol{\Phi}\boldsymbol{\Phi}^\top$ avec un ordinateur, nous n'obtiendrons pas exactement les $\boldsymbol{v}_k$, car ceux-ci ne sont définis qu'à un facteur multiplicatif près. Typiquement, un solveur retourne des vecteurs propres $\boldsymbol{w}_k$ tels que $\|\boldsymbol{w}_k\| = 1$, alors que

$$\|\boldsymbol{v}_k\|^2 = \boldsymbol{v}_k^\top\boldsymbol{v}_k = \frac{1}{\lambda_k^2}\boldsymbol{u}_k^\top\boldsymbol{\Phi}^\top\boldsymbol{\Phi}\boldsymbol{u}_k = \frac{1}{\lambda_k^2}\boldsymbol{u}_k^\top(\lambda_k\boldsymbol{u}_k) = \frac{1}{\lambda_k}\boldsymbol{u}_k^\top\boldsymbol{u}_k = \frac{1}{\lambda_k}$$

Mais il suffit de renormaliser les $\boldsymbol{w}_k$ pour obtenir les $\boldsymbol{v}_k = \boldsymbol{w}_k/\sqrt{\lambda_k}$:

$$\|\boldsymbol{w}_k\| = 1 \qquad \Rightarrow \qquad \|\boldsymbol{w}_k/\sqrt{\lambda_k}\|^2 = 1/\lambda_k$$

Au final, on obtient donc les $\boldsymbol{u}_k$ à partir des $\boldsymbol{w}_k$:

$$\boldsymbol{u}_k = \boldsymbol{\Phi}^\top\boldsymbol{v}_k = \frac{1}{\sqrt{\lambda_k}}\boldsymbol{\Phi}^\top\boldsymbol{w}_k \qquad \text{et} \quad \boldsymbol{U} = \boldsymbol{W}\boldsymbol{\Phi}, \quad \boldsymbol{z}_i = \boldsymbol{U}\overline{\phi}(\boldsymbol{x}_i) = \boldsymbol{W}\boldsymbol{\Phi}\overline{\phi}(x_i)$$

31.1.1 Astuce du noyau

La forme duale de la PCA permet d'éviter le calcul des vecteurs propres en grande dimension. Cependant, il est toujours nécessaire de calculer explicitement la projection des données $\boldsymbol{\Phi}$ et les produits scalaires entre les images $\phi(\boldsymbol{x}_i)$ de tous les points inclus dans le produit $\boldsymbol{\Phi}\boldsymbol{\Phi}^\top$.

Pour palier à cette difficulté, les méthodes à noyaux calculent ces produits scalaires implicitement comme

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

à partir d'une fonction de noyau K valide, par exemple le noyau gaussien : $K(\boldsymbol{x}, \boldsymbol{x}') = \exp(-\|\boldsymbol{x} - \boldsymbol{x}'\|^2/2\sigma^2)$.

Il serait donc possible de calculer les vecteurs propres $\boldsymbol{w}_k$ de

$$\boldsymbol{\Phi}\boldsymbol{\Phi}^\top = \begin{bmatrix} \langle\phi(\boldsymbol{x}_1), \phi(\boldsymbol{x}_1)\rangle & \dots & \langle\phi(\boldsymbol{x}_1), \phi(\boldsymbol{x}_n)\rangle \\ \vdots & \ddots & \\ \langle\phi(\boldsymbol{x}_n), \phi(\boldsymbol{x}_1)\rangle & \dots & \langle\phi(\boldsymbol{x}_n), \phi(\boldsymbol{x}_n)\rangle \end{bmatrix}$$

à partir de

$$\boldsymbol{K} = \begin{bmatrix} K(\boldsymbol{x}_1, \boldsymbol{x}_1) & \dots & K(\boldsymbol{x}_1, \boldsymbol{x}_n) \\ \vdots & \ddots & \\ K(\boldsymbol{x}_n, \boldsymbol{x}_1) & \dots & K(\boldsymbol{x}_n, \boldsymbol{x}_n) \end{bmatrix}$$

Mais il faut en réalité calculer les vecteurs propres d'une version centrée de la matrice :

$$\begin{aligned}\boldsymbol{\Phi}\boldsymbol{\Phi}^\top &= \begin{bmatrix} \langle \phi(\boldsymbol{x}_1) - \phi_0, \phi(\boldsymbol{x}_1) - \phi_0 \rangle & \dots & \langle \phi(\boldsymbol{x}_1) - \phi_0, \phi(\boldsymbol{x}_n) - \phi_0 \rangle \\ \vdots & \ddots & \\ \langle \phi(\boldsymbol{x}_n) - \phi_0, \phi(\boldsymbol{x}_1) - \phi_0 \rangle & \dots & \langle \phi(\boldsymbol{x}_n) - \phi_0, \phi(\boldsymbol{x}_n) - \phi_0 \rangle \end{bmatrix} \\ &= \begin{bmatrix} \overline{K}(\boldsymbol{x}_1, \boldsymbol{x}_1) & \dots & \overline{K}(\boldsymbol{x}_1, \boldsymbol{x}_n) \\ \vdots & \ddots & \\ \overline{K}(\boldsymbol{x}_n, \boldsymbol{x}_1) & \dots & \overline{K}(\boldsymbol{x}_n, \boldsymbol{x}_n) \end{bmatrix} = \overline{\boldsymbol{K}}\end{aligned}$$

où $\boldsymbol{\phi}_0 = \frac{1}{n} \sum_{i=1}^n \phi(\boldsymbol{x}_i)$.

Pour cela, calculons :

$$\begin{aligned}\overline{K}(\boldsymbol{x}, \boldsymbol{x}') &= \langle \phi(\boldsymbol{x}) - \boldsymbol{\phi}_0, \phi(\boldsymbol{x}') - \boldsymbol{\phi}_0 \rangle = \langle \phi(\boldsymbol{x}), \phi(\boldsymbol{x}') \rangle - \langle \phi(\boldsymbol{x}), \boldsymbol{\phi}_0 \rangle - \langle \boldsymbol{\phi}_0, \phi(\boldsymbol{x}') \rangle + \langle \boldsymbol{\phi}_0, \boldsymbol{\phi}_0 \rangle \\ &= K(\boldsymbol{x}, \boldsymbol{x}') - \frac{1}{n} \sum_{i=1}^n K(\boldsymbol{x}, \boldsymbol{x}_i) - \frac{1}{n} \sum_{i=1}^n K(\boldsymbol{x}_i, \boldsymbol{x}') + \frac{1}{n^2} \sum_{i=1}^n \sum_{j=1}^n K(\boldsymbol{x}_i, \boldsymbol{x}_j)\end{aligned}$$

soit, de manière matricielle :

$$\overline{\boldsymbol{K}} = \boldsymbol{K} - \frac{1}{n} \boldsymbol{K} \mathbf{1}\mathbf{1}^\top - \frac{1}{n} \mathbf{1}\mathbf{1}^\top \boldsymbol{K} + \frac{\mathbf{1}^\top \boldsymbol{K} \mathbf{1}}{n^2} \mathbf{1}\mathbf{1}^\top$$

Ainsi, tous les calculs nécessaires peuvent se faire efficacement à partir de la matrice de noyau $\boldsymbol{K}$ standard.

Références

Arthur, D., et S. Vassilvitskii. 2007. « k-means++: the advantages of careful seeding ». In *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms, Orleans, Louisiana*, 1027-35.

Boyd, S., et L. Vandenberghe. 2004. *Convex Optimization*. Cambridge University Press. https://web.stanford.edu/~boyd/cvxbook/.

Kearns, Schapire, M. J. 1994. « Toward efficient agnostic learning ». *Machine Learning* 17 (2-3): 115-41.

Valiant, Leslie G. 1984. « A theory of the learnable ». *Communications of the ACM* 27 (11): 1134-42.

A Python

Tous les exemples de code de ce cours sont donnés en Python. Quelques bibliothèques utiles sont importées implicitement pour faire fonctionner ces codes.

Pour les calculs sur les vecteurs et matrices, nous utilisons NumPy :

```
import numpy as np
```

Pour les graphiques, nous utilisons Matplotlib :

```
import matplotlib.pyplot as plt
```

A.1 Bases de Python

Les programmes python sont enregistrés dans des fichiers `.py`, exécutables avec `python prog.py` ou `python3 prog.py`.

À savoir :

- Les variables n'ont pas besoin d'être déclarées mais elles ont néanmoins un type déterminé automatiquement (par exemple, `int`, `float`, ou `str`) visible avec `type(variable)`.
- Les commentaires sont repérés avec un `#` en début de ligne ou `""" ... """` pour plusieurs lignes.
- Les blocs d'instructions sont délimités par les tabulations (ou espaces en début de ligne).

Exemple :

```
"""
    Premier programme python simple
    qui commence par dire bonjour
"""
print("Bonjour !")

# Saisir une valeur
a = input("valuer de a ? ")
# a est une chaîne (str), il faut la convertir en entier :
```

```
a = int(a)
# et la reconvertir en chaîne avant de l'afficher :
print("2a = " + str(2*a))
```

A.1.1 Conditionnelles

Exemple :

```
if a == 2:
    ... code si a=2 avec une tabulation
else:
    ... code avec une tabulation

... suite du programme sans tabulation
```

Avec plusieurs conditions :

```
if a == 2:
    ... code si a=2
elif a == 3:
    ... code si a=3
else:
    ... code par défaut

... suite du programme
```

Le type booléen exste en python avec les valeurs `True`et `False`. Les booléens peuvent être combinés avec les opérateur `and`, `or`, `not`. L'opératuer de différence est `!=`.

A.1.2 Boucles

Les boucles FOR utilisent souvent la fonction `range` ainsi :

```
for i in range(10):
    ... code répété 10 fois
... suite du programme sans tabulation
```

Attention : la dernière valeur de `range(b)` ou `range(a,b)` est `b-1` :

```
for i in range(1,11):
    ... code répété pour i de 1 à 10
```

La boucle WHILE équivalente s'écrit :

```
i=1
while i<=10:
    ... code répété pour i de 1 à 10
    i=i+1
... suite du programme sans tabulation
```

La boule FOR permet d'itérer directement à travers les éléments d'une liste :

```
liste = [1, 2, "quatre", 8.0]
for i in liste:
    print(i)
```

```
1
2
quatre
8.0
```

A.1.3 Fonctions

On définit une fonction avec `def` :

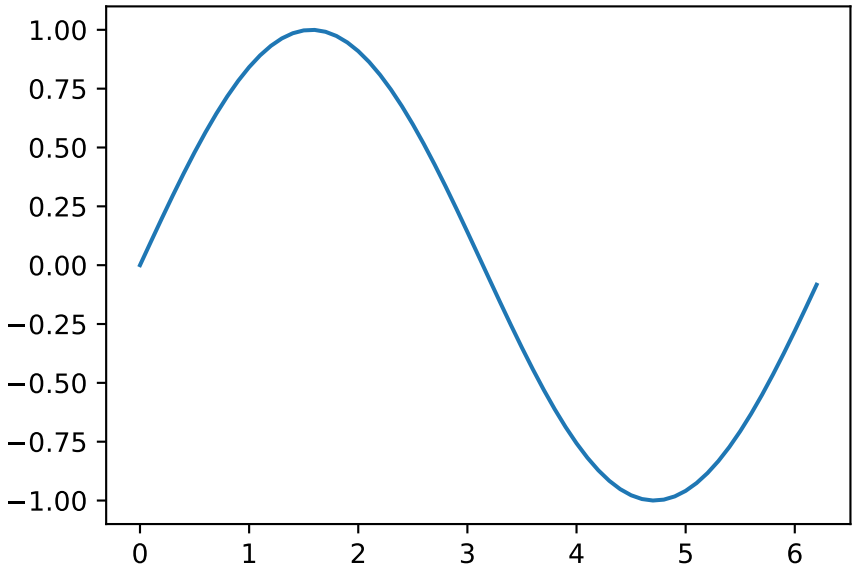
```
def mafonction(x1, x2):
    resultat = x1+2*x2
    return resultat

print( mafonction(4, 6) )
```

```
16
```

A.2 Graphiques

Pour tracer une courbe :

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

x = np.arange(0, 2*np.pi, 0.1)
plt.plot(x, np.sin(x) )
plt.show()
```



Fonctions utiles :

- `plt.figure()` : crée une nouvelle figure
- `plt.plot(...)` : tracer une courbe
- `plt.legend(['premiere courbe', 'deuxieme courbe'])` : légende
- `plt.axis([ax, bx, ay, by])` : zoom/échelle du graphique sur les bornes [ax,bx] et [ay, by]
- `plt.show()` : afficher toutes les figures et mettre le programme en attente jusqu'à la fermeture de des fenêtres

Pour la fonction `plot` :

- `plt.plot(X,Y)` : trace la courbe passant par les points de coordonnées `(X[i], Y[i])`
- `plt.plot(X,Y,'.')` : idem mais trace uniquement les points
- `plt.plot(X,Y,'r.')` : idem mais avec des points rouges (couleurs possibles : `b, r, g, m, y, c, k, w`)
- `plt.plot(Y)` : trace la courbe passant par les points de coordonnées `(i, Y[i])`

B Probabilités

Pour parler de probabilités, il faut une expérience aléatoire à partir de laquelle on définit un espace probabilisé.

B.1 Espace probabilisé

C'est un triplet $(\Omega, \mathcal{A}, P)$ muni de

- un **univers** Ω : l'ensemble de tous les résutlats d'expérience possibles
- une **tribu** (σ-algèbre) $\mathcal{A}$ de Ω : l'ensemble de tous les sous-ensembles « raisonnables » de Ω
  tel que i) $\Omega \in \mathcal{A}$, ii) $A \in \mathcal{A} \Rightarrow \overline{A} = (\Omega \setminus A) \in \mathcal{A}$, iii) $A, B \in \mathcal{A} \Rightarrow A \cup B \in \mathcal{A}$, iv) si $A_i \in \mathcal{A}$ pour tout $i \in \mathbb{N}$ alors $\cup_{i \in \mathbb{N}} A_i \in \mathcal{A}$
- une **mesure de probabilité** P : fonction associant une probabilité entre 0 et 1 à un événement

  $$P : \mathcal{A} \to [0, 1]$$

  telle que

  - $P(\Omega) = 1$ et $P(\emptyset) = 0$
  - Pour un nombre fini d'événements A_i disjoints : $P(\bigcup_i A_i) = \sum_i P(A_i)$

Propriétés de base :

- $P(\overline{A}) = 1 - P(A)$ avec $\overline{A} = \Omega \setminus A$
- $P(A \cup B) = P(A) + P(B) - P(A \cap B)$
- Si A et B disjoints : $P(A \cup B) = P(A) + P(B)$

On a aussi la **borne de l'union**

$$P(A \cup B) \leq P(A) + P(B) \tag{B.1}$$

car $P(A \cap B) \geq 0$.

B.2 Variables aléatoires (v.a.)

Une variable aléatoire réelle X est une *fonction* associant une valeur numérique au résultat d'une expérience

$$X : \Omega \to \mathbb{R}$$

Pour $A \subseteq \mathbb{R}$, l'ensemble des résultats d'expérience conduisant à une valeur de la v.a. dans A est un événement :

$$\{\omega \in \Omega \mid X(\omega) \in A\} \in \mathcal{A}$$

Notations :

- On notera simplement X pour $X(\omega)$ et $\{X \in A\}$ pour $\{\omega \in \Omega \mid X(\omega) \in A\}$.
- Les v.a. sont toujours notées en majuscules et les minuscules représentent des valeurs prises par les variables aléatoires.

La **loi d'une variable aléatoire** est une mesure de probabilité P_X sur $\mathbb{R}$ telle que

$$P_X(A) = P(X \in A)$$

Par exemple, si Ω est l'ensemble des étudiants dans une salle et que l'expérience consiste à tirer un étudian au hasard, alors on peut définir la variable aléatoire X qui mesure la taille de l'étudiant tiré en cm. Ainsi, la loi de X pour l'intervalle $[170, 180]$,

$$P_X([170, 180]) = P(X \in [170, 180]),$$

correspond à la probabilité de tirer un étudiant qui mesure entre 170cm et 180cm.

Une **v.a. discrète** Y ne peut prendre qu'un nombre fini de valeurs distinctes :

$$Y \in \mathcal{Y} = \{y_k\}_{1 \leq k \leq n}$$

La loi P_Y d'une v.a. discrète est donnée par la somme des probabilités de chacune des valeurs possibles :

$$P_Y(A) = \sum_{y \in \mathcal{Y} \cap A} P(Y = y)$$

Une **v.a. continue** X prend ses valeurs dans $\mathbb{R}$ et possède une **densité de probabilité** $p : \mathbb{R} \to \mathbb{R}^+$ telle que

$$P_X(A) = P(X \in A) = \int_A p(x)dx$$

Remarque :

$$\text{pour } a \in \mathbb{R}, \quad P_X(a) = P(X = a) = \int_a^a p(x)dx = 0$$

B.3 Espérance

L'espérance d'une v.a. correspond à la valeur qu'elle prend en moyenne sur une infinité de tirages. Elle peut être calculée :

- pour une **v.a. discrète** par

$$\mathbb{E}[Y] = \sum_{y \in \mathcal{Y}} y \, P(Y = y)$$

- pour une **v.a. continue** par

$$\mathbb{E}[X] = \int_{\mathbb{R}} x \, p(x) \, dx$$

On peut aussi calculer l'espérance d'une fonction de variables aléatoires :

$$\mathbb{E}[f(Y)] = \sum_{y \in \mathcal{Y}} f(y) \, P(Y = y)$$

$$\mathbb{E}[f(X)] = \int_{\mathbb{R}} f(x) \, p(x) \, dx$$

B.3.1 Propriétés

L'espérance est **linéaire** :

$$\mathbb{E}[aX] = a\mathbb{E}[X] \quad et \quad \mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y]$$

Pour deux v.a. indépendantes :

$$\mathbb{E}XY = (\mathbb{E}X)(\mathbb{E}Y)$$

Preuve

Pour des v.a. discrètes :

$$\begin{aligned}
\mathbb{E}XY &= \sum_{(x,y) \in \mathcal{X} \times \mathcal{Y}} xyP(X = x, Y = y) \\
&= \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} xy \; P(X = x)P(Y = y) \\
&= \sum_{x \in \mathcal{X}} xP(X = x) \sum_{y \in \mathcal{Y}} yP(Y = y) \\
&= \left(\sum_{x \in \mathcal{X}} xP(X = x) \right) \left(\sum_{y \in \mathcal{Y}} yP(Y = y) \right) \\
&= (\mathbb{E}X)(\mathbb{E}Y)
\end{aligned}$$

B.3.2 Inégalité de Jensen

Si la fonction $\varphi : \mathbb{R} \to \mathbb{R}$ est convexe, alors

$$\mathbb{E}\varphi(X) \geq \varphi(\mathbb{E}X)$$

Inversement, si la fonction $\varphi : \mathbb{R} \to \mathbb{R}$ est concave, alors

$$\mathbb{E}\varphi(X) \leq \varphi(\mathbb{E}X)$$

B.4 Variance et covariance

La variance d'une variable aléatoire X mesure la dispertion de cette variable, au sens de l'écart moyen entre X et son espérance au carré :

$$Var(X) = \mathbb{E}(X - \mathbb{E}X)^2$$

La **covariance** de deux v.a. X_1 et X_2 est

$$Cov(X_1, X_2) = \mathbb{E}\left[(X_1 - \mathbb{E}X_1)(X_2 - \mathbb{E}X_2)\right]$$

La **matrice de variance-covariance** d'en ensemble de n variables aléatoires X_i est la matrice $n \times n$ symétrique dont chaque élément sur la ligne i et la colonne j est $Cov(X_i, X_j)$. La diagonale contient les variances des variables car $Cov(X_i, X_i) = Var(X_i)$.

B.4.1 Propriétés

$$Var(X) = \mathbb{E}(X^2) - (\mathbb{E}X)^2$$

Preuve

Notons que $\mathbb{E}X$ est un nombre, et donc une constante du point de vue de l'espérance. Ainsi :

$$\begin{aligned}\mathbb{E}(X - \mathbb{E}X)^2 &= \mathbb{E}(X^2) - 2\mathbb{E}[X\mathbb{E}X] + \mathbb{E}[(\mathbb{E}X)^2] \\ &= \mathbb{E}(X^2) - 2\mathbb{E}X\mathbb{E}X + (\mathbb{E}X)^2 \\ &= \mathbb{E}(X^2) - (\mathbb{E}X)^2\end{aligned}$$

Pour deux v.a. indépendantes :

$$Cov(X, Y) = 0$$

Preuve

$$\begin{aligned} Cov(X_1, X_2) &= \mathbb{E}[(X_1 - \mathbb{E}X_1)(X_2 - \mathbb{E}X_2)] \\ &= \mathbb{E}[X_1X_2 - X_1\mathbb{E}X_2 - X_2\mathbb{E}X_1 + \mathbb{E}X_1\mathbb{E}X_2] \\ &= \mathbb{E}[X_1X_2] \ - \mathbb{E}X_1\mathbb{E}X_2 - \mathbb{E}X_2\mathbb{E}X_1 + \mathbb{E}X_1\mathbb{E}X_2 \\ &= \mathbb{E}[X_1X_2] - \mathbb{E}X_1\mathbb{E}X_2 \\ &= 0 \end{aligned}$$

car, pour X_1 et X_2 indépendantes, $\mathbb{E}X_1X_2 = \mathbb{E}X_1\mathbb{E}X_2$.

Pour deux v.a. indépendantes, on a aussi

$$Var(X_1 + X_2) = Var(X_1) + Var(X_2)$$

De manière générale :

$$Var(X_1 + X_2) = Var(X_1) + Var(X_2) + 2Cov(X_1, X_2)$$

Preuve

$$\begin{aligned} Var(X_1 + X_2) &= \mathbb{E}(X_1 + X_2 - \mathbb{E}X_1 - \mathbb{E}X_2)^2 = \mathbb{E}[(X_1 - \mathbb{E}X_1) + (X_2 - \mathbb{E}X_2)]^2 \\ &= \mathbb{E}[(X_1 - \mathbb{E}X_1)^2 + (X_2 - \mathbb{E}X_2)^2 + 2(X_1 - \mathbb{E}X_1)(X_2 - \mathbb{E}X_2)] \\ &= Var(X_1) + Var(X_2) + 2\mathbb{E}[(X_1 - \mathbb{E}X_1)(X_2 - \mathbb{E}X_2)] \end{aligned}$$

La **variance d'une v.a. bornée** est aussi bornée : pour $X \in [a, b]$:

$$Var(X) \leq \frac{(b-a)^2}{4}$$

Preuve

Premièrement,

$$\forall x, \quad Var(X) \leq \mathbb{E}(X - x)^2$$

car $f(x) = \mathbb{E}(X - x)^2 = (\mathbb{E}X)^2 - 2x\mathbb{E}X + x^2$ est une fonction quadratique de x qui atteint son minimum au point où $\frac{\mathrm{d}f(x)}{\mathrm{d}x} = 0$, c'est-à-dire lorsque $-2\mathbb{E}X + 2x) = 0$ et donc $x = \mathbb{E}X$ et $f(x) = Var(X)$.
Il suffit maintenant de choisir $x = (b-a)/2$ pour obtenir $Var(X) \leq f(x) = \mathbb{E}(X - (b-a)/2))^2$, où, avec $X \in [a, b]$, $|X - (b-a)/2| \leq (b-a)/2$ et donc $f(x) \leq (b-a)^2/4$.

B.5 Indicatrice

La fonction indicatrice agit comme un test et retourne 0 ou 1 : $\mathbf{1}(X \in A) = 1$ si l'événement $X \in A$ est observé, 0 sinon.

L'*espérance de l'indicatrice d'un événement est la probabilité de l'événement* :

$$\mathbb{E}_Y[\mathbf{1}(Y \in A)] = \sum_{y \in \mathcal{Y}} \mathbf{1}(Y \in A) P(Y = y) = \sum_{y \in \mathcal{Y} \cap A} P(Y = y) = P_Y(A)$$

et en continu :

$$\mathbb{E}_X[\mathbf{1}(X \in A)] = \int_{\mathbb{R}} \mathbf{1}(X \in A)\, p(x)\, dx = \int_A 1\, p(x)\, dx + \int_{\overline{A}} 0\, p(x)\, dx = P_X(A)$$

B.6 Couples de variables aléatoires

Un **couple de v.a. discrètes** $(X, Y) \in \mathcal{X} \times \mathcal{Y}$, $|\mathcal{X}| < \infty, |\mathcal{Y}| < \infty$, a une loi de probabilité $P_{X,Y}$ qu'on appelle **loi jointe** du couple et qui donne pour chaque ensemble de couples de valeurs $A \subseteq \mathcal{X} \times \mathcal{Y}$, la probabilité d'observer des couples (X, Y) dans cet ensemble :

$$P_{X,Y}(A) = P((X, Y) \in A) = \sum_{(x,y) \in A} P(X = x, Y = y)$$

(la virgule dans l'argument de la probabilité représente ici un ET entre deux conditions)

L'espérance par rapport à un couple se calcule de manière similaire au cas à une seule variable, simplement en moyennant sur des couples de valeurs :

$$\mathbb{E}_{(X,Y)}[f(X, Y)] = \sum_{(x,y) \in \mathcal{X} \times \mathcal{Y}} f(x, y)\, P(X = x, Y = y)$$

Un **couple de v.a. continues** $(X, Y) \in \mathcal{X} \times \mathcal{Y} \subseteq \mathbb{R}^2$ possède une loi jointe de densité $p_{X,Y}(x, y)$ qui est une fonction de deux variables, avec toujours la même relation à la loi de probabilité : pour $A \subseteq \mathcal{X} \times \mathcal{Y}$,

$$P_{X,Y}(A) = P((X, Y) \in A) = \int_A p_{X,Y}(x, y)\, dxdy$$

L'espérance se calcule comme une somme double selon les deux axes correspondant aux deux variables :

$$\mathbb{E}_{(X,Y)}[f(X, Y)] = \iint_{\mathbb{R}^2} f(x, y)\, p_{X,Y}(x, y)\; dxdy$$

B.7 Probabilités conditionnelles

- Probabilité de A sachant B : $P(A|B) = \frac{P(A,B)}{P(B)}$
  (si $P(B) \neq 0$)
- Pour des v.a. discrètes : $P(X = x|Y = y) = \frac{P(X=x,Y=y)}{P(Y=y)}$
- Pour des v.a. continues : $p_{X|Y}(x|y) = \frac{p_{X,Y}(x,y)}{p_Y(y)}$

> **Note**
>
> $P(X = x|Y = y)$ est une loi de probabilité sur X dont la définition dépend de la valeur de y. Ainsi, pour y fixé, $P(X = x|Y = y)$ respecte les mêmes propriétés qu'une probabilité classique. Par exemple, $\sum_{x\in\mathcal{X}} P(X = x|Y = y) = 1$.

B.8 Factorisation d'une loi jointe

Il est possible d'exprimer la loi jointe d'un couple à partir d'une probabilité conditionnelle. Cette « factorisation » s'obtient en faisant passer le dénominateur de l'autre côté dans la définition dans la probabilité conditionnelle :

- En discret :

$$P(X = x, Y = y) = P(X = x|Y = y)P(Y = y) = P(Y = y|X = x)P(X = x)$$

- En continu : $p_{X,Y}(x, y) = p_{X|Y}(x|y)\, p_Y(y) = p_{Y|X}(y|x)\, p_X(x)$

B.9 Lois marginales

Les lois marginales correspondent aux lois des variables considérées seules (par ex. $P(X = x)$ est une loi marginale pour le couple (X, Y)). Celles-ci peuvent être calculées à partir de la loi jointe grâce à la **loi des probabiltés totales** :

$$P(X = x) = \sum_{y\in\mathcal{Y}} P(X = x, Y = y)$$

B.10 Règle de Bayes

En écrivant les définitions des deux probabilités conditionnelles $P(X = x|Y = y)$ et $P(Y = y|X = x)$, on retrouve la règle de Bayes qui permet d'exprimer l'une en fonction de l'autre :

$$P(Y = y|X = x) = \frac{P(X = x, Y = y)}{P(X = x)}$$

$$\Rightarrow \text{factorisation } P(X = x, Y = y) = P(Y = y|X = x)P(X = x)$$

$$P(X = x|Y = y) = \frac{P(X = x, Y = y)}{P(Y = y)}$$

et donc

$$P(X = x|Y = y) = \frac{P(Y = y|X = x)P(X = x)}{P(Y = y)}$$

B.11 Indépendance

Deux v.a. X et Y sont **indépendantes** si et seulement si

$$P(X = x, Y = y) = P(X = x)P(Y = y)$$

Ainsi, pour deux variables indépendantes, connaître la valeur de l'une d'elles n'influence pas la probabilité de la seconde :

$$P(X = x|Y = y) = \frac{P(X = x, Y = y)}{P(Y = y)} = \frac{P(X = x)P(Y = y)}{P(Y = y)} = P(X = x)$$

Les variables X et Y sont **conditionnellement indépendantes** sachant Z si et seulement si

$$P(X = x, Y = y|Z = z) = P(X = x|Z = z)P(Y = y|Z = z)$$

> **Avertissement**
>
> L'indépendance conditionnelle n'implique pas forcément l'indépendance simple et vice versa. Nous verrons quelques exemples avec les réseaux bayésiens.

B.12 Espérance conditionnelle

L'espérance conditionnelle $\mathbb{E}[X|Y]$ est une variable aléatoire, fonction de la variable aléatoire Y. Elle correspond à une approximation de X contrainte à des valeurs constantes sur les sous-ensembles de Ω où Y est constante.

La notation $\mathbb{E}[X|Y = y]$ fait référence à la valeur que prend cette fonction en $Y = y$.

Le **théorème de l'espérance totale** garantie que

$$\mathbb{E}\mathbb{E}[X|Y] = \mathbb{E}X$$

ou encore, que

$$\mathbb{E}f(X, Y) = \mathbb{E}\mathbb{E}[f(X, Y)|Y]$$

C Algèbre linéaire

C.1 Vecteurs et matrices

Un vecteur de $\mathbb{R}^n$ peut être vu comme un tableau de dimension n : $\boldsymbol{a} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}$ est un vecteur de dimension 3, contenant 3 composantes notées a_i.

La matrice $\boldsymbol{A} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \\ A_{31} & A_{32} \end{bmatrix}$ est de dimension 3×2 et contient des composantes A_{ij}, où i est l'indice de ligne et j l'indice de colonne.

Par défaut, tous les vecteurs peuvent être considérés comme des matrices colonnes (n lignes et une seule colonne).

Vecteurs et matrices en python avec NumPy

En python, nous utilisons la bibliothèque numpy pour gérer les vecteurs et matrices.

```
import numpy as np

a = np.array([ 1, 2, 3])
A = np.array([[ 1, 2, 3], [4, 5, 6]])

print("Vecteur a de dimension " + str(len(a)) + " :\n", a)
print("Matrice A de taille " + str(A.shape) + " :\n", A)

print("a_1 =", a[0])
print("A_23 =", A[1, 2])
```

```
Vecteur a de dimension 3 :
 [1 2 3]
Matrice A de taille (2, 3) :
 [[1 2 3]
 [4 5 6]]
a_1 = 1
A_23 = 6
```

Remarque : numpy affiche toujours les vecteurs en ligne, mais nous pouvons considérer

qu'ils sont en colonne dans nos opérations (numpy se charge de le transposer dans le bon sens).

C.1.1 Transposée

La transposée $\boldsymbol{A}^T = \begin{bmatrix} A_{11} & A_{21} & A_{31} \\ A_{12} & A_{22} & A_{32} \end{bmatrix}$ de la matrice $\boldsymbol{A}$ s'obtient en inversant les lignes et les colonnes.

La transposée $\boldsymbol{a}^T = \begin{bmatrix} a_1 & a_2 & a_3 \end{bmatrix}$ du vecteur $\boldsymbol{a}$ est donc une matrice ligne.

Si $\boldsymbol{A}^T = \boldsymbol{A}$, alors $\boldsymbol{A}$ est dite **symétrique**.

En python

En python/numpy, la transposée s'écrit « .T » :

```
print("Matrice A:\n", A)
print("Transposée de A :\n", A.T)
```

```
Matrice A:
 [[1 2 3]
 [4 5 6]]
Transposée de A :
 [[1 4]
 [2 5]
 [3 6]]
```

Remarque : `A.T` est une simple « vue » sur la matrice, ce qui signifie qu'elle ne nécessite pas de calculs pour l'obtenir, mais aussi que modifier `A.T` modifie aussi `A`.

C.1.2 Addition

Les additions de vecteurs ou de matrices de mêmes dimensions se font composante à composante (case par case).

En python

En python/numpy, les opérateurs d'addition/soustraction sont étendus aux vecteurs et matrices :

```
A = np.array([[ 1, 2, 3], [4, 5, 6]])
B = np.array([[ 1, -2, -3], [4, 5, 6]])
print("A + B :\n", A+B)
```

```
A + B :
 [[ 2  0  0]
 [ 8 10 12]]
```

C.1.3 Produit scalaire

Le produit scalaire entre deux vecteurs $\boldsymbol{a}$ et $\boldsymbol{b}$ de même dimension n est donné par la somme des produits des composantes :

$$\langle \boldsymbol{a}, \boldsymbol{b} \rangle = \sum_{i=1}^{n} a_i b_i$$

En notation matricielle, le produit scalaire correspond à la multiplication matricielle de la transposée de $\boldsymbol{a}$ avec $\boldsymbol{b}$:

$$\langle \boldsymbol{a}, \boldsymbol{b} \rangle = \boldsymbol{a}^T \boldsymbol{b}$$

A noter : **le produit scalaire est symétrique**, $\boldsymbol{a}^T \boldsymbol{b} = \boldsymbol{b}^T \boldsymbol{a}$.

C.1.4 Multiplication matricielle

Soit $\boldsymbol{A}$ de dimension $m \times n$ et $\boldsymbol{B}$ de dimension $n \times p$. Le produit $\boldsymbol{C} = \boldsymbol{AB}$ a m lignes et p colonnes, et chaque composante de $\boldsymbol{C}$ correspond au produit scalaire entre une ligne de la matrice $\boldsymbol{A}$ et une colonne de la matrice $\boldsymbol{B}$:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj} = \boldsymbol{A}_{i,:} \boldsymbol{B}_{:,j}$$

où $\boldsymbol{A}_{i,:}$ est la $i$ème ligne de $\boldsymbol{A}$ et $\boldsymbol{B}_{:,j}$ est la $j$ème colonne de $\boldsymbol{B}$.

Par exemple, avec

$$\boldsymbol{A} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \\ A_{31} & A_{32} \end{bmatrix}, \qquad \boldsymbol{B} = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

on obtient $\boldsymbol{C}$ de dimension 3×2 avec

$$\boldsymbol{C} = \boldsymbol{AB} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \\ A_{31} & A_{32} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \\ A_{31}B_{11} + A_{32}B_{21} & A_{31}B_{12} + A_{32}B_{22} \end{bmatrix}$$

À noter : la multiplication matricielle n'est pas symétrique, en général $\boldsymbol{AB} \neq \boldsymbol{BA}$.

La transposée d'un produit matriciel inverse l'ordre du produit :

$$(\boldsymbol{AB})^T = \boldsymbol{B}^T \boldsymbol{A}^T$$

En python

En python/numpy, la multiplication mattricielle s'écrit « @ » :

```
print("Produit scalaire a^T a :\n", a.T @ a)
print("Produit matriciel A^T A :\n", A.T @ A)
```

```
Produit scalaire a^T a :
 14
Produit matriciel A^T A :
 [[17 22 27]
 [22 29 36]
 [27 36 45]]
```

Remarque : on trouve la notation `A.dot(B)` ou `np.dot(A, B)` qui sont équivalents à `A @ B`.

C.1.5 Normes

La norme euclidienne d'un vecteur est

$$\|\boldsymbol{a}\| = \sqrt{\sum_{i=1}^{n} a_i^2} = \sqrt{\boldsymbol{a}^T \boldsymbol{a}}$$

La norme ℓ_1 d'un vecteur est

$$\|\boldsymbol{a}\|_1 = \sum_{i=1}^{n} |a_i|$$

Il existe aussi beaucoup de normes matricielles différentes.

En python

En python/numpy, les normes sont implémentée par `np.linalg.norm` :

```
print("Norme euclidienne de a =", np.linalg.norm(a), " ou", np.sqrt( a.T@a ) )
print("Norme l1 de a =", np.linalg.norm(a, ord=1) , " ou", np.sum( np.abs(a) ) )
```

```
Norme euclidienne de a = 3.7416573867739413  ou 3.7416573867739413
Norme l1 de a = 6.0  ou 6
```

C.1.6 Vecteurs et matrices particuliers

La notation $\mathbf{0}$ est utilisée à la fois pour le vecteur et les matrices dont toutes les composantes sont nulles. De même, on note parfois $\mathbf{1}$ le vecteur dont toutes les composantes valent 1.

La **matrice identité** est la matrice carrée notée $\boldsymbol{I}$ neutre pour le produit matriciel :

$$\boldsymbol{I} = \begin{bmatrix} 1 & & 0 \\ & \ddots & \\ 0 & & 1 \end{bmatrix} = diag(\mathbf{1})$$

et, pour toute matrice $\boldsymbol{A}$ carrée et vecteur $\boldsymbol{a}$,

$$\boldsymbol{AI} = \boldsymbol{IA} = \boldsymbol{A} \qquad et \qquad \boldsymbol{Ia} = \boldsymbol{a}$$

En python

En python/numpy :

```
n=3
print("Vecteur nul : ", np.zeros(n) )
print("Vecteur de 1 : ", np.ones(n) )
print("Matrice de zéros :\n", np.zeros((n, 2*n)) )
print("Matrice identité :\n", np.eye(n) )
```

```
Vecteur nul :  [0. 0. 0.]
Vecteur de 1 :  [1. 1. 1.]
Matrice de zéros :
 [[0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]]
Matrice identité :
 [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

C.2 Inégalité de Cauchy-Schwarz

L'inégalité de Cauchy-Schwarz permet de borner l'amplitude d'un produit scalaire en fonction des normes des vecteurs : pour tout vecteurs $\boldsymbol{u}$ et $\boldsymbol{v}$,

$$\left|\boldsymbol{u}^T\boldsymbol{v}\right| \leq \|\boldsymbol{u}\|\|\boldsymbol{v}\|$$

et la borne est atteinte, $|\boldsymbol{u}^T\boldsymbol{v}| = \|\boldsymbol{u}\|\|\boldsymbol{v}\|$, si et seulement si les vecteurs sont colinéaires, c'est-à-dire si $\boldsymbol{u} = \lambda\boldsymbol{v}$.

C.3 Matrice inverse

Une matrice carrée $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ est dite **inversible** si sa matrice inverse $\boldsymbol{A}^{-1}$ existe telle que

$$\boldsymbol{A}\boldsymbol{A}^{-1} = \boldsymbol{A}^{-1}\boldsymbol{A} = \boldsymbol{I}$$

C.4 Rang d'une matrice

Le rang d'une matrice est le nombre de lignes ou colonnes linéairement indépendantes. De manière générale, pour une matrice à m lignes et n colonnes :

$$\text{rang}(\boldsymbol{A}) \leq \min\{m, n\}$$

et

$$\text{rang}(\boldsymbol{AB}) \leq min\{\text{rang}(\boldsymbol{A}), \text{rang}(\boldsymbol{B})\}$$

(car la multiplication par $\boldsymbol{B}$ revient à créer des combinaisons linéaires des colonnes de $\boldsymbol{A}$, et donc ne peut pas rendre indépendantes des colonnes qui ne l'étaient pas).

Le rang d'une matrice carrée correspond au nombre de valeurs propres non nulles. Pour une matrice rectangulaire, le rang correspond au nombre de valeurs singulières non nulles.

C.5 Décomposition en vecteurs et valeurs propres

Pour une matrice carrée $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ « diagonalisable » :

$$\boldsymbol{A} = \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^{-1}$$

avec

$$\boldsymbol{\Lambda} = \begin{bmatrix} \lambda_1 & & 0 \\ & \ddots & \\ 0 & & \lambda_n \end{bmatrix} \qquad \boldsymbol{U} = \begin{bmatrix} | & & | \\ \boldsymbol{u}_1 & \cdots & \boldsymbol{u}_n \\ | & & | \end{bmatrix}$$

où

- $\boldsymbol{\Lambda}$ est une matrice diagonale contenant les n **valeurs propres** λ_i,
- $\boldsymbol{U} \in \mathbb{R}^{n \times n}$ est une matrice carrée contenant les n **vecteurs propres** $\boldsymbol{u}_i$ associés tels que

  $$\boldsymbol{A}\boldsymbol{u}_i = \lambda_i \boldsymbol{u}_i$$

  avec $\boldsymbol{u}_i \neq \boldsymbol{0}$.

Pour une matrice $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ **symétrique** ($\boldsymbol{A}^T = \boldsymbol{A}$) : $\boldsymbol{U}$ est orthogonale ($\boldsymbol{U}^{-1} = \boldsymbol{U}^T$) et les λ_i sont réelles.

Exemple en python

```
M = A.T @ A
print("Matrice M :\n", M)

# eigh calcule les vecteurs propres d'une matrice symétrique
Lambda, U = np.linalg.eigh(M)

print("U :\n", U, "\nValeurs propres (par ordre croissant) :\n", Lambda)

print("ULU^T :\n", U @ np.diag(Lambda) @ U.T )

print("M u = lambda * u ? Vérification pour u3 :\n")
print(M @ U[:,2])
print(Lambda[2] * U[:,2])
```

```
Matrice M :
 [[17 22 27]
 [22 29 36]
 [27 36 45]]
U :
 [[ 0.40824829 -0.80596391 -0.42866713]
 [-0.81649658 -0.11238241 -0.56630692]
 [ 0.40824829  0.58119908 -0.7039467 ]]
Valeurs propres (par ordre croissant) :
 [2.91325866e-15 5.97327474e-01 9.04026725e+01]
ULU^T :
 [[17. 22. 27.]
 [22. 29. 36.]
 [27. 36. 45.]]
M u = lambda * u ? Vérification pour u3 :

[-38.7526545  -51.19565893 -63.63866337]
[-38.7526545  -51.19565893 -63.63866337]
```

C.6 Décomposition en valeurs singulières (SVD)

Toute matrice $\boldsymbol{X} \in \mathbb{R}^{n \times d}$ possède une décomposition en valeurs singulières :

$$\boldsymbol{X} = \boldsymbol{U}\boldsymbol{S}\boldsymbol{V}^T \qquad ou \quad \underbrace{\begin{bmatrix} — & \boldsymbol{x}_1^\top & — \\ & \vdots & \\ — & \boldsymbol{x}_n^\top & — \end{bmatrix}}_{n \times d} = \underbrace{\begin{bmatrix} | & & | \\ \boldsymbol{u}_1 & \cdots & \boldsymbol{u}_n \\ | & & | \end{bmatrix}}_{n \times n} \underbrace{\begin{bmatrix} \sigma_1 & & 0 & 0 \\ & \ddots & & \vdots \\ 0 & & \sigma_n & 0 \end{bmatrix}}_{n \times d} \underbrace{\begin{bmatrix} — & \boldsymbol{v}_1^\top & — \\ & \vdots & \\ — & \boldsymbol{v}_d^\top & — \end{bmatrix}}_{d \times d}$$

avec les matrices

$$\boldsymbol{U}^T\boldsymbol{U} = \boldsymbol{U}\boldsymbol{U}^T = \boldsymbol{I}, \quad et \quad \boldsymbol{V}^T\boldsymbol{V} = \boldsymbol{V}\boldsymbol{V}^T = \boldsymbol{I}$$

qui contiennent les **vecteurs singuliers** à gauche et à droite et $\boldsymbol{S}$ qui contient les **valeurs singluières**.

Exemple en python

```
M = A.T @ A
print("Matrice M :\n", M)

# eigh calcule les vecteurs propres d'une matrice symétrique
Lambda, U = np.linalg.eigh(M)

print("U :\n", U, "\nValeurs propres (par ordre croissant) :\n", Lambda)

print("ULU^T :\n", U @ np.diag(Lambda) @ U.T )

print("M u = lambda * u ? Vérification pour u3 :\n")
print(M @ U[:,2])
print(Lambda[2] * U[:,2])
```

```
Matrice M :
 [[17 22 27]
 [22 29 36]
 [27 36 45]]
U :
 [[ 0.40824829 -0.80596391 -0.42866713]
 [-0.81649658 -0.11238241 -0.56630692]
 [ 0.40824829  0.58119908 -0.7039467 ]]
Valeurs propres (par ordre croissant) :
 [2.91325866e-15 5.97327474e-01 9.04026725e+01]
ULU^T :
 [[17. 22. 27.]
 [22. 29. 36.]
 [27. 36. 45.]]
M u = lambda * u ? Vérification pour u3 :

[-38.7526545  -51.19565893 -63.63866337]
[-38.7526545  -51.19565893 -63.63866337]
```

C.6.1 Lien avec les vecteurs et valeurs propres

Dans la décomposition en valeurs singulières, les vecteurs $\boldsymbol{v}_k$ et les valeurs singulières σ_k sont liés aux vecteurs/valeurs propres de $\boldsymbol{X}^T\boldsymbol{X}$:

$$\boldsymbol{X}^T\boldsymbol{X} = \boldsymbol{V}\boldsymbol{S}^T\boldsymbol{U}^T\boldsymbol{U}\boldsymbol{S}\boldsymbol{V}^T = \boldsymbol{V}\boldsymbol{S}^T\boldsymbol{S}\boldsymbol{V}^T = \boldsymbol{V}\boldsymbol{S}^2\boldsymbol{V}^T$$

où $\boldsymbol{S}^2 = diag(\sigma_1^2, \dots, \sigma_n^2, 0, \dots)$. Donc $\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{v}_k = \sigma_k^2\boldsymbol{v}_k$, car, par exemple pour $k = 1$:

$$\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{v}_1 = \boldsymbol{V}\boldsymbol{S}^2\boldsymbol{V}^T\boldsymbol{v}_1 = \boldsymbol{V}\boldsymbol{S}^2\begin{bmatrix}1\\0\\|\end{bmatrix} = \boldsymbol{V}\begin{bmatrix}\sigma_1^2\\0\\|\end{bmatrix} = \sigma_1^2\boldsymbol{v}_1$$

Exemple en python

```
M = A.T @ A

# eigh calcule les vecteurs propres d'une matrice symétrique
Lambda, U = np.linalg.eigh(M)

print("U :\n", U[:,::-1], "\nValeurs propres (par ordre décroissant) :\n", Lambda[::-1]

# svd
U, s, Vt = np.linalg.svd(A)
print("Vecteurs propres par SVD :\n", Vt.T, "\nValeurs propres par SVD (par ordre décroi
```

```
U :
 [[-0.42866713 -0.80596391  0.40824829]
 [-0.56630692 -0.11238241 -0.81649658]
 [-0.7039467   0.58119908  0.40824829]]
Valeurs propres (par ordre décroissant) :
 [9.04026725e+01 5.97327474e-01 2.91325866e-15]
Vecteurs propres par SVD :
 [[-0.42866713  0.80596391  0.40824829]
 [-0.56630692  0.11238241 -0.81649658]
 [-0.7039467  -0.58119908  0.40824829]]
Valeurs propres par SVD (par ordre décroissant en ignorant les valeurs nulles) :
 [90.40267253  0.59732747]
```

A noter : les vecteurs propres ne sont retrouvés qu'au signe près, car si $\boldsymbol{M}\boldsymbol{u} = \lambda\boldsymbol{u}$ alors $\boldsymbol{M}(-\boldsymbol{u}) = \lambda(-\boldsymbol{u})$ aussi.

C.6.2 SVD fine ou tronquée

Si $d > n$, alors la **SVD fine** construit une matrice $\boldsymbol{S}_n$ ne contenant que les n premières colonnes de $\boldsymbol{S}$ et une matrice $\boldsymbol{V}_n^T$ ne contenant que les n premières lignes de $\boldsymbol{V}^T$, sans

modifier le résultat :

$$\boldsymbol{X} = \boldsymbol{U}\boldsymbol{S}_n\boldsymbol{V}_n^T$$

La **SVD tronquée** reprend ce principe, mais limite les calculs à $p < n$ composantes. En revanche, dans ce cas, le résultat peut être modifié et $\boldsymbol{X} \neq \hat{\boldsymbol{X}} = \boldsymbol{U}\boldsymbol{S}_p\boldsymbol{V}_p^T$ si $p < rang(\boldsymbol{X})$. Cependant, $\hat{\boldsymbol{X}}$ constitue la meilleure approximation de rang p de $\boldsymbol{X}$.

D Calcul différentiel

Le calcul différentiel s'intéresse aux variations de la sortie d'une fonction relativement aux variations de ses entrées.

Remarque: toutes les définitions ci-dessous sont données sous réserve d'existence des limites et de dérivabilité des fonctions.

D.1 Dérivée d'une fonction d'une variable

Le dérivée d'une fonction $f(x)$ pour $x \in \mathbb{R}$ est donnée par

$$f'(x) = \frac{\mathrm{d}f(x)}{\mathrm{d}x} = \lim_{\substack{\delta \to 0 \\ \delta \neq 0}} \frac{f(x+\delta) - f(x)}{\delta}$$

et représente la variation observée en sortie de la fonction lors d'une infime variation de son entrée. C'est aussi la pente de la tangente à la courbe de f au point x.

D.2 Dérivées partielles et gradient d'une fonction de plusieurs variables

Pour une fonction de plusieurs variables, $f(\boldsymbol{x}) = f(x_1, \ldots, x_n)$, les dérivées partielles

$$\frac{\partial f(\boldsymbol{x})}{\partial x_k} = \lim_{\substack{\delta \to 0 \\ \delta \neq 0}} \frac{f(x_1, \ldots, x_k + \delta, \ldots, x_n) - f(\boldsymbol{x})}{\delta}$$

représentent les variatons de la fonction lors d'infimes variations de ses entrées x_k considérées indépendamment les unes des autres.

Le **gradient** de cette fonction est le vecteur concatenant toutes les dérivées partielles :

$$\frac{\mathrm{d}f(\boldsymbol{x})}{\mathrm{d}\boldsymbol{x}} = \begin{bmatrix} \frac{\partial f(\boldsymbol{x})}{\partial x_1} \\ \vdots \\ \frac{\partial f(\boldsymbol{x})}{\partial x_n} \end{bmatrix}$$

Il correspond à la direction de la plus grande pente de la fonction.

Direction de la plus grande pente

La direction de la plus grande pente correspond plus précisément à la direction dans laquelle l'approximation linéaire de f au point $\boldsymbol{x}$ (l'hyperplan tangent à la courbe) possède la plus grande pente, mesurée par le rapport entre la variation de la sortie et la variation de l'entrée, elle-même considérée au travers de la norme euclidienne de la variation $\|\boldsymbol{\delta}\|$.

L'approximation linéaire de f en $\boldsymbol{x}_0$ est $\hat{f}(\boldsymbol{x}) = \frac{\mathrm{d}f(\boldsymbol{x})}{\mathrm{d}\boldsymbol{x}}\Big|_{\boldsymbol{x}=\boldsymbol{x}_0}^{\top} \boldsymbol{x}$. Ainsi, $\hat{f}(\boldsymbol{x}_0+\boldsymbol{\delta}) - \hat{f}(\boldsymbol{x}_0) = \frac{\mathrm{d}f(\boldsymbol{x})}{\mathrm{d}\boldsymbol{x}}\Big|_{\boldsymbol{x}=\boldsymbol{x}_0}^{\top} \boldsymbol{\delta}$ et la direction

$$\boldsymbol{u} = \arg\max_{\boldsymbol{\delta}\in\mathbb{R}^d} \frac{|\hat{f}(\boldsymbol{x}_0+\boldsymbol{\delta}) - \hat{f}(\boldsymbol{x}_0)|}{\|\boldsymbol{\delta}\|} = \arg\max_{\boldsymbol{\delta}\in\mathbb{R}^d} \frac{\left|\frac{\mathrm{d}f(\boldsymbol{x})}{\mathrm{d}\boldsymbol{x}}\Big|_{\boldsymbol{x}=\boldsymbol{x}_0}^{\top} \boldsymbol{\delta}\right|}{\|\boldsymbol{\delta}\|}$$

est donnée par $\boldsymbol{u} = \frac{\mathrm{d}f(\boldsymbol{x})}{\mathrm{d}\boldsymbol{x}}\Big|_{\boldsymbol{x}=\boldsymbol{x}_0}$ (cas d'égalité de l'inégalité de Cauchy-Schwarz).

D.3 Matrice jacobienne

Lorsque l'on dérive une fonction à valeur vectorielle

$$\boldsymbol{f}(\boldsymbol{x}) = \begin{bmatrix} f_1(\boldsymbol{x}) \\ \vdots \\ f_m(\boldsymbol{x}) \end{bmatrix} \in \mathbb{R}^m$$

par rapport à un vecteur $\boldsymbol{x} \in \mathbb{R}^n$ de n variables, il existe $m \times n$ dérivées partielles $\frac{\partial f_i(\boldsymbol{x})}{\partial x_j}$, $i = 1, \ldots, m,\ j = 1, \ldots, n$ que l'on peut organiser dans un tableau à double entrée, ou une matrice, que l'on appelle la matrice jacobienne de $\boldsymbol{f}$:

$$\frac{\mathrm{d}\boldsymbol{f}(\boldsymbol{x})}{\mathrm{d}\boldsymbol{x}} = \begin{bmatrix} \left(\frac{\mathrm{d}f_1(\boldsymbol{x})}{\mathrm{d}\boldsymbol{x}}\right)^{\top} \\ \vdots \\ \left(\frac{\mathrm{d}f_m(\boldsymbol{x})}{\mathrm{d}\boldsymbol{x}}\right)^{\top} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1(\boldsymbol{x})}{\partial x_1} & \cdots & \frac{\partial f_1(\boldsymbol{x})}{\partial x_m} \\ \vdots & & \vdots \\ \frac{\partial f_m(\boldsymbol{x})}{\partial x_1} & \cdots & \frac{\partial f_m(\boldsymbol{x})}{\partial x_m} \end{bmatrix}$$

On remarque que chaque ligne de cette matrice n'est autre que la transposée du gradient d'une composante de $\boldsymbol{f}$.

D.4 Quelques règles utiles

D.4.1 Dérivation en chaîne

Si f est définie comme la composition de deux fonctions $g : \mathbb{R}^d \to \mathbb{R}$ et $h : \mathbb{R} \to \mathbb{R}$,

$$f(\boldsymbol{x}) = h(g(\boldsymbol{x})) = h(u), \quad u = g(\boldsymbol{x}),$$

alors sa dérivée et le produit des dérivées des deux fonctions g et h :

$$\frac{\mathrm{d}f(\boldsymbol{x})}{\mathrm{d}\boldsymbol{x}} = \frac{\mathrm{d}h(u)}{\mathrm{d}u}\frac{\mathrm{d}g(\boldsymbol{x})}{\mathrm{d}\boldsymbol{x}}$$

D.4.2 Gradient d'un produit scalaire

Soit la fonction linéaire

$$f(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} = \sum_{k=1}^{d} w_k x_k$$

définie par un produit scalaire pour $\boldsymbol{x} \in \mathbb{R}^d$. Alors, son gradient correspond au vecteur des coefficients :

$$\frac{\partial f(\boldsymbol{x})}{\partial x_k} = w_k, \quad k = 1, \ldots, d, \quad \Rightarrow \quad \frac{\mathrm{d}f(\boldsymbol{x})}{\mathrm{d}\boldsymbol{x}} = \boldsymbol{w}.$$

D.4.3 Gradient d'une fonction quadratique

Soit la fonction quadratique

$$f(\boldsymbol{x}) = \|\boldsymbol{x}\|^2 = \boldsymbol{x}^T\boldsymbol{x} = \sum_{k=1}^{d} x_k^2$$

définie pour $\boldsymbol{x} \in \mathbb{R}^d$. Alors, son gradient se calcule ainsi :

$$\frac{\partial f(\boldsymbol{x})}{\partial x_k} = 2x_k, \quad k = 1, \ldots, d, \quad \Rightarrow \quad \frac{\mathrm{d}f(\boldsymbol{x})}{\mathrm{d}\boldsymbol{x}} = 2\boldsymbol{x}.$$

D.4.4 Matrice jacobienne d'un produit matrice-vecteur

La dérivée de la fonction linéaire à valeur vectorielle

$$\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{A}\boldsymbol{x} = \begin{bmatrix} \boldsymbol{a}_1^T \\ \vdots \\ \boldsymbol{a}_n^T \end{bmatrix} \boldsymbol{x} = \begin{bmatrix} \boldsymbol{a}_1^T\boldsymbol{x} \\ \vdots \\ \boldsymbol{a}_n^T\boldsymbol{x} \end{bmatrix} = \begin{bmatrix} f_1(\boldsymbol{x}) \\ \vdots \\ f_n(\boldsymbol{x}) \end{bmatrix},$$

ou plutôt sa matrice jacobienne, est $\boldsymbol{A}$ car

$$\frac{\mathrm{d}f_i(\boldsymbol{x})}{\mathrm{d}\boldsymbol{x}} = \boldsymbol{a}_i$$

d'après la règle de dérivation des produits scalaires.

E Optimisation

L'optimisation consiste à recherche le minimum (ou le maximum) d'une fonction d'une ou plusieurs variables, éventuellement sujettes à des contraintes.

E.1 Optimisation sans contrainte

E.1.1 Descente de gradient

E.1.1.1 Pour minimiser une fonction d'une seule variable

À partir d'un point $x \in \mathbb{R}$ quelconque, la dérivée $f'(x)$ donne toute l'information nécessaire pour minimiser la fonction $f(x)$:

- $f'(x) > 0$ indique que la fonction est croissante $\Rightarrow$ il faut reculer pour descendre ;
- $f'(x) < 0$ indique que la fonction décroissante $\Rightarrow$ il faut avancer pour descendre ;
- $f'(x) = 0$ indique un point critique (minimum ou maximum).

Ainsi, l'algorithme suivant permet de converger vers la solution $x^* = \arg\min_{x \in \mathbb{R}} f(x)$:

$$x \leftarrow x - \mu f'(x) \qquad \text{tant que } f'(x) \neq 0$$

où $\mu \in (0, 1]$ est un paramètre réglant la taille du pas effectué dans la direction opposée à la dérivée.

Cependant, la convergence n'est garantie que si μ est choisi suffisamment petit, comme illustré dans l'exemple ci-dessous.

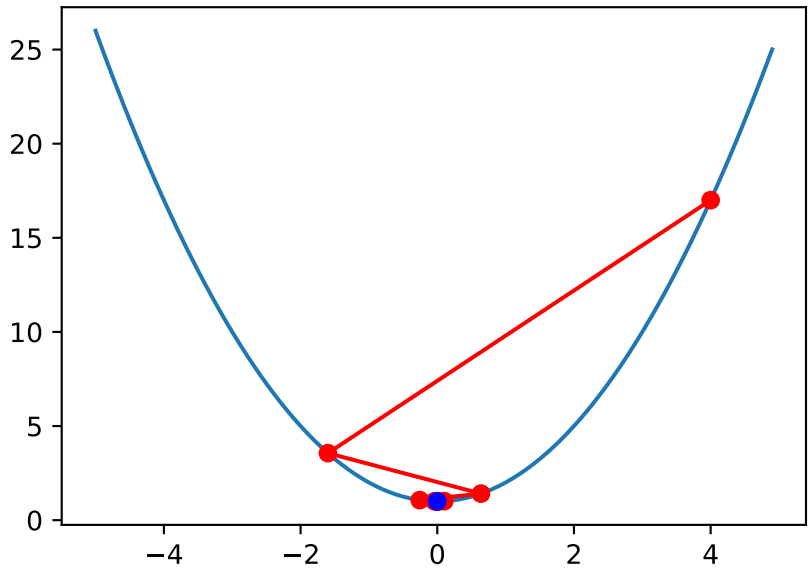
Exemple qui ne convergence pas avec mu=1

Soit $f(x) = 1 + x^2$, de dérivée $\frac{\mathrm{d}f(x)}{\mathrm{d}dx} = 2x$.
Si l'algorithme est initialisé à $x = 4$, alors $\frac{\mathrm{d}f(x)}{\mathrm{d}x} = 8$, ce qui amène pour $\mu = 1$ à $x \leftarrow 4 - 8 = -4$ et à $\frac{\mathrm{d}f(x)}{\mathrm{d}x} = -8$. Les itérations suivantes alternent donc entre $x = 4$ et $x = -4$ indéfiniment.

Exemple de convergence rapide

```python
def f(x):
    return 1 + x**2

def df_dx(x):
    return 2 * x

# initialisation
x = 4
print(f"x = {x:.5f}, f(x) = {f(x):.5f}, df_df(x) = {df_dx(x):.5f}")

# taille du pas
mu = 0.7

while abs(df_dx(x)) > 1e-3:
    x -= mu * df_dx(x)
    print(f"x = {x:.5f}, f(x) = {f(x):.5f}, df_df(x) = {df_dx(x):.5f}")
```

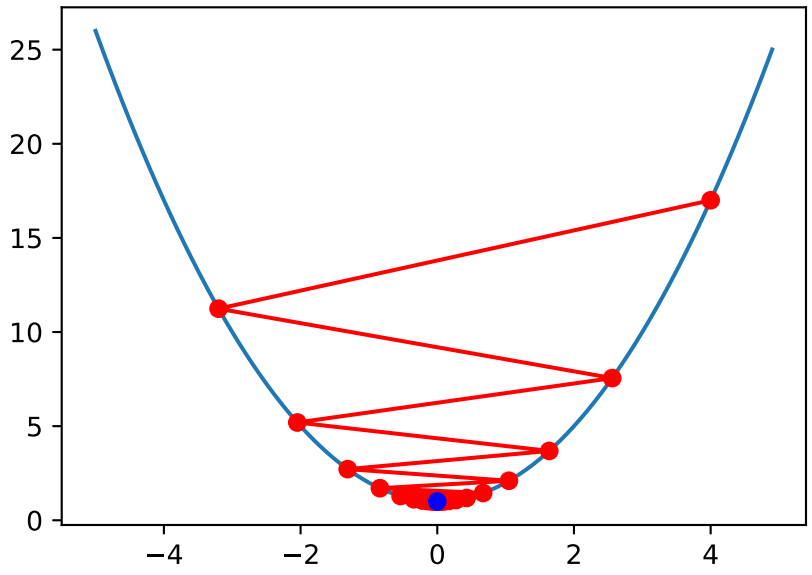
```
x = 4.00000, f(x) = 17.00000, df_df(x) = 8.00000
x = -1.60000, f(x) = 3.56000, df_df(x) = -3.20000
x = 0.64000, f(x) = 1.40960, df_df(x) = 1.28000
x = -0.25600, f(x) = 1.06554, df_df(x) = -0.51200
x = 0.10240, f(x) = 1.01049, df_df(x) = 0.20480
x = -0.04096, f(x) = 1.00168, df_df(x) = -0.08192
x = 0.01638, f(x) = 1.00027, df_df(x) = 0.03277
x = -0.00655, f(x) = 1.00004, df_df(x) = -0.01311
x = 0.00262, f(x) = 1.00001, df_df(x) = 0.00524
x = -0.00105, f(x) = 1.00000, df_df(x) = -0.00210
x = 0.00042, f(x) = 1.00000, df_df(x) = 0.00084
```



Exemple de convergence plus lente avec mu=0.9

```
x = 4.00000, f(x) = 17.00000, df_df(x) = 8.00000
x = -3.20000, f(x) = 11.24000, df_df(x) = -6.40000
x = 2.56000, f(x) = 7.55360, df_df(x) = 5.12000
x = -2.04800, f(x) = 5.19430, df_df(x) = -4.09600
x = 1.63840, f(x) = 3.68435, df_df(x) = 3.27680
x = -1.31072, f(x) = 2.71799, df_df(x) = -2.62144
x = 1.04858, f(x) = 2.09951, df_df(x) = 2.09715
x = -0.83886, f(x) = 1.70369, df_df(x) = -1.67772
x = 0.67109, f(x) = 1.45036, df_df(x) = 1.34218
x = -0.53687, f(x) = 1.28823, df_df(x) = -1.07374
x = 0.42950, f(x) = 1.18447, df_df(x) = 0.85899
x = -0.34360, f(x) = 1.11806, df_df(x) = -0.68719
x = 0.27488, f(x) = 1.07556, df_df(x) = 0.54976
x = -0.21990, f(x) = 1.04836, df_df(x) = -0.43980
x = 0.17592, f(x) = 1.03095, df_df(x) = 0.35184
x = -0.14074, f(x) = 1.01981, df_df(x) = -0.28147
x = 0.11259, f(x) = 1.01268, df_df(x) = 0.22518
x = -0.09007, f(x) = 1.00811, df_df(x) = -0.18014
x = 0.07206, f(x) = 1.00519, df_df(x) = 0.14412
x = -0.05765, f(x) = 1.00332, df_df(x) = -0.11529
x = 0.04612, f(x) = 1.00213, df_df(x) = 0.09223
x = -0.03689, f(x) = 1.00136, df_df(x) = -0.07379
x = 0.02951, f(x) = 1.00087, df_df(x) = 0.05903
x = -0.02361, f(x) = 1.00056, df_df(x) = -0.04722
```

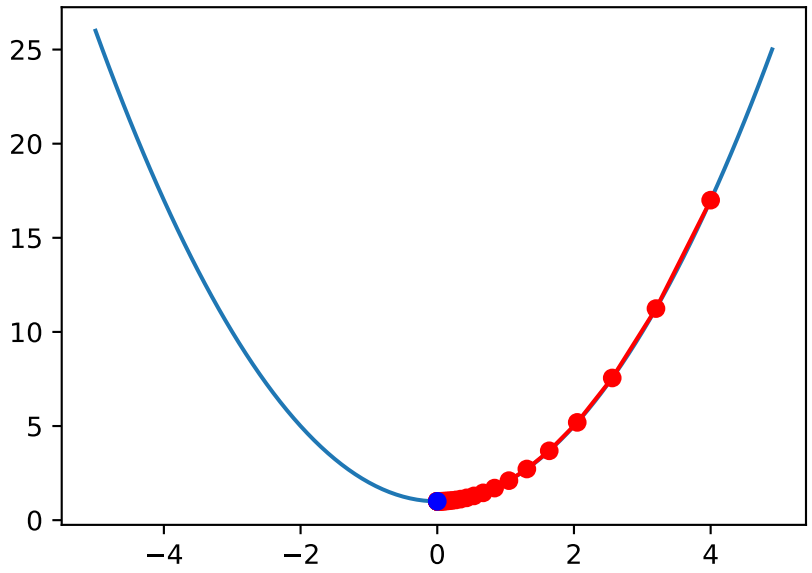
```
x = 0.01889, f(x) = 1.00036, df_df(x) = 0.03778
x = -0.01511, f(x) = 1.00023, df_df(x) = -0.03022
x = 0.01209, f(x) = 1.00015, df_df(x) = 0.02418
x = -0.00967, f(x) = 1.00009, df_df(x) = -0.01934
x = 0.00774, f(x) = 1.00006, df_df(x) = 0.01547
x = -0.00619, f(x) = 1.00004, df_df(x) = -0.01238
x = 0.00495, f(x) = 1.00002, df_df(x) = 0.00990
x = -0.00396, f(x) = 1.00002, df_df(x) = -0.00792
x = 0.00317, f(x) = 1.00001, df_df(x) = 0.00634
x = -0.00254, f(x) = 1.00001, df_df(x) = -0.00507
x = 0.00203, f(x) = 1.00000, df_df(x) = 0.00406
x = -0.00162, f(x) = 1.00000, df_df(x) = -0.00325
x = 0.00130, f(x) = 1.00000, df_df(x) = 0.00260
x = -0.00104, f(x) = 1.00000, df_df(x) = -0.00208
x = 0.00083, f(x) = 1.00000, df_df(x) = 0.00166
x = -0.00066, f(x) = 1.00000, df_df(x) = -0.00133
x = 0.00053, f(x) = 1.00000, df_df(x) = 0.00106
x = -0.00043, f(x) = 1.00000, df_df(x) = -0.00085
```



Atre exemple de convergence lente avec mu trop petit

```
x = 4.00000, f(x) = 17.00000, df_df(x) = 8.00000
x = 3.20000, f(x) = 11.24000, df_df(x) = 6.40000
x = 2.56000, f(x) = 7.55360, df_df(x) = 5.12000
x = 2.04800, f(x) = 5.19430, df_df(x) = 4.09600
x = 1.63840, f(x) = 3.68435, df_df(x) = 3.27680
x = 1.31072, f(x) = 2.71799, df_df(x) = 2.62144
```

```
x = 1.04858, f(x) = 2.09951, df_df(x) = 2.09715
x = 0.83886, f(x) = 1.70369, df_df(x) = 1.67772
x = 0.67109, f(x) = 1.45036, df_df(x) = 1.34218
x = 0.53687, f(x) = 1.28823, df_df(x) = 1.07374
x = 0.42950, f(x) = 1.18447, df_df(x) = 0.85899
x = 0.34360, f(x) = 1.11806, df_df(x) = 0.68719
x = 0.27488, f(x) = 1.07556, df_df(x) = 0.54976
x = 0.21990, f(x) = 1.04836, df_df(x) = 0.43980
x = 0.17592, f(x) = 1.03095, df_df(x) = 0.35184
x = 0.14074, f(x) = 1.01981, df_df(x) = 0.28147
x = 0.11259, f(x) = 1.01268, df_df(x) = 0.22518
x = 0.09007, f(x) = 1.00811, df_df(x) = 0.18014
x = 0.07206, f(x) = 1.00519, df_df(x) = 0.14412
x = 0.05765, f(x) = 1.00332, df_df(x) = 0.11529
x = 0.04612, f(x) = 1.00213, df_df(x) = 0.09223
x = 0.03689, f(x) = 1.00136, df_df(x) = 0.07379
x = 0.02951, f(x) = 1.00087, df_df(x) = 0.05903
x = 0.02361, f(x) = 1.00056, df_df(x) = 0.04722
x = 0.01889, f(x) = 1.00036, df_df(x) = 0.03778
x = 0.01511, f(x) = 1.00023, df_df(x) = 0.03022
x = 0.01209, f(x) = 1.00015, df_df(x) = 0.02418
x = 0.00967, f(x) = 1.00009, df_df(x) = 0.01934
x = 0.00774, f(x) = 1.00006, df_df(x) = 0.01547
x = 0.00619, f(x) = 1.00004, df_df(x) = 0.01238
x = 0.00495, f(x) = 1.00002, df_df(x) = 0.00990
x = 0.00396, f(x) = 1.00002, df_df(x) = 0.00792
x = 0.00317, f(x) = 1.00001, df_df(x) = 0.00634
x = 0.00254, f(x) = 1.00001, df_df(x) = 0.00507
x = 0.00203, f(x) = 1.00000, df_df(x) = 0.00406
x = 0.00162, f(x) = 1.00000, df_df(x) = 0.00325
x = 0.00130, f(x) = 1.00000, df_df(x) = 0.00260
x = 0.00104, f(x) = 1.00000, df_df(x) = 0.00208
x = 0.00083, f(x) = 1.00000, df_df(x) = 0.00166
x = 0.00066, f(x) = 1.00000, df_df(x) = 0.00133
x = 0.00053, f(x) = 1.00000, df_df(x) = 0.00106
x = 0.00043, f(x) = 1.00000, df_df(x) = 0.00085
```



E.1.1.2 Pour minimiser une fonction de plusieurs variables

À partir d'un point $\boldsymbol{x} \in \mathbb{R}^n$ quelconque, le gradient donne toute l'information nécessaire pour minimiser la fonction, car le gradient $\frac{\mathrm{d}f(\boldsymbol{x})}{\mathrm{d}\boldsymbol{x}}$ correspond à la direction de la plus grande pente. Ainsi, la direction opposée au gradient indique la plus grande pente descendante et l'algorithme

$$\boldsymbol{x} \leftarrow \boldsymbol{x} - \mu \frac{\mathrm{d}f(\boldsymbol{x})}{\mathrm{d}\boldsymbol{x}} \qquad \Leftrightarrow \quad x_k \leftarrow x_k - \mu \frac{\partial f(\boldsymbol{x})}{\partial x_k}, \quad k = 1, \ldots, n$$

converge vers l'optimum à $\boldsymbol{x}^*$, où le gradient est nul :

$$\frac{\mathrm{d}f(\boldsymbol{x}^*)}{\mathrm{d}\boldsymbol{x}} = \mathbf{0}$$

E.1.2 Optimisation locale vs globale

La descente de gradient permet de se rapporcher d'un minimum de la fonction objectif. Cependant, seule la convergence vers un minimum local est garantie, c'est-à-dire qu'aucun point dans un voisinnage de la solution ne peut être meilleur :

$$\exists \epsilon > 0, \ \forall \boldsymbol{x}, \ \textit{tel que} \ \|\boldsymbol{x} - \boldsymbol{x}^*\| \leq \epsilon, \quad f(\boldsymbol{x}) \geq f(\boldsymbol{x}^*)$$

Un minimum global localisé en $\boldsymbol{x}^*$ est tel que

$$\forall \boldsymbol{x}, \quad f(\boldsymbol{x}) \geq f(\boldsymbol{x}^*)$$

et garantie donc qu'aucune autre solution n'est meilleure.

Cas particulier : *tous les minima locaux d'une fonction convexe sont aussi des minima globaux* (et inversement, tous les maxima locaux d'une fonction concave sont aussi globaux).

E.1.3 Fonctions convexes et concaves

Une fonction $f : \mathbb{R}^d \to \mathbb{R}$ est **convexe** si, pour tout $\boldsymbol{x} \in \mathbb{R}^d$, $\boldsymbol{x}' \in \mathbb{R}^d$, et $\lambda \in [0, 1]$,

$$f(\lambda \boldsymbol{x} + (1 - \lambda)\boldsymbol{x}') \leq \lambda f(\boldsymbol{x}) + (1 - \lambda) f(\boldsymbol{x}')$$

Cela signifie que tout segment entre deux points du graphique de f se situe *au-dessus* du graphique de f.

Inversement, une fonction $f : \mathbb{R}^d \to \mathbb{R}$ est **concave** si tout segment entre deux points du graphique de f se situe *en-dessous* du graphique de f, ou encore, si $-f$ est convexe.

Exemples de fonctions convexes et concaves :

- les fonctions linéaires ou affines qui sont à la fois convexes et concaves ;
- les normes, $\|\boldsymbol{x}\|_p$ pour $p \geq 1$, sont convexes ;
- la norme d'une fonction linéaire, $\|\boldsymbol{A}\boldsymbol{x} + \boldsymbol{b}\|$, est convexe ;
- la somme de deux fonctions convexes est convexe ;
- le maximum point à point de deux fonctions f_1 et f_2 convexes, $f(\boldsymbol{x}) = \max\{f_1(\boldsymbol{x}), f_2(\boldsymbol{x})\}$, est convexe ;
- la racine carrée est concave.

Les fonctions quadratiques sont soit convexe, soit concave.

Plus de détails sur l'optimisation de fonctions convexes peuvent être trouvés dans Boyd et Vandenberghe (2004).

E.1.3.1 Malédiction de la dimension

Le problème des minima locaux peut être résolu en lançant de multiples initialisations à partir d'intialisations différentes. Cependant, le nombre d'initialisations nécessaires croît rapidement avec la dimension d du problème.

Pour le voir, imaginons que le domaine de recherche de la solution soit limité au cube unité : $\boldsymbol{x} \in [0, 1]^d$. Dans ce cas, pour $d = 1$, il suffit de choisir $1/\epsilon$ initialisations équiréparties entre 0 et 1 pour être sûr de lancer l'optimisation à une distance inférieure à ϵ de la solution globale, et si le bassin d'attraction de cette solution (l'ensemble des initialisations qui convergent vers elle) est plus large que ϵ, alors elle sera trouvée. Donc si $\epsilon = 0.1$, il suffit de choisir $N = 10$ initialisations.

Si $d = 2$, alors l'ensemble des initialisations devient une grille en 2D, avec $N = 1/\epsilon^2 = 100$ points. Pour $d = 3$, la grille comprend $N = 1/\epsilon^3 = 1000$ points.

Pour $d > 4$, les choses sont en fait pires, dans le sens où $N = 1/\epsilon^d$ ne suffit plus. En effet, une telle grille garantie uniquement d'avoir une distance $1/\epsilon$ entre deux coins d'un petit hypercube délimité par la grille. Mais la solution du problème recherchée pourrait se trouver au centre de l'hypercube.

Pour $d = 1$, cela ne change rien. Pour $d = 2$, le centre d'un petit carré de côté ϵ est à une distance $\epsilon\sqrt{2}/2 < \epsilon$ des coins, car la diagonale du carré mesure $\sqrt{\epsilon^2 + \epsilon^2}$.

Pour $d = 3$, la diagonale du cube mesure $\sqrt{\epsilon^2 + diagonale(face\ carree)^2} = \sqrt{3\epsilon^2}$ et la distance devient $\epsilon\sqrt{3}/2$, ce qui est encore inférieur à ϵ. Mais pour $d > 4$, la distance au centre est $\epsilon\sqrt{d}/2 > \epsilon$. Il est donc nécessaire de choisir une grille plus fine avec une distance entre deux coins valant $2\epsilon/\sqrt{d}$. Ainsi, la distance au centre sera $2\epsilon/\sqrt{d} \times \sqrt{d}/2 = \epsilon$.

Pour $d > 4$, il faut donc

$$N = \left(\frac{\sqrt{d}}{2\epsilon}\right)^d$$

pour garantir une distance maximale de ϵ entre la solution recherchée et une intialisation. Pour $d = 10$, ce nombre est déjà proche de 10 milliards.

Notons que cette formule est en fait valable pour tout $d \geq 1$. Par exemple, pour $d = 1$, il aurait suffit de choisir 5 points espacés de 0.2 pour garantir d'avoir une initialisation à une distance maximale de 0.1 de la solution.

E.2 Optimisation sous contraintes

Considérons maintenant le problème d'optimisation soumis à des contraintes suivant

$$\begin{aligned} &\min_{\boldsymbol{x} \in \mathbb{R}^d} f(\boldsymbol{x}) \\ &\text{s.c. } g(\boldsymbol{x}) \geq 0 \end{aligned}$$

TODO...

F Inégalités de concentration

En probabilités, les inégalités de concentration caractérisent la concentration de la moyenne empirique autour de l'espérance, c'est-à-dire de la véritable moyenne.

F.1 Inégalité de Markov

L'inégalité de Markov relie la probabilité d'observer des valeurs de grande amplitude à la moyenne de l'amplitude : pour toute variable aléatoire Z,

$$\forall t > 0, \quad P(|Z| \geq t) \leq \frac{\mathbb{E}|Z|}{t}$$

Preuve

Pour une variable aléatoire continue $Z \in \mathbb{R}$, et tout $t > 0$,

$$\begin{aligned}
\mathbb{E}|Z| &= \int_{\mathbb{R}} |z|\, p_Z(z)\, dz \\
&= \int_{|z|<t} |z|\, p_Z(z)\, dz + \int_{|z|\geq t} |z|\, p_Z(z)\, dz \\
&\geq \int_{|z|\geq t} |z|\, p_Z(z)\, dz && \text{(car la premierère intégrale est positive)} \\
&\geq \int_{|z|\geq t} t\, p_Z(z)\, dz = t \int_{|z|\geq t} p_Z(z)\, dz && \text{(car } |z| \geq t\text{)} \\
&\geq tP(|Z| \geq t)
\end{aligned}$$

F.2 Inégalité de Bienaymé-Chebyshev

L'inégalité de Bienaymé-Chebyshev relie la distance entre une variable aléatoire et son espérance à sa variance :

$$\forall t > 0, \quad P(|Z - \mathbb{E}Z| \geq t) \leq \frac{Var(Z)}{t^2}$$

Preuve

Pour toute variable aléatoire Z :

$$P(|Z - \mathbb{E}Z| \geq t) = P((Z - \mathbb{E}Z)^2 \geq t^2)$$

L'inégalité de Markov appliquée à la variable $Z' = (Z - \mathbb{E}Z)^2$ avec la constante t^2 donne

$$P((Z - \mathbb{E}Z)^2 \geq t^2) \leq \frac{\mathbb{E}(Z - \mathbb{E}Z)^2}{t^2} = \frac{Var(Z)}{t^2}$$

F.3 Inégalité de Chebyshev pour les moyennes

Pour n variables Z_i, indépendantes et identiquement distribuées comme Z (des copies indépendantes de Z),

$$P\left\{\left|\frac{1}{n}\sum_{i=1}^{n} Z_i - \mathbb{E}Z\right| \geq t\right\} \leq \frac{Var(Z)}{nt^2}$$

Preuve

En appliquant l'inégalité de Bienaymé-Chebyshev à la variable $\frac{1}{n}\sum_{i=1}^{n} Z_i$, nous avons

$$P\left\{\left|\frac{1}{n}\sum_{i=1}^{n} Z_i - \mathbb{E}\left[\frac{1}{n}\sum_{i=1}^{n} Z_i\right]\right| \geq t\right\} \leq \frac{Var(\frac{1}{n}\sum_{i=1}^{n} Z_i)}{t^2},$$

où la linéarité de l'espérance conduit à

$$\mathbb{E}\left[\frac{1}{n}\sum_{i=1}^{n} Z_i\right] = \frac{1}{n}\sum_{i=1}^{n}\mathbb{E}Z_i = \frac{1}{n}\sum_{i=1}^{n}\mathbb{E}Z = \mathbb{E}Z$$

car les Z_i sont identiquement distrubées et partagent l'espérance de Z. Puisque les Z_i sont indépendantes, la variance se calcule comme

$$Var\left(\frac{1}{n}\sum_{i=1}^{n} Z_i\right) = \frac{1}{n^2}\sum_{i=1}^{n} Var(Z_i) = \frac{1}{n^2}\sum_{i=1}^{n} Var(Z) = \frac{Var(Z)}{n},$$

ce qui conclut la preuve.

F.4 Borne de Chernoff

Pour toute variable aléatoire Z,

$$\forall t > 0,\ s > 0, \quad P(Z \geq t) \leq \frac{\mathbb{E}e^{sZ}}{e^{st}}$$

Cette borne est au cœur de la méthode de Chernoff qui consiste ensuite à trouver une valeur de s qui minimise le membre de droite.

Preuve

Puisque l'exponentielle est strictement croissante et que s est positif,

$$P(Z \geq t) = P(e^{sZ} \geq e^{st})$$

Le reste est une conséquence directe de l'inégalité de Markov appliquée à la variable positive e^{sZ} avec la constante e^{st}.

F.5 Inégalité de Hoeffding

Pour n variables Z_i, indépendantes et identiquement distribuées comme la variable bornée $Z \in [a, b]$ (des copies indépendantes de Z), pour tout $\epsilon > 0$,

$$P\left\{\frac{1}{n}\sum_{i=1}^{n} Z_i - \mathbb{E}Z \geq \epsilon\right\} \leq \exp\left(\frac{-2n\epsilon^2}{(b-a)^2}\right),$$

$$P\left\{\frac{1}{n}\sum_{i=1}^{n} Z_i - \mathbb{E}Z \leq -\epsilon\right\} \leq \exp\left(\frac{-2n\epsilon^2}{(b-a)^2}\right),$$

$$P\left\{\left|\frac{1}{n}\sum_{i=1}^{n} Z_i - \mathbb{E}Z\right| \geq \epsilon\right\} \leq 2\exp\left(\frac{-2n\epsilon^2}{(b-a)^2}\right).$$

Preuve

Etape 1 :
Montrons tout d'abord que, pour une variable bornée $Z \in [a, b]$ et centrée, c'est-à-dire telle que $\mathbb{E}Z = 0$,

$$\mathbb{E}e^{sZ} \leq e^{s^2(b-a)^2/8}$$

Pour tout $z \in [a, b]$, écrit comme $z = \lambda a + (1-\lambda)b$, la convexité de l'exponentielle implique que $e^z \leq \lambda e^a + (1-\lambda)e^b$. Un raisonnement similaire appliqué à $sz \in [sa, sb]$ avec $\lambda = (b-z)/(b-a)$ conduit à

$$e^{sz} \leq \frac{b-z}{b-a}e^{sa} + \left(1 - \frac{b-z}{b-a}\right)e^{sb} = \frac{b-z}{b-a}e^{sa} + \frac{z-a}{b-a}e^{sb}$$

Ainsi,

$$\begin{aligned}\mathbb{E}e^{sZ} &\leq \mathbb{E}\left[\frac{b-Z}{b-a}e^{sa}+\frac{Z-a}{b-a}e^{sb}\right]\\ &\leq \frac{b}{b-a}e^{sa}-\frac{a}{b-a}e^{sb}+\mathbb{E}Z\left(\frac{-1}{b-a}e^{sa}+\frac{1}{b-a}e^{sb}\right) \quad \text{(par linéarité de l'espérance)}\\ &\leq \frac{b}{b-a}e^{sa}-\frac{a}{b-a}e^{sb} \quad \text{(car } \mathbb{E}Z=0\text{)}\end{aligned}$$

Définissons $c=\frac{-a}{b-a}$, $u=s(b-a)$ et $f(u)=-cu+\log(1-c+ce^u)$. Alors, $a=-c(b-a)$, $b=(1-c)(b-a)$ et

$$\begin{aligned}\mathbb{E}e^{sZ} &\leq (1-c)e^{sa}+ce^{sb}=(1-c)e^{-cs(b-a)}+ce^{(1-c)s(b-a)}\\ &\leq (1-c+ce^{s(b-a)})e^{-cs(b-a)}=(1-c+ce^u)e^{-cu}\\ &\leq e^{f(u)}\end{aligned}$$

Nous avons $f(0)=0$ et la dérivée

$$f'(u)=-c+\frac{ce^u}{1-c+ce^u}=-c+\frac{c}{(1-c)e^{-u}+c}$$

qui vaut aussi zéro en $u=0$. La dérivée seconde est bornée par

$$f''(u)=\frac{c(1-c)e^{-u}}{((1-c)e^{-u}+c)^2}=\frac{\alpha\beta}{(\alpha+\beta)^2}\leq\frac{1}{4}$$

puisque $(\alpha+\beta)^2-4\alpha\beta=(\alpha-\beta)^2\geq 0$. Ainsi, le développement de Taylor de f en 0 donne, pour un certain $\theta\in[0,u]$,

$$f(u)=f(0)+uf'(0)+\frac{u^2}{2}f''(\theta)\leq\frac{u^2}{8}=\frac{s^2(b-a)^2}{8}$$

Etape 2 :
Nous pouvons maintenant appliquer la borne de Chernoff à la variable $\sum_{i=1}^n(Z_i-\mathbb{E}Z)$:

$$P\left\{\sum_{i=1}^n(Z_i-\mathbb{E}Z)\geq t\right\}\leq e^{-st}\mathbb{E}\left[e^{s\sum_{i=1}^n(Z_i-\mathbb{E}Z)}\right]=e^{-st}\mathbb{E}\left[\prod_{i=1}^n e^{s(Z_i-\mathbb{E}Z)}\right]$$

Grâce à l'indépendance des Z_i, l'espérance peut être insérée dans le produit :

$$P\left\{\sum_{i=1}^n(Z_i-\mathbb{E}Z)\geq t\right\}\leq e^{-st}\prod_{i=1}^n\mathbb{E}\left[e^{s(Z_i-\mathbb{E}Z)}\right]$$

Utilisons maintenant le résultat de l'Etape 1 pour les variables $(Z_i-\mathbb{E}Z)$, bornées par

$$a_i-\mathbb{E}Z\leq Z_i-\mathbb{E}Z\leq b_i-\mathbb{E}Z$$

et centrées car

$$\mathbb{E}\left[Z_i - \mathbb{E}Z\right] = \mathbb{E}Z_i - \mathbb{E}[\mathbb{E}Z] = \mathbb{E}Z - \mathbb{E}Z = 0.$$

Cela donne

$$\mathbb{E}\left[e^{s(Z_i - \mathbb{E}Z)}\right] \leq e^{s^2(b_i-a_i)^2/8}$$

La borne de Chernoff devient alors :

$$\begin{aligned} P\left\{\sum_{i=1}^{n}(Z_i - \mathbb{E}Z) \geq t\right\} &\leq e^{-st}\prod_{i=1}^{n} e^{s^2(b_i-a_i)^2/8} \\ &= \exp\left(-st + \frac{s^2\sum_{i=1}^{n}(b_i-a_i)^2}{8}\right) \end{aligned}$$

Si l'on choisit $s = 4t/\sum_{i=1}^{n}(b_i - a_i)^2$, nous obtenons

$$\begin{aligned} P\left\{\sum_{i=1}^{n}(Z_i - \mathbb{E}Z) \geq t\right\} &\leq \exp\left(\frac{-4t^2}{\sum_{i=1}^{n}(b_i-a_i)^2} + \frac{16t^2}{8\sum_{i=1}^{n}(b_i-a_i)^2}\right) \\ &= \exp\left(\frac{-2t^2}{\sum_{i=1}^{n}(b_i-a_i)^2}\right) \end{aligned}$$

Etape 3 :
La version de l'inégalité de Hoeffding impliquant la moyenne plutôt que la somme (la première des trois présentées ci-dessus) s'obtient ensuite ainsi :

$$\begin{aligned} P\left\{\frac{1}{n}\sum_{i=1}^{n} Z_i - \mathbb{E}Z \geq \frac{t}{n}\right\} &= P\left\{\frac{1}{n}\sum_{i=1}^{n}(Z_i - \mathbb{E}Z) \geq \frac{t}{n}\right\} \\ &= P\left\{\sum_{i=1}^{n}(Z_i - \mathbb{E}Z) \geq t\right\} \\ &\leq \exp\left(\frac{-2t^2}{\sum_{i=1}^{n}(b_i-a_i)^2}\right) \end{aligned}$$

en choisissant $t = n\epsilon$ avec $a_i = a$ and $b_i = b$.
Etape 4 :
La seconde des trois inégalités est obtenue de manière similaire en considérant la variable $\sum_{i=1}^{n}\left(\mathbb{E}Z - Z_i\right)$ au lieu de $\sum_{i=1}^{n}\left(Z_i - \mathbb{E}Z\right)$. En utilisant les bornes

$$\mathbb{E}Z - b_i \leq \mathbb{E}Z - Z_i \leq \mathbb{E}Z - a_i$$

cela conduit au même type de résultat :

$$P\left\{\mathbb{E}Z - \frac{1}{n}\sum_{i=1}^{n} Z_i \geq \epsilon\right\} \leq \exp\left(\frac{-2n\epsilon^2}{(b-a)^2}\right)$$

à partir duquel nous pouvons déduire :

$$P\left\{\frac{1}{n}\sum_{i=1}^{n} Z_i - \mathbb{E}Z \leq -\epsilon\right\} = P\left\{\mathbb{E}Z - \frac{1}{n}\sum_{i=1}^{n} Z_i \geq \epsilon\right\} \leq \exp\left(\frac{-2n\epsilon^2}{(b-a)^2}\right).$$

Enfin, la troisième inégalité impliquant la valeur absolue est une conséquence directe des deux précédentes et de la borne de l'union (Équation B.1) :

$$\begin{aligned} P\left\{\left|\frac{1}{n}\sum_{i=1}^{n} Z_i - \mathbb{E}Z\right| > \epsilon\right\} &= P\left\{\left\{\frac{1}{n}\sum_{i=1}^{n} Z_i - \mathbb{E}Z > \epsilon\right\} \cup \left\{\frac{1}{n}\sum_{i=1}^{n} Z_i - \mathbb{E}Z < -\epsilon\right\}\right\} \\ &\leq P\left\{\frac{1}{n}\sum_{i=1}^{n} Z_i - \mathbb{E}Z > \epsilon\right\} + P\left\{\frac{1}{n}\sum_{i=1}^{n} Z_i - \mathbb{E}Z < -\epsilon\right\} \\ &\leq 2\exp\left(\frac{-2n\epsilon^2}{(b-a)^2}\right). \end{aligned}$$

F.6 Inégalité des différences bornées

L'inégalité des différences bornées (ou de McDiarmid), borne la probabilité d'observer une grande différence entre une fonction de plusieurs variables et son espérance lorsque l'effet d'une variation d'une seule variable sur la fonction est borné.

Pour une fonction $f : \mathcal{Z}^n \to \mathbb{R}$ de n variables telle que pour des constantes non négatives $c_i \geq 0$, $i = 1, \ldots, n$,

$$\sup_{(z_1,\ldots,z_n)\in\mathcal{Z}^n, z_i'\in\mathcal{Z}} |f(z_1, \ldots, z_n) - f(z_1, \ldots, z_{i-1}, z_i', z_{i+1}, \ldots, z_n)| \leq c_i, \quad i = 1, \ldots, n,$$

soit valide et n variables aléatoires Z_i indépendantes (mais pas nécessairement identiquement distribuées),

$$P\left\{f(Z_1, \ldots, Z_n) - \mathbb{E}f(Z_1, \ldots, Z_n) \geq \epsilon\right\} \leq \exp\left(\frac{-2\epsilon^2}{\sum_{i=1}^{n} c_i^2}\right),$$

$$P\left\{\mathbb{E}f(Z_1, \ldots, Z_n) - f(Z_1, \ldots, Z_n) \geq \epsilon\right\} \leq \exp\left(\frac{-2\epsilon^2}{\sum_{i=1}^{n} c_i^2}\right),$$

et

$$P\left\{|f(Z_1, \ldots, Z_n) - \mathbb{E}f(Z_1, \ldots, Z_n)| \geq \epsilon\right\} \leq 2\exp\left(\frac{-2\epsilon^2}{\sum_{i=1}^{n} c_i^2}\right).$$