

MI201: Introduction à l' apprentissage automatique

Gianni Franchi, Stephane Herbin, Adrien Chan Hon Tong

Ecrit par Gianni Franchi et Estéban Rousseau

Contents

1 Apprentissage automatique: introduction	7
1.1 Généralités	7
1.2 Exemples d'application	7
1.3 Problématique générale	8
1.3.1 Point de vue du cours	8
1.3.2 Formalisation mathématique	8
1.3.3 Description d'un premier exemple étape par étape	11
1.4 Extraction des caractéristiques	14
1.4.1 Travailler avec les données	14
1.4.2 Problématiques de l'extraction de caractéristiques	15
1.4.3 Chaîne de prédiction générique	16
1.5 Un concept central: la généralisation	17
1.5.1 Présentation des modèles linéaires généralisés	18
1.5.2 Différence entre les données d'apprentissage (training) et les données opérationnelles (test):	20
1.5.3 L'erreur de généralisation	21
1.5.4 Résumé de la construction d'un chantier d'apprentissage	21
1.5.5 Le modèle de régression linéaire en Python	22
1.6 Modélisation bayésienne	23
1.6.1 Approche bayésienne multivariée:	24
1.6.2 Approche bayésienne naïve	25
1.6.3 Résumé concernant l'approche bayésienne	26
1.6.4 L'approche bayésienne en Python	26
1.7 La classification Plus Proche Voisin (PPV)	28
1.7.1 Principe général de l'approche 1-PPV	28
1.7.2 L'approche des k-plus proches voisins (dite "k-NN")	30
1.7.3 Le coût de la prédiction k-ppv	31
1.7.4 La malédiction des grandes dimensions	31
1.7.5 Le comportement des approches PPV	31
1.7.6 Résumé concernant l'approche plus proches voisins	32
1.8 L'essentiel à retenir du chapitre	32
2 Arbres de décision et méthodes ensemblistes	33
2.1 Les arbres de décision	33
2.1.1 Des graphes aux arbres	33
2.1.2 Présentation de la structure d'arbre	33
2.1.3 Principe général	34

2.1.4	Un premier exemple	35
2.1.5	Les différentes questions possibles	36
2.1.6	Formalisation de la structure d'un arbre de décision	36
2.1.7	Apprentissage hiérarchique	36
2.1.8	Algorithme de construction d'un arbre de décisions	37
2.1.9	Résumé concernant les arbres de décision	41
2.2	Introduction des méthodes ensemblistes	42
2.3	Bagging	42
2.4	Les forêts aléatoires (ou Random Forests)	44
2.4.1	Algorithme de construction d'une forêt aléatoire	44
2.4.2	Biais et variance	45
2.4.3	Résumé concernant les Random Forests	46
2.5	Boosting	47
2.5.1	Principe de la méthode du boosting	47
2.5.2	L'algorithme Adaboost	47
2.5.3	Gradient Boosting	48
2.5.4	Résumé concernant le Boosting	49
3	Régularisation / SVM	51
3.1	Régularisation	51
3.1.1	Définition de la régularisation	52
3.1.2	Modélisation mathématique de la régularisation	52
3.1.3	Résultats de la régularisation	53
3.2	Validation croisée	54
3.2.1	Qu'est-ce que la validation croisée ?	54
3.2.2	Les intérêts de la validation croisée	55
3.2.3	La validation croisée stratifiée	56
3.2.4	La validation croisée <i>leave-one-out</i>	56
3.3	Les "Support Vector Machines" (ou SVM)	56
3.3.1	Position du problème	57
3.3.2	Choix de l'hyperplan	57
3.3.3	Formulation de la SVM à marge rigide	58
3.3.4	Formulation de la SVM à marge souple	60
3.3.5	Forme duale du SVM à marge souple	62
3.3.6	Données non linéairement séparables	65
3.3.7	Les noyaux dans les SVM	66
3.3.8	Résumé concernant les SVM	67
3.4	Multiclasse	68
3.4.1	Le One vs One	68
3.4.2	Le One vs Rest	69
3.4.3	Evaluation du multiclasse	69
4	Introduction aux réseaux de neurones	71
4.1	Le modèle du neurone	71
4.1.1	Motivation du modèle	71
4.1.2	Passage du neurone au réseau	72
4.1.3	Les réseaux de neurones et les SVM	74
4.1.4	Preuves de l'universalité des réseaux <i>relu</i>	74

4.2	Apprentissage des réseaux <i>relu</i>	76
4.2.1	Une erreur à éviter : apprendre la base de données par cœur	76
4.2.2	L'algorithme perceptron	77
4.2.3	Un nouveau paradigme	80
4.3	La descente de gradient stochastique	82
4.3.1	La descente de gradient	82
4.3.2	Les limites de la descente de gradient	83
4.3.3	L'algorithme du gradient stochastique	84
4.3.4	La méthode mini Batch SGD	84
4.4	Le réseau neuronal convolutif	85
4.4.1	Le neurone convolutif	86
4.4.2	Le pooling	87
4.4.3	Les réseaux	87
5	Unsupervised Learning	89
5.1	Réduction de dimension	89
5.1.1	La malédiction des grandes dimensions	90
5.1.2	Analyse en composantes principales (ou <i>Principal Component Analysis</i>)	91
5.1.3	Utilisation de noyaux (kernels)	94
5.1.4	Positionnement multidimensionnel (MDS)	97
5.1.5	Algorithme t-SNE	101
5.1.6	Auto-encodeur	102
5.2	Clustering	104
5.2.1	Algorithme K-means	104
5.2.2	DBSCAN	107
Bibliography		111

Chapter 1

Apprentissage automatique: introduction

1.1 Généralités

L'apprentissage automatique, ou Machine Learning, est un domaine très éclectique. En effet il met notamment à contribution des notions de statistique, d'intelligence artificielle, de *computer science* ou encore de traitement du signal. De plus les techniques utilisées dans ce domaine sont également généralistes puisqu'elles font intervenir de la gestion de base de données, de l'optimisation numérique ainsi que du hardware.

Le Machine Learning est un secteur en plein essor avec l'avènement de l'intelligence artificielle qui a amené de nouveaux domaines comme le *deep learning* ou encore le *big data*. Cet engouement résulte de l'incapacité à modéliser des problèmes trop complexes malgré un grand nombre d'exemples à disposition. D'un point de vue scientifique, le Machine Learning est essentiel puisqu'il permet de donner une forme d'apprentissage aux machines, qui est une des facultés les plus essentielles des êtres vivants. Enfin le Machine Learning devenait nécessaire en économie où la récolte de données est bien moins contraignante que le développement d'expertise.

Au-delà des raisons qui ont poussé le Machine Learning à se développer, de nombreux domaines techniques lui ont trouvé une utilité. Tout d'abord il peut être utilisé comme un outil de conception en robotique, en data mining, pour la recherche dans des bases de données, en vision et reconnaissance des formes par vision par ordinateur ou dans le traitement de la parole et du langage. Dans un second temps le Machine Learning peut se révéler comme un outil explicatif, notamment en neurosciences ou en psychologie.

1.2 Exemples d'application

Les exemples d'applications les plus connus et qui servent en quelque sorte de vitrine aux progrès du Machine Learning sont les jeux. En effet, même si le terme a été employé pour la première fois en 1959 par l'informaticien américain Arthur Samuel, c'est en 1997 que le grand public découvre véritablement l'existence et le potentiel de l'intelligence artificielle et du Machine Learning lorsque l'IA *DeepBlue* bat aux échecs le champion du monde incontesté de l'époque Garry Kasparov [1]. Suite à cet accomplissement les différents jeux de réflexion de ce type vont tous tomber un par un jusqu'à ce qu'il ne reste plus que le jeu de GO, qui résistera pendant quelques années avant que le champion du monde Ke Jie soit finalement battu par *Alpha GO* en 2017[2].

Une fois que les jeux de réflexion classiques ont tous été dominés par les machines, les développeurs de tels programmes se sont penchés sur les jeux vidéo. Ainsi, à peine deux ans après le jeu de GO, *AlphaStar* devenait champion de StarCraft, un célèbre jeu de stratégie[3].

Derrière cette vitrine qui a fait connaître le Machine Learning et plus généralement l'intelligence artificielle au grand public, l'apprentissage automatique s'est peu à peu implanté dans notre quotidien, si bien qu'aujourd'hui il fait partie intégrante de notre quotidien sans que nous ne nous en rendions forcément compte. S'il est parfois facile à déceler, comme un système anti-spam pour boîte mail, un chat bot ou la détection de visages sur un appareil photo, il peut aussi nous affecter de manière plus discrète, avec notamment les recommandations (ou publicités) ciblées en fonction des recherches effectuées récemment sur internet.

1.3 Problématique générale

1.3.1 Point de vue du cours

Dans ce cours, nous appréhendrons l'apprentissage automatique (la dénomination Machine Learning sera aussi utilisée pour éviter la redondance) comme une démarche de conception d'une fonction de prédiction à partir d'une base de données préexistante ou qu'il faudra constituer. Cette démarche devra être effectuée par une modélisation, ou programmation non explicite, sur les exemples de cette base qui pourront être de natures très variées comme des images, des signaux, des textes, des mesures, etc...

1.3.2 Formalisation mathématique

Formalisation générale

Avant d'entrer dans le vif du sujet, il est nécessaire de formaliser mathématiquement les différents éléments qui seront utiles tout au long du cours. On peut formaliser un problème d'apprentissage supervisé de la façon suivante : étant données n observations $\{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n\}$, avec $\forall i, \mathbf{x}_i \in \mathcal{X}$ où \mathcal{X} est l'espace des observations, et les *predictions* ou *étiquettes* associées à ces observations $\{y_1, y_2, \dots, y_n\}$, avec $\forall i, y_i \in \mathcal{Y}$ où \mathcal{Y} est l'espace des étiquettes, l'objectif principal de l'apprentissage automatique supervisé est de déterminer une fonction $f : \mathcal{X} \rightarrow \mathcal{Y}$ telle que $\forall (\mathbf{x}_i, y_i) \in \mathcal{X} \times \mathcal{Y}$ ayant la même relation que les paires observées, $\arg \max_y \{f(\mathbf{x}_i)\} = y_i$. On notera $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1, \dots, n}$ l'échantillon, l'ensemble d'apprentissage, ou encore jeu d'apprentissage.

Comme indiqué précédemment, les données peuvent être de types très différents et il est nécessaire de faire un travail de modélisation de ces dernières en amont pour chaque problème différent. Dans de nombreuses situations on se ramènera néanmoins au cas où $\mathcal{X} = \mathbb{R}^p$ et on dira que les observations sont représentées par p variables. Dans ce cas on appellera matrice de données ou matrice de design la matrice $X \in \mathbb{R}^{n \times p}$ telle que $X_{ij} = x_{ij}$, c'est-à-dire la j -ème composante de la i -ème observation.

Ce que l'on décrit comme prédiction peut être tout aussi varié. Il peut s'agir d'une décision, d'une action, d'une réponse, d'un groupe, d'une commande, d'une valeur, d'une préférence, etc... Ainsi la phase de modélisation de cette prédiction est tout aussi importante que celle des données et dépend également du problème traité.

Enfin l'échantillon correspond aux exemples de données et de prédictions (pouvant être bonnes ou mauvaises) dont on dispose. Il s'agit en fait de la base de données évoquée en **1.3.1** et qui va servir de base d'apprentissage pour déterminer la prédiction à partir des données. On fait ici l'hypothèse (très) forte que les échantillons exemples contiennent toute l'information exploitable et utile.

Cette interpolation se déroule en deux phases distinctes que sont l'apprentissage et la prédiction qui peuvent être représentées schématiquement et de manière plus détaillées sur la figure suivante:

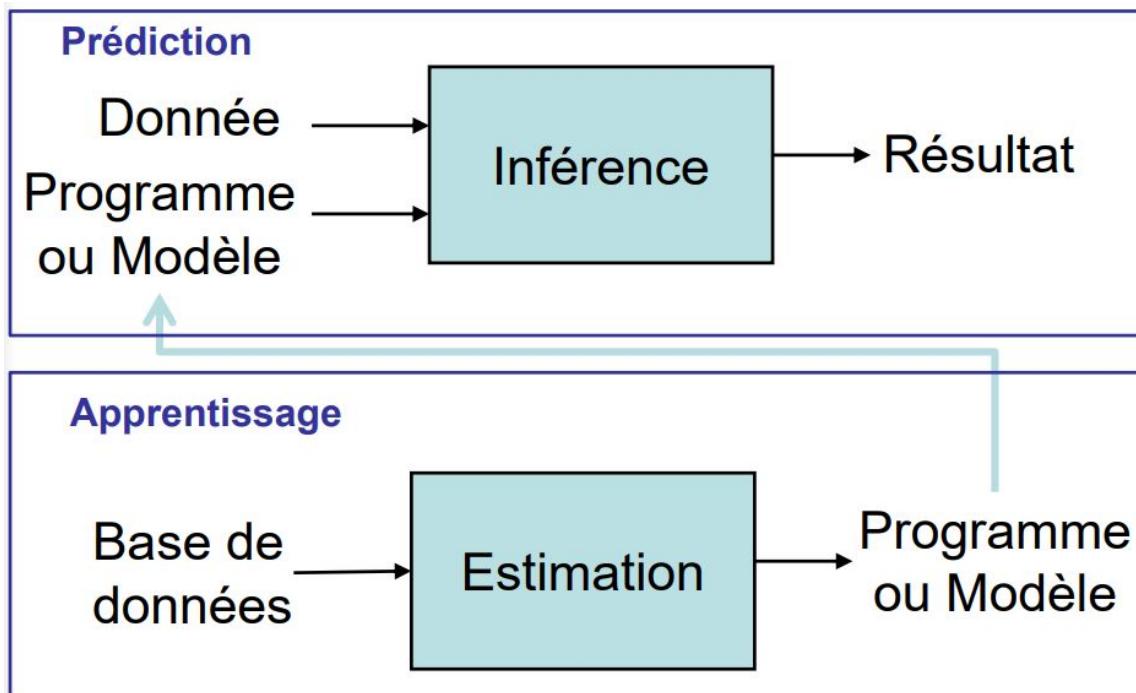


Figure 1.1: Schéma de principe de l'apprentissage automatique

Ainsi la phase d'apprentissage consiste à trouver les paramètres ω de telle sorte que $y_i = f(\mathbf{x}_i, \omega) \forall (\mathbf{x}_i, y_i) \in \mathcal{D}$. On utilisera ensuite ω pendant la phase de prédiction sur de nouveaux exemples.

Le but de la phase d'apprentissage devient alors de rechercher le ω qui permet d'optimiser un certain critère L que nous détaillerons plus tard dans le paragraphe **1.5**:

$$\omega = \arg \min_{\omega'} L(\mathcal{D}, \omega')$$

Les différents types d'apprentissages

Il existe plusieurs formes d'apprentissages dans le domaine du Machine Learning. Parmi elles se trouvent les apprentissages supervisé et non supervisé. La principale différence entre ces deux types d'apprentissages est que, dans le cadre du premier, on a une connaissance préalable de ce que devraient être les valeurs de sortie de nos échantillons. L'algorithme peut s'entraîner sur une base de données dédiée pour laquelle on connaît les "bonnes" réponses avant de pouvoir s'appliquer sur d'autres données.

Pour l'apprentissage non supervisé, c'est un peu plus compliqué: il ne dispose pas de réponse correcte ni d'enseignant. En effet sa tâche est de déterminer lui-même une modélisation de la structure des données pour en extraire une information.

Des deux opposés précédents résulte l'apprentissage semi-supervisé pour lequel on possède une base dont on connaît partiellement les valeurs de sortie.

On peut également citer l'apprentissage par transfert pour lequel on utilise des données d'apprentissage proches du problème que l'on souhaite résoudre: on utilise une situation pour laquelle on a déjà des exemples pour en étudier une proche pour laquelle on manque de données, ce qui évite d'avoir à construire une base d'apprentissage et fait gagner du temps.

Enfin on peut citer l'apprentissage par renforcement pour lequel les prédictions sont issues d'une séquence d'actions et sont plus ou moins bien récompensées. Le programme ne reçoit pas de données d'entraînement mais est envoyé dans un environnement et prend ses décisions au fur et à mesure en fonction de son état courant. Ces décisions vont lui permettre d'obtenir des récompenses qui le guideront dans la suite de son apprentissage. Cette technique est notamment utilisée par *Alpha GO Zero*, un programme de jeu de GO qui, en partant de rien, est devenu plus fort qu'*Alpha GO* en seulement quelques jours![4]

Les différents types de prédictions

Tout comme la phase d'apprentissage, la phase de prédiction est elle aussi subdivisées en plusieurs catégories. La première d'entre elles est la classification. Comme son nom l'indique cette prédiction doit déterminer à quelle classe appartient la donnée en entrée. Il peut s'agir d'une prédiction binaire (si un mail est un spam ou non) ou d'une identification (reconnaissance d'une lettre manuscrite). Ce cas est caractérisé par un espace des étiquettes $\mathcal{Y} = \{1, 2, \dots, C\}$, avec $C \geq 2$.

Le deuxième type de prédiction qui vient naturellement à l'esprit est la régression, qui regroupe par exemple les prédictions de températures, de cours de bourse ou encore la localisation d'un objet dans une image. Ici la prédiction doit évaluer une variable continue, en opposition avec la classification qui est fondamentalement la prédiction d'une valeur discrète. Ici l'espace des étiquettes est $\mathcal{Y} = \mathbb{R}$.

Les prédictions peuvent être plus complexes avec les prédictions de structure où l'algorithme cherche à déterminer des objets structurels plus volumineux comme des arbres ou des graphes, car ces derniers ont une forte capacité de représentation d'objets réels comme les articulations d'un être humain par exemple.

Le Machine Learning peut également permettre de regrouper des données en fonction de leurs caractéristiques: ce sont les prédictions de type regroupement. On peut par exemple regrouper des photos de voitures en fonctions de leur couleur ou encore classer les selfies en fonction des personnes présentes.

Pour finir les prédictions peuvent prendre la forme d'un texte dans le cas où les données sont des images ou des vidéos. Ainsi une vidéo pourrait donner la prédiction "C'est un chat qui saute sur une table".

1.3.3 Description d'un premier exemple étape par étape

Afin d'illustrer notre propos et de décomposer la marche à suivre pour appréhender un problème, nous allons traiter de la problématique de la reconnaissance de chiffres manuscrits sur des enveloppes postales américaines:

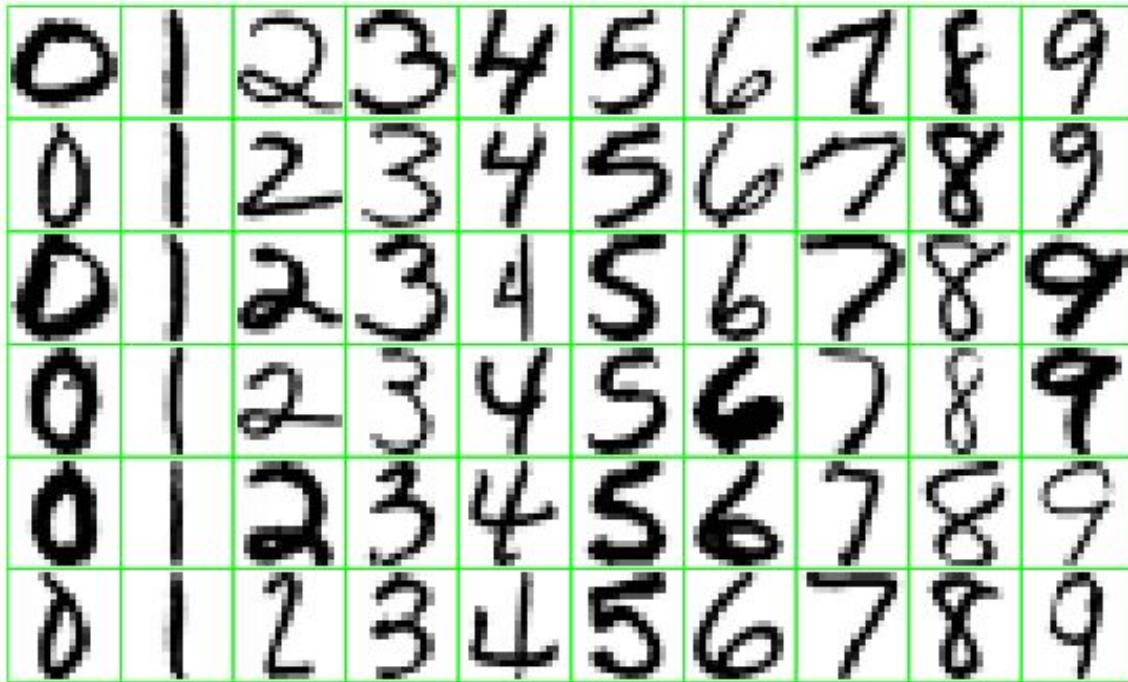


Figure 1.2: Exemples de chiffres manuscrits d'enveloppes postales américaines [5]

La première difficulté est de savoir comment définir les différents éléments \mathcal{D} , ω , x et y ainsi que les fonctions d'apprentissage, permettant d'obtenir ω à partir de \mathcal{D} , et de prédiction $f(x; \omega)$.

Première étape: choisir une base de données

Pour commencer on choisit donc une base de données \mathcal{D} qui servira de base d'apprentissage pour la suite. Deux cas de figure sont alors possibles: soit des bases existent déjà (c'est le cas sur cet exemple) et dans ce cas on peut en utiliser une pour gagner du temps soit il faut la construire en utilisant un recueil de données existantes ou en expérimentant soi-même.

Deuxième étape: mettre en forme les données

L'objectif de cette étape est de transformer les données initiales, qui ne sont que de simples images en deux dimensions où apparaît du bruit et qui sont très volumineuses (une valeur par pixel de l'image), en une forme plus exploitable dans laquelle les données auront un format identique et seront d'une dimension plus raisonnable.

Pour cela on ne va plus décrire une image pixel par pixel mais plutôt la décrire en utilisant certaines de ses caractéristiques. Dans notre exemple des chiffres manuscrits on peut prendre comme critère les profils de l'image, qui nous donnent accès aux contours du chiffre en question : on crée 4 images blanches et on colorie chacune de ces images tant qu'on ne rencontre pas un pixel noir en partant d'une direction différente (voir exemple ci-dessous), au nombre de cavités sur l'image pour pouvoir l'associer à un résultat, enfin on pourrait découper l'image en sous-sections et observer lesquelles sont occupées:

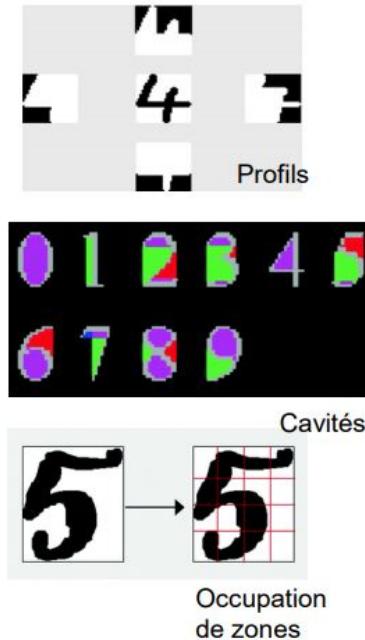


Figure 1.3: Trois exemples de critères pour la reconnaissance de chiffres manuscrits

L'utilisation de tels critères permet d'obtenir à terme des données sous forme de vecteurs de petite dimension permettant un temps de calcul moins exorbitant.

Troisième étape: choisir l'approche du problème

Cette étape a pour but de déterminer quels sont les éléments du problème traité pouvant être pris en compte dans le choix de la nature du prédicteur : quel est le type d'apprentissage? quel est le type de prédiction? quelle est la forme des données?...

Chacune de ces informations va avoir une influence sur le choix de la nature du prédicteur, car chacune de ces approches a ses points forts et ses points faibles (puissance et temps de calcul nécessaires, plus efficace sur un problème en petite/grande dimension...). Il faut alors essayer de trouver la nature de prédicteur qui sera la plus apte à répondre correctement au problème posé. Maintenant que l'on dispose des matériaux de base pour notre exemple, il est temps de créer les outils qui vont nous permettre de les travailler pour obtenir un prédicteur efficace. Le problème de prédiction est de type classification: chaque donnée est associée à une classe appartenant à $\mathcal{G} = \{0, 1, \dots, 9\}$. De plus on connaît les classes cibles, on est donc dans le cadre d'un apprentissage supervisé.

Les données sont représentées par des vecteurs, certes de taille fixe, mais qui restent de grande taille malgré la description des images par certains critères. Ces caractéristiques demandent des algorithmes ayant un bon contrôle de la régularisation, c'est-à-dire l'ajout d'information permettant d'éviter le surapprentissage (ou overfitting)¹.

¹Cette notion est définie en 1.5.3

La base de données est de grande taille (≥ 10000 exemples) car il en existe plusieurs pour ce problème ce qui va rendre notre optimisation efficace.

La nature fonctionnelle du prédicteur dépend de la forme des données ainsi que du type de prédiction. Parmi les principales natures on peut citer: les plus proches voisins, les ensembles de classifieurs (forêts aléatoires, "boosting"...), la programmation logique, les modèles probabilistes (réseaux bayésiens, chaînes de Markov...), les machines à vecteurs de supports (ou SVM), les réseaux de neurones ou encore les arbres de décisions. Les spécificités de ces différentes approches seront détaillées dans les chapitres dédiés.

Quatrième étape: optimisation

L'apprentissage nécessite de définir un espace fonctionnel et un critère paramétrique, comme le coût ou l'énergie par exemple. En outre on applique un optimiseur et on en règle les paramètres avant de vérifier que l'apprentissage fonctionne bien en évaluant sa capacité de généralisation ainsi que sa convergence.

On peut citer quatre grands types d'optimisations: l'optimisation convexe pour minimiser un critère convexe dans un ensemble lui aussi convexe comme pour la minimisation séquentielle d'un problème quadratique; l'optimisation stochastique dans le cas où certaines contraintes dépendent de variables aléatoires, avec l'exemple de la descente de gradient stochastique ou celui des algorithmes génétiques; l'optimisation sous contraintes où l'on évalue l'ensemble des scénarios possibles pour un problème donné pour en déduire les chemins puis les choix optimaux comme en programmation linéaire; l'optimisation combinatoire consiste à trouver un des meilleurs sous-ensembles réalisables, c'est le principe des algorithmes gloutons.

Cinquième étape: évaluation

Pour pouvoir évaluer l'efficacité de l'apprentissage réalisé, il est nécessaire d'introduire une métrique d'évaluation. Il en existe de nombreuses pouvant avoir différentes formes. La plus simple est le taux d'erreur moyen que l'on peut définir comme ceci :

$$\mathcal{E} = \frac{\text{nombre d'étiquettes erronées}}{\text{nombre total de données}}$$

Dans le cadre d'un problème de classification, on peut aussi utiliser une matrice de confusion. Chaque composante de cette matrice $M = m_{i,j}$ correspond au nombre d'éléments qui sont de la classe i ayant été estimés comme appartenant à la classe j . Une telle matrice permet à la fois de voir si le programme parvient à classifier correctement et quelles sont les erreurs les plus fréquentes. Pour les classifications binaires on construit fréquemment une courbe ROC (Receiver Operator Characteristic), notamment dans le domaine médical pour évaluer l'efficacité d'un test de détection d'un virus ou d'une bactérie. Une telle courbe s'obtient à partir du taux positif vrai, c'est-à-dire la proportion d'observations qui ont été correctement prédites positives :

$$TPV = \frac{p_v}{p_v + n_f}$$

Où p_v correspond au nombre de cas prédits positifs qui le sont réellement et n_f le nombre de cas prédits négatifs qui sont en réalité positifs. De même on définit le taux faux positif :

$$TFP = \frac{p_f}{n_v + p_f}$$

Avec p_f les prédicts positifs qui sont négatifs et n_v les négatifs qui le sont vraiment. Pour construire la courbe on fait alors varier le seuil de score à partir duquel le classificateur (qui doit être probabiliste) prédit qu'une donnée appartient à une classe plutôt qu'à l'autre :

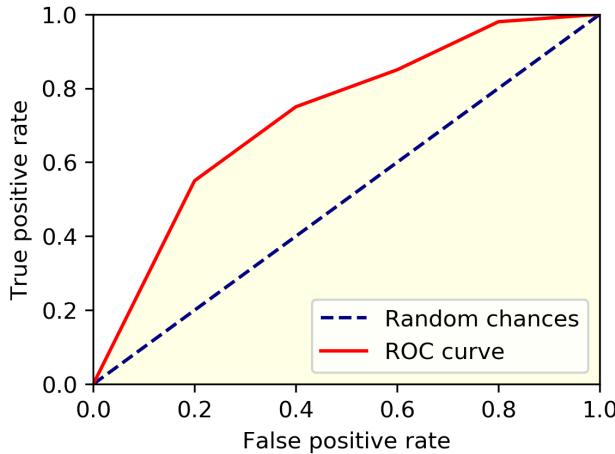


Figure 1.4: Exemple de courbe ROC[6]

Une fois une telle courbe obtenue, l'aire sous la courbe indique la qualité de notre prédicteur. Pour ce qui est des problèmes de type régression, on utilise en général l'erreur quadratique :

$$\hat{R}_n(f)^1 = \frac{1}{n} \sum_{i=1}^n (f(\mathbf{x}_i) - y_i)^2$$

Conclusion

La résolution d'un problème par apprentissage automatique demande une certaine rigueur et doit se tenir à une feuille de route pour pouvoir progresser. Ces étapes sont cruciales puisque chacune d'elles se construit sur le modèle bâti par les précédentes. Enfin, il s'agit d'une méthode itérative : si le modèle doit être modifié, il est nécessaire de revérifier le travail effectué au sein de chaque étape pour que le nouveau modèle puisse coller au formatage des données et aux caractéristiques que ce format permet d'extraire.

1.4 Extraction des caractéristiques

1.4.1 Travailler avec les données

Pour pouvoir utiliser des données dans le cadre de l'apprentissage automatique, il faut passer par deux étapes indispensables et complémentaires. Il faut commencer par préparer les données, c'est une étape certes coûteuse mais toute aussi importante puisque c'est ce travail de mise en forme qui rend les données exploitable pour un programme. Ce passage obligatoire en Machine Learning est si crucial qu'il a même engendré la création du métier de Data engineer, un ingénieur spécialisé sur les applications pratiques de la collecte de données et de leur analyse.

De ce travail de préparation découle la transformation des données. L'objectif principal est alors d'extraire des données formalisées les informations qu'elles peuvent apporter. On parle de caractéristiques (ou "features" en anglais).

¹Cette fonction de coût sera introduite dans la section 1.5

1.4.2 Problématiques de l'extraction de caractéristiques

Comme nous l'avons déjà vu dans l'exemple de la reconnaissance de chiffres manuscrits, les données brutes contenues dans les bases de données ne sont pas exploitables directement par l'algorithme. Ces données brutes sont très souvent sujettes au bruit et leur dimension est trop grande pour espérer obtenir un résultat probant en un temps raisonnable. En outre les données de notre exemple ne sont que des images d'une taille relativement petite dont les pixels ne peuvent être que noirs ou blancs. Ces données sont déjà trop conséquentes alors qu'il existe une multitude de problèmes sur des éléments infiniment plus volumineux comme des images en haute résolution et en couleurs voire sur une vidéo qui est une succession d'images de ce type. Les données brutes sont donc volumineuses et imprécises ce qui noie l'information utile dans un océan de superflu.

La définition de ces caractéristiques est notamment cruciale pour tout problème de reconnaissance des formes qui, par définition, travaille sur des données brutes initiales sous forme d'images. Tout est une question d'équilibre entre de nombreux critères incompatibles, symbolisé par l'opposition entre la robustesse et le coût de calcul. En effet si on privilégie la robustesse en voulant utiliser des caractéristiques trop riches, notre algorithme aura une complexité très élevée qui réduira considérablement son efficacité. En plus de cela on resterait sensible au bruit et le programme aurait une grande variabilité.

D'un autre côté, si l'on privilégie un coût de calcul faible en usant de caractéristiques trop simples, il est tout à fait logique et moral que l'on obtienne de nombreuses confusions, puisqu'on ne disposera alors que de peu d'informations par manque de travail sur les données.

Il existe alors deux cas de figure pour définir les caractéristiques que l'on souhaite extraire de la masse de données brutes. Soit on sait quelles sont les caractéristiques importantes et pourquoi grâce à une expertise métier. C'est-à-dire que les expériences passées sur le sujet en question ou sur des problèmes proches ont permis de déterminer quelles informations étaient pertinentes à prendre en ligne de compte. On peut alors directement modéliser notre problème en formalisant les données brutes conformément à ces caractéristiques pertinentes.

Si, au contraire, on ne dispose d'aucun indice sur les points importants à prendre en compte, alors il est inévitable de devoir passer par une phase d'apprentissage de ces critères importants en faisant différents tests. Ce n'est qu'après avoir effectué ce travail supplémentaire que l'on peut passer à la modélisation du problème posé.

1.4.3 Chaîne de prédiction générique

Les algorithmes d'apprentissage automatique suivent tous plus ou moins une certaine feuille de route. En effet la démarche générale reste toujours fixe et pourrait être représentée comme suit pour l'exemple de reconnaissance de chiffres:

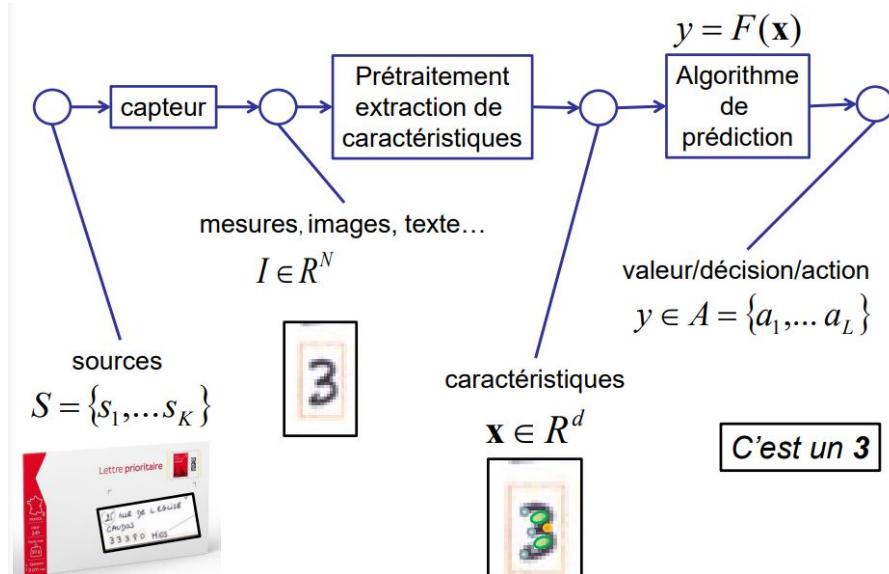


Figure 1.5: Chaîne de prédiction générique

Ce schéma montre bien la première phase de construction d'une représentation des données jusqu'à ce que l'on obtienne les caractéristiques de ces dernières en sortie de la phase de prétraitement. Ce n'est qu'après avoir obtenu ces informations que l'on peut passer à la phase de prédiction à proprement parler pendant laquelle les renseignements extraits sont passés dans l'algorithme de prédiction.

Exemples de caractéristiques

Les caractéristiques que l'on peut observer sont divisées en deux grandes classes. Dans l'exemple de reconnaissance de chiffres manuscrits, les caractéristiques retenues sont de type forme. Elles sont détectées grâce à un contraste avec le fond de l'image, il peut s'agir de segmentations ou de saillances comme par exemple les extémités des chiffres (les endroits où l'on pose ou lève le stylo). Il peut également s'agir de caractéristiques structurelles.

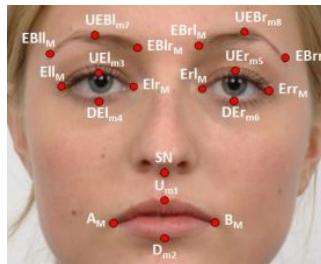


Figure 1.6: Exemple de détection de caractéristiques de formes [7]

Les autres caractéristiques d'images sont celles de texture. Elles sont plus difficiles à associer à un objet précis. En effet cette analyse vise à renvoyer des informations sur l'arrangement spatial d'une image en utilisant ses couleurs. Ces caractéristiques peuvent être notamment utiles dans le domaine de l'imagerie médicale puisque ce type d'analyse permet de délimiter les contours de certains éléments aux textures différentes du reste de l'image:



Figure 1.7: Exemple de détection d'une caractéristique de texture sur une imagerie Doppler

Une approche plus moderne: le Deep Learning

Aujourd'hui la démarche présentée précédemment laisse de plus en plus la place à une nouvelle vision de la chaîne de prédiction. Désormais il est devenu commun de prédire tout en extrayant les données et l'on utilise alors des caractéristiques appelées "Deep features". Pour être clair il s'agit d'apprendre les caractéristiques de l'image via des réseaux convolutifs, qui sont développés dans le chapitre 5 consacré au Deep Learning.

1.5 Un concept central: la généralisation

Pour obtenir un prédicteur de bonne qualité, il ne faut pas juste que le programme se contente "d'apprendre par cœur" les résultats de l'échantillon, il faut utiliser ces données bien sûr mais dans l'optique de faire face à de nouveaux cas de figure. Mais tout d'abord il est nécessaire d'introduire quelques notions avant d'aller plus loin sur le sujet de la généralisation.

Il faut commencer par définir une *fonction de coût*, il s'agit d'une fonction $L : \mathcal{Y} \times \mathcal{Y} \rightarrow \mathbb{R}$ utilisée pour quantifier la qualité d'une prédiction : $L(y, f(\mathbf{x}))$ est proportionnel à l'éloignement entre les valeurs de $f(\mathbf{x})$ et de y . On cherche donc f qui minimise cette fonction de coût.

En Machine Learning, on appelle *risque* l'espérance d'une fonction de coût:

$$\mathcal{R}(f) = \mathbb{E}_{\mathcal{X} \times \mathcal{Y}}[L(y, f(\mathbf{x}, \boldsymbol{\omega}))] = \int_{\mathcal{X} \times \mathcal{Y}} L(y, f(\mathbf{x}, \boldsymbol{\omega})) dP(\mathbf{x}, y)$$

Où $dP(\mathbf{x}, y)$ est la probabilité jointe de \mathbf{x} et y . On en déduit que la fonction \hat{f} recherchée vérifie:

$$\hat{f} = \arg \min_f \mathbb{E}_{\mathcal{X} \times \mathcal{Y}}[L(y, f(\mathbf{x}))]$$

Cependant ce problème reste généralement insoluble sans l'addition d'hypothèses. On approchera cette espérance par l'estimateur de la moyenne empirique, ce qui nous ammène la définition du *risque empirique*:

$$\widehat{\mathcal{R}}_n(f) = \frac{1}{n} \sum_{i=1}^n L(y_i, f(\mathbf{x}_i))$$

D'où le prédicteur par *minimisation du risque empirique*:

$$\widehat{f} = \arg \min_f \frac{1}{n} \sum_{i=1}^n L(y_i, f(\mathbf{x}_i))$$

Pour ce qui est des fonctions de coût en elles-mêmes, il en existe une multitude mais on peut citer la fonction de coût 0/1 pour la classification binaire:

$$L_{0/1} : \mathcal{Y} \times \mathcal{Y} \rightarrow \mathbb{R}$$

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

Le risque empirique vaut alors le nombre moyen d'erreurs de prédiction.

Afin d'illuster le concept de généralisation, nous allons nous intéresser à l'exemple de la régression polynomiale. Nous cherchons ici à approximer une fonction non polynomiale par un polynôme à partir de valeurs uniformément échantillonées en \mathbf{x} mais bruitées en y . Pour évaluer la pertinence de la fonction polynomiale on calculera une une erreur de régression qui sera la distance au carré entre les points échantillonés et le polynôme estimé.

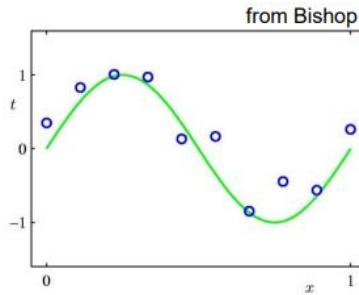


Figure 1.8: Exemple de courbe à approximer (en vert) grâce aux données échantillonées (en bleu) [8]

1.5.1 Présentation des modèles linéaires généralisés

La prédiction utilisera des fonctions de base de la régression polynomiale permettant d'encoder les données sources: $\phi_k(\mathbf{x}) = x^k$

La fonction de prédiction sera donc de la forme:

$$y = f(x, \boldsymbol{\omega}) = \omega_0 + \omega_1 \phi_1(x) + \dots + \omega_M \phi_M(x)$$

Dans le cadre de l'apprentissage on utilisera un ω qui sera issu maximum de vraisemblance:

$$\omega_{ML} = (\Phi^T \Phi)^{-1} \Phi^T \mathbf{y}$$

$$\Phi = \begin{bmatrix} \phi_0(x_1) & \phi_1(x_1) & \cdots & \phi_{M-1}(x_1) \\ \phi_0(x_2) & \phi_1(x_2) & \cdots & \phi_{M-1}(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_0(x_n) & \phi_1(x_n) & \cdots & \phi_{M-1}(x_n) \end{bmatrix} \in \mathcal{M}_{n,M}(\mathbb{R})$$

Voici les résultats obtenus dans les cas où l'on utilise $M = 0$, $M = 1$, $M = 3$ et $M = 9$:

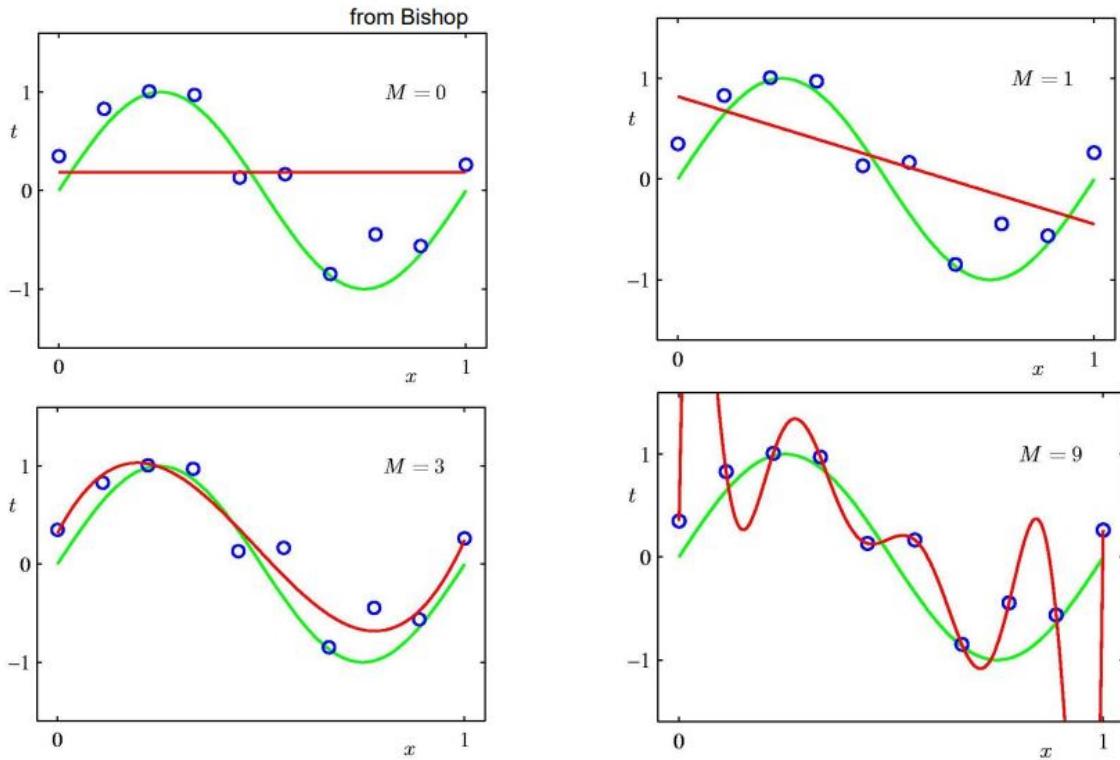


Figure 1.9: Approximation de la courbe (en rouge) en utilisant des fonctions polynomiales de degrés 0, 1, 3 et 9 [8]

Avec ces résultats vient une des idées les plus cruciales dans le domaine de l'apprentissage automatique:
il ne faut jamais faire aveuglément confiance à ses données!

En effet, si ces dernières servent de base à l'apprentissage du programme, elles ne résultent souvent que d'observation ou de mesures qui sont sujettes à une certaine marge d'erreur. Vouloir parfaitement coller à ses données provoque inévitablement un sur-apprentissage qui éloigne considérablement la prédiction de la réalité dans la quasi-totalité des cas. C'est une fois de plus une question de compromis entre l'utilisation des données, qui est nécessaire, et la conservation d'un certain détachement de celles-ci et de leurs erreurs.

1.5.2 Différence entre les données d'apprentissage (training) et les données opérationnelles (test):

L'erreur de régression, qui constitue la métrique d'évaluation dans cet exemple, est calculée sur des jeux de données $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}$ et s'obtient avec la formule suivante:

$$\hat{R}_n(\boldsymbol{\omega}) = \sum_{i=1}^n (\boldsymbol{\omega}^T \phi(\mathbf{x}_i) - y_i)^2$$

On voit que le risque fait intervenir la norme 2 entre $f_{\boldsymbol{\omega}}(\mathbf{x}_i)$ et y_i . Cependant on ne cherche pas à atteindre le même but en fonction des données que l'on évalue. Si l'on se place dans le cas de données d'apprentissage, certes on cherche toujours à minimiser l'erreur de régression, mais c'est dans l'optique d'affiner notre modélisation car l'on connaît déjà les "bonnes" prédictions sans avoir besoin de l'algorithme. Ces exemples servent à améliorer le modèle pour que le programme ait une capacité de prédiction plus précise.

Au contraire, sur des données opérationnelles, on se place dans une situation réelle. On a besoin que le programme nous donne une bonne prédiction sans lui donner la réponse. Ces données ne sont pas là pour entraîner à proprement parler le programme mais pour permettre de vérifier si un apprentissage sur les données d'entraînement est efficace ou non.

En pratique, dans ce problème de régression polynomiale, l'erreur en fonction du degré du polynôme estimateur a une évolution à peu près semblable entre les données d'entraînement et les données de test jusqu'à ce que l'on atteigne un état de surapprentissage.

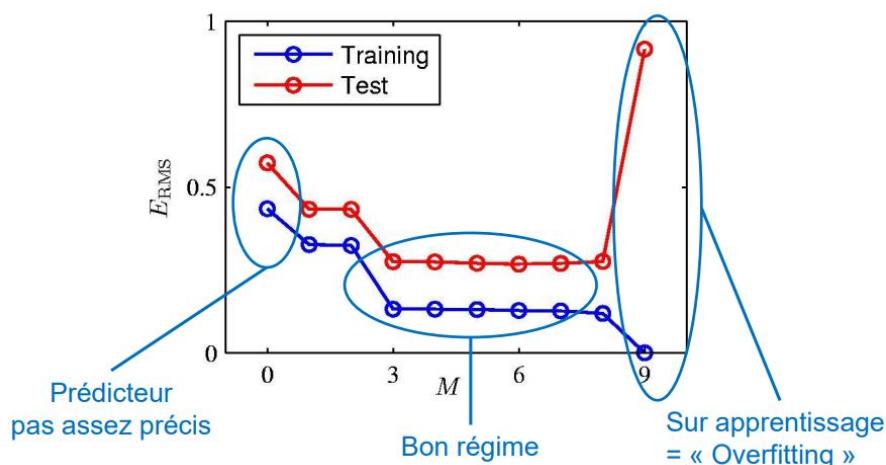


Figure 1.10: Graphique représentant l'évolution de l'erreur de régression en fonction du degré du polynôme approximateur [8]

Pour mesurer le bon fonctionnement du prédicteur on utilise l'erreur sur les données de test qui sont nouvelles pour lui. C'est ainsi que l'on peut vérifier que le prédicteur construit est capable de traiter les problèmes inconnus sans se contenter de mémoriser les résultats. Les données d'apprentissage sont les "cours" de la machine tandis que les données opérationnelles sont ses "exercices".

1.5.3 L'erreur de généralisation

La problématique de la généralisation aux données nouvelles, c'est-à-dire celles qui ne sont ni dans les données d'entraînement ni dans les données de test. Ainsi, il s'agit de données qui sont par définition inconnues. Autrement on s'en servirait pour l'apprentissage ou le test du prédicteur. Ainsi il est nécessaire de faire des hypothèses sur la nature de ces données et sur le modèle de prédiction.

Le graphique présenté en Figure 1.10 met en lumière les deux principaux phénomènes à éviter que l'on peut contrôler:

- le simplisme; on dit d'un modèle qui est trop simple pour avoir de bonnes performances même sur les données utilisées pour le construire qu'il *sous-apprend*, on parle d'*underfitting* en anglais.
- le sur-apprentissage; on dit d'un modèle qui, plutôt que de capturer la nature des objets à étiqueter, modélise aussi le bruit et ne sera pas en mesure de généraliser qu'il *sur-apprend*, on parle d'*overfitting* en anglais. Autrement dit l'algorithme apprend "par cœur" les résultats des données d'apprentissage, qui sont bruitées, plutôt que de "comprendre" réellement comment associer une prédiction correcte à n'importe quelle donnée.

Il est possible d'introduire la notion de *compromis biais-variance* pour mieux comprendre le risque d'un modèle $f : \mathcal{X} \rightarrow \mathcal{Y}$. Il suffit d'abord de comparer ce modèle à l'erreur minimale \mathcal{R}^* qui peut-être atteinte par n'importe quelle fonction mesurable de \mathcal{X} dans \mathcal{Y} pour obtenir ce que l'on appelle l'*excès d'erreur*:

$$\mathcal{R}(f) - \mathcal{R}^* = \left[\mathcal{R}(f) - \mathcal{R}_h(h) \right] + \left[\mathcal{R}_h(h) - \mathcal{R}^* \right]$$

Sous cette forme décomposée on obtient deux termes distincts, le premier quantifie la distance entre le modèle f et le meilleur modèle possible sur l'espace de fonctions considéré. Le second quantifie quant à lui la qualité du choix de cet espace défini par les différentes hypothèses faites lors de la création du modèle. Cette écriture permet de comprendre aisément qu'il y a un compromis à faire lors de la définition de l'espace des hypothèses. En effet plus celui-ci sera grand, plus il sera probable qu'il contienne une solution proche de la solution optimale. Cependant un espace plus grand implique que cette fonction sera plus difficile à déterminer. C'est ce compromis entre l'erreur d'approximation et l'erreur d'estimation qui porte le nom de *compromis biais-variance*.

1.5.4 Résumé de la construction d'un chantier d'apprentissage

Pour créer un chantier d'apprentissage qui sera apte à améliorer un programme de Machine Learning, on commence par préparer les données, en les faisant passer de résultats bruts bruités et de grande dimension à des informations simplifiées, formatées et homogénéisées.

On divise ensuite ces données préparées en deux grands ensembles: celui d'apprentissage, qui permet d'optimiser les paramètres du modèle, et celui de test, qui permet d'évaluer la qualité de l'apprentissage sur les données d'entraînement dans son contexte d'utilisation. Les tests permettent alors de connaître l'erreur de généralisation du prédicteur construit.

Enfin il faut garder à l'esprit de **ne jamais utiliser l'ensemble de test pour l'apprentissage**, sinon quoi l'on se retrouverait dans l'incapacité d'estimer la qualité du prédicteur sans avoir à recréer de nouvelles données.

1.5.5 Le modèle de régression linéaire en Python

Cette partie va nous permettre de voir comment on utilise ce modèle de régression en pratique avec la bibliothèque scikit-learn de Python. Commençons par la génération des puissances avec la fonction `PolynomialFeatures`:

```
from sklearn.preprocessing import PolynomialFeatures
x = np.array([2, 3, 4])
poly = PolynomialFeatures(3, include_bias=False)
poly.fit_transform(x[:, None])

array([[ 2.,  4.,  8.],
       [ 3.,  9., 27.],
       [ 4., 16., 64.]])
```

Figure 1.11: Exemple d'utilisation de `PolynomialFeatures` [9]

On peut ensuite utiliser `make_pipeline` pour créer un modèle de régression au degré souhaité, par exemple 7 dans l'exemple suivant:

```
from sklearn.pipeline import make_pipeline
poly_model = make_pipeline(PolynomialFeatures(7),
                           LinearRegression())
```

Figure 1.12: Exemple d'utilisation de `make_pipeline` [9]

On crée ensuite un échantillon de données puis on applique notre modèle. La phase d'apprentissage s'effectue avec la méthode `fit` et la phase de prédiction avec `predict`:

```
rng = np.random.RandomState(1)
x = 10 * rng.rand(50)
y = np.sin(x) + 0.1 * rng.randn(50)

poly_model.fit(x[:, np.newaxis], y)
yfit = poly_model.predict(xfit[:, np.newaxis])
```

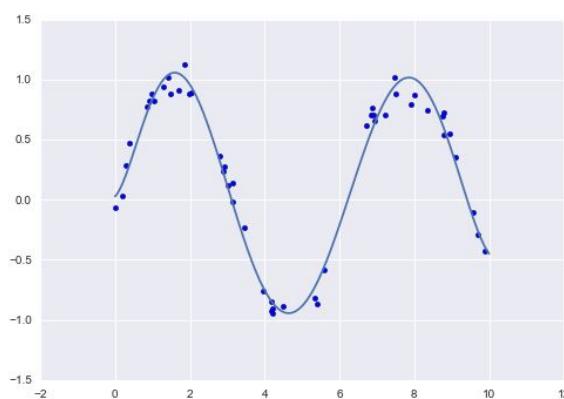


Figure 1.13: Exemple d'utilisation du modèle de régression en Python[9]

1.6 Modélisation bayésienne

Dans le cadre de la théorie bayésienne de la décision, on considère les données \mathbf{x}, y comme des variables aléatoires. On va alors les modéliser par des lois de probabilités:

- $P(X = \mathbf{x})$, $P(Y = y)$: lois a priori (ou marginales)

- $P(X = \mathbf{x}, Y = y)$: loi jointe

- $P(X = \mathbf{x}|Y = y)$: vraisemblance conditionnelle

- $P(Y = y|X = \mathbf{x})$: loi a posteriori

À partir de maintenant on écrira $P(\mathbf{x})$ à la place de $P(X = \mathbf{x})$ et il en sera de même pour les autres variables aléatoires, le but étant de simplifier l'écriture.

Dans le cadre de la classification on a la prédition $y \in \{1, 2, \dots, N\}$ et est appelée étiquette. Dans cette section on se placera en classification binaire, ainsi il n'y aura que deux classes possibles : "1" et "2". On cherche à prédire une unique étiquette y^* à partir de \mathbf{x} :

$$\mathbf{x} \mapsto y^*$$

Pour prédire la classe de l'observation y , on va comparer $P(Y = 1|\mathbf{x})$ et $P(Y = 2|\mathbf{x})$. Ainsi on définit la règle suivante :

$$y^* = \begin{cases} 1 & \text{si } P(Y = 1|\mathbf{x}) > P(Y = 2|\mathbf{x}) \\ 2 & \text{sinon} \end{cases}$$

Dans le cas multiclass la règle est la suivante :

$$y^* = \arg \max_y P(y|\mathbf{x})$$

Cette approche du problème pose deux questions fondamentales qui trouvent leurs réponse grâce aux deux phases de l'apprentissage automatique. Celle d'apprentissage permet le calcul de $P(y|\mathbf{x})$ tandis que la phase de prédiction permet de trouver le maximum de $P(y|\mathbf{x})$ et donc l'étiquette y^* correspondante.

Pour simplifier le problème on peut utiliser la loi de Bayes:

$$P(y|\mathbf{x}) = \frac{P(\mathbf{x}|y)P(y)}{P(\mathbf{x})}$$

En règle générale on connaît la fréquence d'occurrence des classes y ce qui nous permet de connaître $P(y)$. De plus il est plus facile de calculer la vraisemblance $P(\mathbf{x}|y)$ que la loi a posteriori $P(y|\mathbf{x})$: "si je sais dans quelle classe je suis, je sais décrire le comportement/la distribution de mes données". De la loi de Bayes il découle que le maximum sur y ne dépend que de $P(\mathbf{x}|y)$ et de $P(y)$. Ainsi on recherchera le maximum sous la forme:

$$y^* = \arg \max_y P(\mathbf{x}|y)P(y)$$

1.6.1 Approche bayésienne multivariée:

Ici le calcul de la loi conditionnelle se fait selon un modèle multivarié. Supposons qu'on ait un ensemble de données d'entraînement $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$, on suppose qu'il s'agit d'un problème de classification binaire donc $\forall i \in [|1, n|]^1$, $y_i \in \{1, 2\}$. On considère que $\mathbf{x}_i \in \mathbb{R}^d$ avec $d \in \mathbb{N}$ et donc $\mathbf{x}_i = [x_{i1}, x_{i2}, \dots, x_{id}]$. Supposons que les vraisemblances des classes sont égales à (modèle gaussien):

$$P(X = \mathbf{x}_i | Y = 1; \boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1) = \frac{1}{(2\pi)^{d/2} \sqrt{|\boldsymbol{\Sigma}_1|}} \exp \left[-\frac{1}{2} (\mathbf{x}_i - \boldsymbol{\mu}_1)' \boldsymbol{\Sigma}_1^{-1} (\mathbf{x}_i - \boldsymbol{\mu}_1) \right]$$

$$P(X = \mathbf{x}_i | Y = 2; \boldsymbol{\mu}_2, \boldsymbol{\Sigma}_2) = \frac{1}{(2\pi)^{d/2} \sqrt{|\boldsymbol{\Sigma}_2|}} \exp \left[-\frac{1}{2} (\mathbf{x}_i - \boldsymbol{\mu}_2)' \boldsymbol{\Sigma}_2^{-1} (\mathbf{x}_i - \boldsymbol{\mu}_2) \right]$$

Dans le cas où $P(Y = 1) = P(Y = 2) = 0.5$ on a la règle de classification suivante :

$$y^* = \begin{cases} 1 & \text{si } P(X = \mathbf{x}_i | Y = 1; \boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1) > P(X = \mathbf{x}_i | Y = 2; \boldsymbol{\mu}_2, \boldsymbol{\Sigma}_2) \\ 2 & \text{sinon} \end{cases}$$

$$y^* = \begin{cases} 1 & \text{si } \log(P(X = \mathbf{x}_i | Y = 1; \boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1)) > \log(P(X = \mathbf{x}_i | Y = 2; \boldsymbol{\mu}_2, \boldsymbol{\Sigma}_2)) \\ 2 & \text{sinon} \end{cases}$$

$$y^* = \begin{cases} 1 & \text{si } (\mathbf{x}_i - \boldsymbol{\mu}_1)' \boldsymbol{\Sigma}_1^{-1} (\mathbf{x}_i - \boldsymbol{\mu}_1) - (\mathbf{x}_i - \boldsymbol{\mu}_2)' \boldsymbol{\Sigma}_2^{-1} (\mathbf{x}_i - \boldsymbol{\mu}_2) < \log\left(\frac{|\boldsymbol{\Sigma}_2|}{|\boldsymbol{\Sigma}_1|}\right) \\ 2 & \text{sinon} \end{cases}$$

Cette approche permet de décrire les corrélations entre les dimensions. Toutefois elle demande de connaître la forme des distributions et se limite malheureusement aux petites dimensions. Dans le cas d'un modélisation gaussienne à deux classes, la prédiction se réduit alors à calculer une fonction de degré 2 et la classe est prédite en fonction de sa position par rapport à la séparatrice de l'espace:

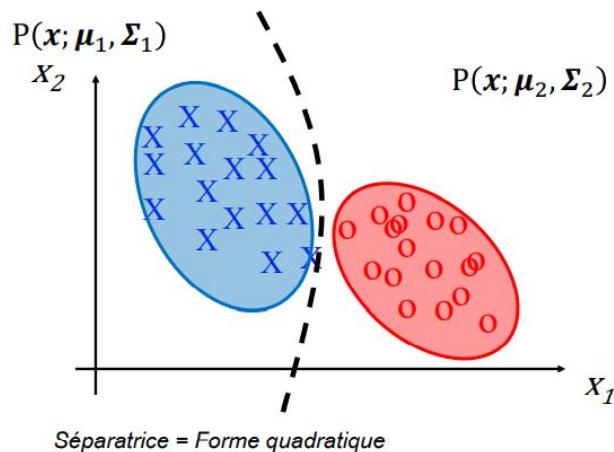


Figure 1.14: Exemple d'approche gaussienne multivariée

¹Ensemble des entiers naturels de 1 à n.

1.6.2 Approche bayésienne naïve

On utilise une hypothèse d'indépendance pour faire le calcul de la loi conditionnelle:

$$\begin{aligned} P(x_1, x_2, \dots, x_d|y) &= P(x_1|x_2, \dots, x_d, y)P(x_2, \dots, x_d|y) \\ &= P(x_1|y)P(x_2, \dots, x_d|y) \\ &= P(x_1|y)P(x_2|y)\dots P(x_d|y) \end{aligned}$$

On peut alors calculer la vraisemblance dimension par dimension ce qui est beaucoup plus simple. Le problème devient un ensemble de problèmes 1D, qui sont des modèles faciles à estimer (gaussien, binomial, histogrammes...).

Cette approche permet de traiter des problèmes de plus grande dimension grâce à ce découpage en sous-problèmes.

En pratique on utilise plutôt la log-vraisemblance afin de conserver une meilleure stabilité numérique:

$$\log P(\mathbf{x}|y) = \sum_i \log P(x_i|y)$$

$$y^* = \arg \max_y \log P(\mathbf{x}|y) + \log P(y)$$

Cette approche naïve peut se révéler comme étant très efficace malgré l'hypothèse d'indépendance assez forte. Comparons-la à l'approche multivariée sur un exemple:

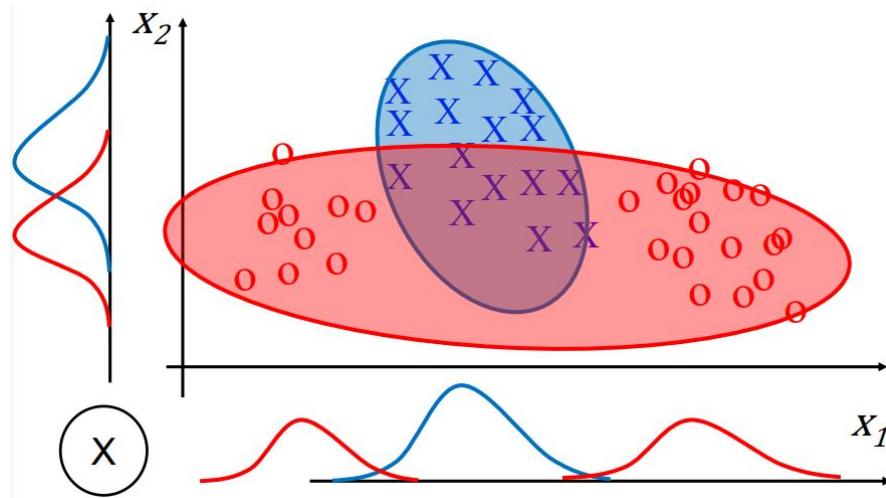


Figure 1.15: Comparaison entre l'approche bayésienne multivariée (schématisée par les ensembles colorés) et naïve (schématisée par les courbes à côté des axes)

On voit que l'approche multivariée désigne deux zones de l'espace qui ont une grande intersection, ce qui pourrait prêter à confusion, là où l'approche naïve (représentée par les courbes près des axes) délimite bien la zone de chaque classe. Cette approche dite naïve n'est donc pas à négliger malgré sa dénomination péjorative.

1.6.3 Résumé concernant l'approche bayésienne

L'approche bayésienne est une théorie probabiliste de la décision qui nécessite le calcul de la loi a posteriori afin de déterminer une prédition.

Pour trouver l'expression de cette loi a posteriori on peut poser une hypothèse d'indépendance conditionnelle pour se ramener en dimension 1 ou alors utiliser un modèle gaussien multivarié.

La phase d'apprentissage consiste alors en une estimation de lois paramétriques simples. Pendant la phase de prédition le programme va effectuer le calcul de la log-vraisemblance et la maximiser sur les différentes hypothèses.

Cependant cette approche se limite aux petits problèmes bien modélisés dans le cas du modèle gaussien multivarié et aux caractéristiques non corrélées pour une approche bayésienne naïve (il est toutefois possible que ce modèle fonctionne dans des cas particuliers où les caractéristiques sont corrélées).

1.6.4 L'approche bayésienne en Python

Dans cette partie nous allons voir comment générer un exemple d'échantillon qui peut être étudié par une approche bayésienne naïve à l'aide de la bibliothèque `scikit-learn`. On va plus particulièrement utiliser la fonction `make_blobs` qui permet de générer un échantillon gaussien isotrope :

```
from sklearn.datasets import make_blobs
X, y = make_blobs(100, 2, centers=2, random_state=2, cluster_std=1.5)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='RdBu');
```

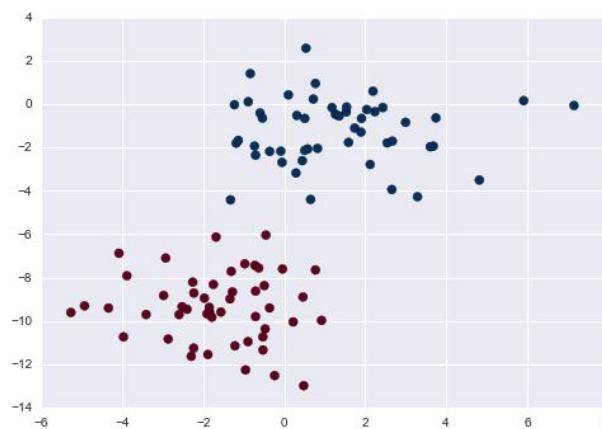


Figure 1.16: Exemple de génération par la fonction `make_blobs` [9]

Le modèle bayésien naïf gaussien est inclus dans la bibliothèque sous le nom `GaussianNB`, on lui donne les données générées précédemment pour son apprentissage:

```
from sklearn.naive_bayes import GaussianNB
model = GaussianNB()
model.fit(X, y);
```

Figure 1.17: Définition du modèle bayésien naïf gaussien[9]

Après ça il ne reste plus qu'à générer d'autres données pour déterminer approximativement où se trouve la surface de séparation de décision:

```
rng = np.random.RandomState(0)
Xnew = [-6, -14] + [14, 18] * rng.rand(2000, 2)
ynew = model.predict(Xnew)
```

```
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='RdBu')
lim = plt.axis()
plt.scatter(Xnew[:, 0], Xnew[:, 1], c=ynew, s=20, cmap='RdBu', alpha=0.1)
plt.axis(lim);
```

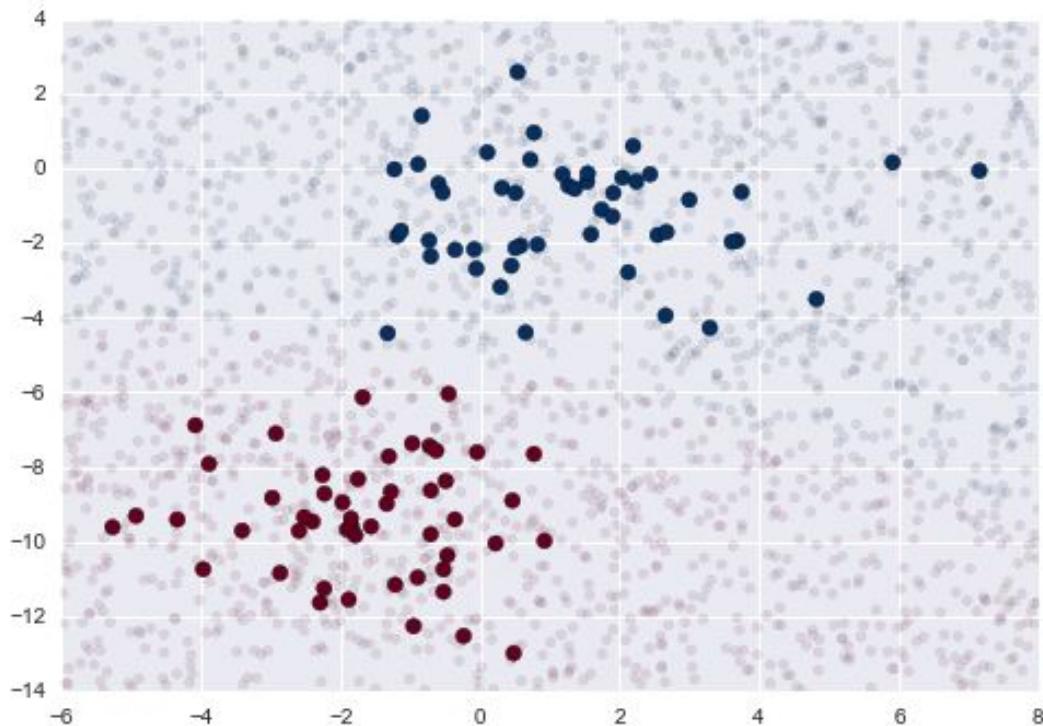


Figure 1.18: Exemple d'utilisation de l'approche bayésienne naïve en Python[9]

1.7 La classification Plus Proche Voisin (PPV)

1.7.1 Principe général de l'approche 1-PPV

On se place dans un cas dans lequel on dispose de deux échantillons proches dans l'espace de représentation et dont les prédictions sont identiques au sein de chacun de ces échantillons.

Pour prédire la classe d'une nouvelle donnée on va alors chercher l'exemple annoté le plus proche et lui associer son annotation:

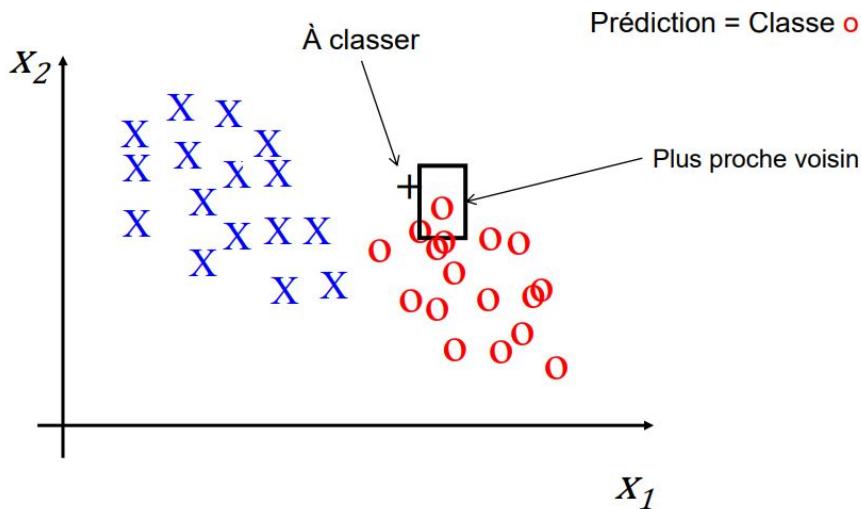


Figure 1.19: Exemple illustrant le principe de fonctionnement de l'approche plus proche voisin

Cette approche repose sur une notion de proximité entre les données qui nécessite la définition d'une métrique, ou mesure de similarité $d(\mathbf{x}, \mathbf{x}')$. Il existe plusieurs métriques qui peuvent être utilisées dans le cadre d'un algorithme qui suit le principe de classification PPV. On peut citer les distances euclidienne, city-block (dite "de Manhattan"), Minkowski et Mahalanobis.

Il est également envisageable d'apprendre la métrique ou mesure de similarité la plus pertinente pour un problème en particulier.

Il faut maintenant définir formellement ce que l'on entend par "plus proche" voisin:

On dispose d'une base d'échantillons annotés $\mathcal{L} = \{(x_1, y_1), (x_2, y_2), \dots (x_n, y_n)\}$

La recherche de l'échantillon le plus proche de notre nouvelle donnée \mathbf{x} se résume à la recherche de i^* tel que $i^* = \arg \min_{i^*} d(\mathbf{x}, \mathbf{x}_i)$

Une fois ce plus proche voisin déterminé on assigne son annotation à la prédiction de la nouvelle donnée: $y^* = y_{i^*}$

On voit que, certes la fonction d'approximation n'est pas continue, mais elle a l'avantage de passer par tous les points de notre base de données initiale sans avoir de variations excessives comme c'était le cas dans le modèle linéaire utilisé en 1.5.1 qui atteignait un état de sur-apprentissage.

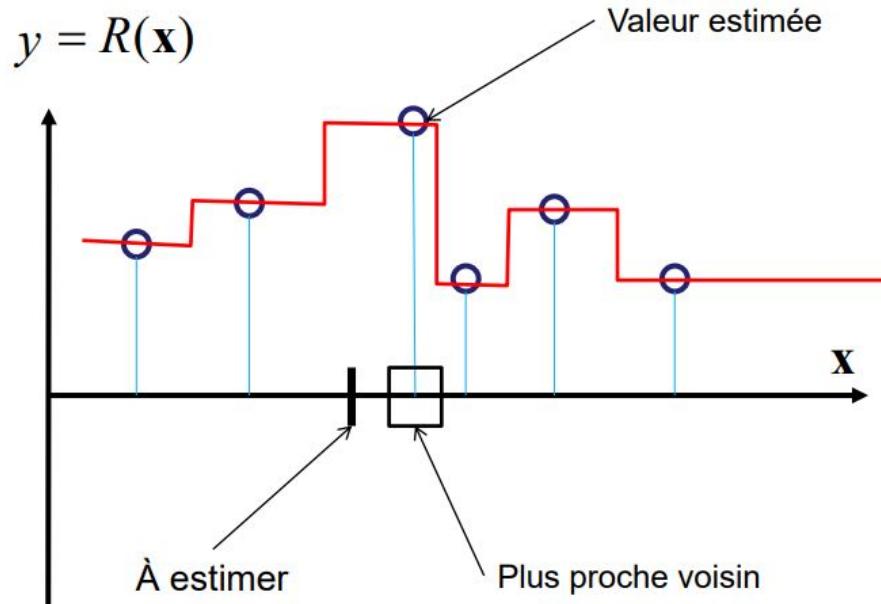


Figure 1.20: Exemple d'application du schéma plus proche voisin en régression

Cependant ce modèle n'est pas exempt de tout défaut notamment sur des fonctions de classifications :

Données bruitées → Régions isolées → mauvaise régularité des prédictions

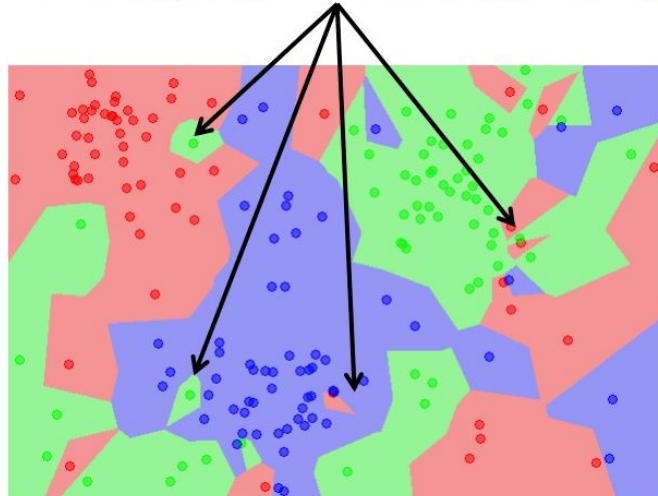


Figure 1.21: Exemple d'application du schéma plus proche voisin en deux dimensions [10]

On voit sur cet exemple que des données initiales bruitées conduisent à des prédictions irrégulières. En effet il plus probable que, sur cet exemple, le seul point vert en haut à gauche soit dû au bruit et ne devrait pas avoir autant d'influence sur la zone. Pour remédier à ce problème on va pousser un peu plus le principe PPV.

1.7.2 L'approche des k-plus proches voisins (dite "k-NN")

Pour approfondir le principe PPV de base, on va maintenant utiliser plusieurs exemples de la base de données d'apprentissage pour déterminer une prédiction pour la nouvelle donnée. On commence donc par ordonner les échantillons d'apprentissage en fonction de leur distance à la donnée à classer:

$$d(\mathbf{x}, \mathbf{x}_1) \leq d(\mathbf{x}, \mathbf{x}_2) \leq \dots \leq d(\mathbf{x}, \mathbf{x}_n)$$

Cette ordonnancement va nous permettre de choisir les k exemples de la base les plus proches de la nouvelle donnée. Après cela on va prédire la classe de cette donnée en choisissant celle qui recueille le plus de "votes" parmi ces k plus proches voisins:

$$y^* = \arg \max_y \sum_{i=1}^k \delta(y, y_i)$$

Où δ est la fonction de Kronecker, qui vaut 1 si ses deux arguments sont égaux et qui vaut 0 sinon.
Attention: s'il n'y a pas de maximum, cela signifie qu'il y a ambiguïté sur la prédiction. Dans ce cas on ne décide pas et on laisse la donnée nouvelle sans annotation!
Il est également possible de pondérer les votes, pour donner plus de poids aux voisins les plus proches par exemple. Dans ce cas on écrit:

$$y^* = \arg \max_y \sum_{i=1}^k K(\mathbf{x}, \mathbf{x}_i) \delta(y, y_i)$$

Observons les effets de cette nouvelle approche sur l'exemple de tout à l'heure en utilisant les cinq voisins les plus proches de chaque point de l'espace:

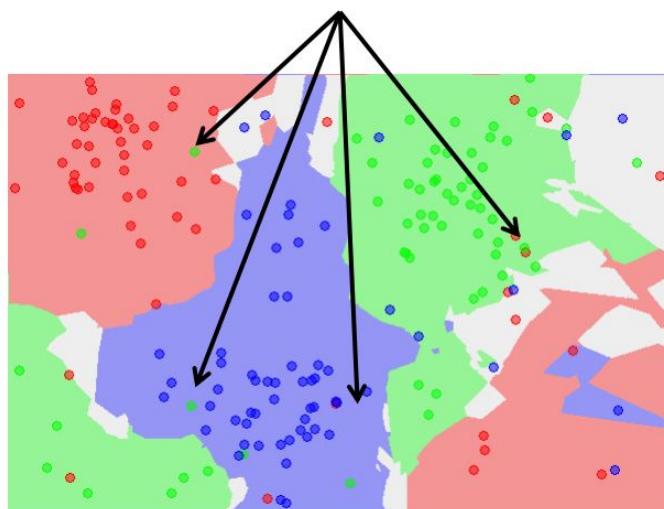


Figure 1.22: Reprise de l'exemple présenté en Figure 1.14 en prenant cette fois-ci en compte les 5 plus proches voisins [10]

On voit que les points qui semblaient peut cohérents, sans doute à cause du bruit, n'ont désorais plus d'influence ce qui donne des régions plus homogènes au sein de l'espace de représentation.

Cette approche des k-plus proches voisins dispose de la propriété statistique suivante:

$$E_{Bayes} \leq \lim_{N \rightarrow \infty} E_{kNN} \leq E_{Bayes} \left(2 - \frac{L E_{Bayes}}{L - 1} \right)$$

Où E_{Bayes} est l'erreur théorique optimale (dite de Bayes)¹, L est le nombre de classes et E_{kNN} est l'erreur des k-plus proches voisins. Il en découle entre autre que l'erreur de l'approche k-NN est au plus deux fois moins bonne que l'erreur minimale théorique.

1.7.3 Le coût le la prédiction k-ppv

Si la prédiction du k-plus proches voisins peut sembler efficace, elle a un coût non négligeable puisqu'elle nécessite un calcul pour chaque exemple \mathbf{x} ainsi qu'un tri par rapport aux n exemples de la base.

Pour n et d (la dimension des données) grands, le coût de la recherche exhaustive est important puisque ce dernier croît en $O(nd)$.

On peut néanmoins aborder le problème d'une autre manière en pré-calculant les surfaces de séparation entre les classes. La complexité de prédiction est alors liée à la complexité de la surface et/ou de son approximation.

1.7.4 La malédiction des grandes dimensions

Lorsque la dimension d de l'espace de représentation des données augmente, les points de cet espace sont tous situés, plus ou moins, à la même distance les uns des autres. Les données étant de plus en plus dispersées, elles apportent moins d'information pour la méthode des plus proches voisins.

Nonobstant cette propriété ne s'applique heureusement pas aux distributions structurées. On peut interpréter les techniques de Machine Learning comme des moyens de détection des bonnes corrélations entre les données.

Toutefois la propriété précédente limite les approches plus proches voisins aux faibles dimensions, à moins qu'il ne soit possible de réduire le nombre de dimensions de représentation des données avant de calculer les distances comme en apprentissage non supervisé.

1.7.5 Le comportement des approches PPV

Le schéma des plus proches voisins dispose de plusieurs avantages intéressants, notamment sa flexibilité, sa facilité de mise en œuvre, le choix de la définition d'une similarité entre les données ainsi que ses bonnes propriétés statistiques asymptotiques lorsque $n \rightarrow \infty$.

Toutefois son temps de calcul l'exclue totalement des problèmes aux bases de données conséquentes et la malédiction des grandes dimensions l'empêche de fonctionner pour des données aux dimensions trop élevées. Enfin la régularité de l'algorithme dans ce schéma dépend des données et non de l'apprentissage, même si le k-PPV permet de réduire l'effet du bruit sur les données.

¹Pour information $E_{Bayes} = E \left[\sum_{k=1}^K (1 - \delta(c_k, f(\mathbf{x})) P(c_k | \mathbf{x}) \right]$ où les c_k constituent l'ensemble des classes possibles et δ est la fonction de Kronecker.

1.7.6 Résumé concernant l'approche plus proches voisins

Pour pouvoir utiliser ce schéma, il faut d'abord pouvoir faire l'hypothèse que le problème étudié possède une certaine régularité. C'est-à-dire que des observations dont les caractéristiques sont proches auront un comportement similaire.

Après avoir vérifié que cette hypothèse est cohérente, il faut répondre à deux questions. Tout d'abord, il faut définir ce que "proche" signifie dans le problème étudié et introduire une métrique en accord avec cette définition. Il faut ensuite déterminer quelles données de la base sont les plus proches de la donnée à classer. L'approche des plus proches voisins n'a pas de phase d'apprentissage à proprement parler puisqu'elle se contente d'utiliser les données de la base. En contrepartie la phase de prédiction est plus onéreuse car elle nécessite de trier les distances aux échantillons de la nouvelle donnée avant de procéder à un vote entre les k plus proches.

L'utilisation de ce schéma est pertinente sur des problèmes d'une taille modeste aussi bien en dimensions des données que du nombre d'exemples de la base de données. Autrement on risque de se heurter à des problèmes de temps de calcul et à la malédiction des grandes dimensions qui sont difficiles à résoudre. Il faut également s'assurer de disposer d'une mesure de similarité adaptée aux données du problème pour obtenir une classification optimale.

1.8 L'essentiel à retenir du chapitre

Nous avons commencé ce chapitre avec les principes de base de l'apprentissage automatique, et notamment que son objectif était de programmer à partir des données avec une démarche qui se décompose en deux phases que sont l'apprentissage et la prédiction et que chacune de ces phases peut être de plusieurs variétés.

Nous avons ensuite parcouru la démarche générique du traitement d'un problème à travers l'exemple des chiffres manuscrits à reconnaître. Cette démarche est en cinq étapes qui ont chacune leur importance dans le développement du programme:

1. la constitution d'une base d'apprentissage
2. l'analyse préliminaire des données ainsi que leur préparation
3. la conception du modèle
4. l'optimisation
5. l'évaluation de l'efficacité du prédicteur construit

La minimisation de l'erreur de généralisation est l'objectif principal du Machine Learning et pour atteindre ce but on divise la base de données initiale en deux échantillons distincts. Le premier est utilisé pour l'apprentissage de l'algorithme puis on évalue l'efficacité de cet apprentissage sur l'échantillon test. Il ne faut surtout pas utiliser l'échantillon test pour la phase d'entraînement de l'algorithme sous peine de ne plus pouvoir effectuer de vérification sans obtenir de nouvelles données.

Enfin nous avons terminé avec la présentation de deux approches élémentaires que sont la modélisation bayésienne, qui traite x et y comme des variables aléatoires en calculant la loi a posteriori de y , et l'approche des plus proches voisins, qui annote les nouvelles données en fonction des échantillons les plus proches de la base de données sous hypothèse de régularité.

Chapter 2

Arbres de décision et méthodes ensemblistes

2.1 Les arbres de décision

2.1.1 Des graphes aux arbres

Definition 2.1 (Graphe). Un graphe que l'on note $G = (V, E)$ est constitué d'un ensemble non vide V de sommets (ou noeuds) et d'un ensemble E d'arêtes qui relient les sommets. Chaque arête a soit un ou deux sommets qui lui sont associés, appelés ses points d'extrémité.

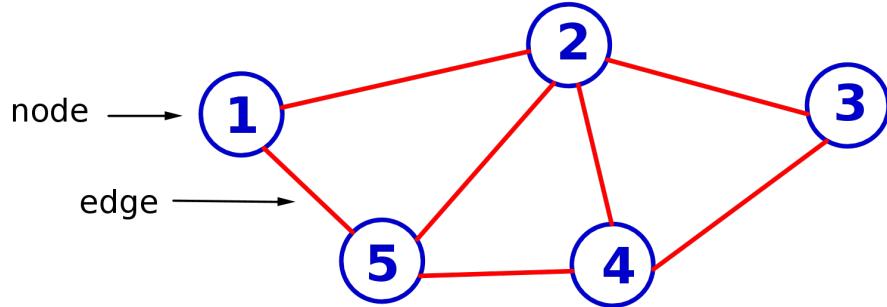


Figure 2.1: Exemple d'un graphe avec 5 sommets et 7 arrêtes.

Dans la Figure 2.1 on a représenté un graphe simple.

Definition 2.2 (Arbre). Un arbre est un graphe non orienté dans lequel deux sommets quelconques sont connectés par exactement un chemin, ou de manière équivalente un graphe non orienté acyclique connecté.

Definition 2.3 (Forêt). Une forêt est un graphe non orienté dans lequel deux sommets quelconques sont connectés par au plus un chemin.

Definition 2.4 (Arbre enraciné). Un arbre enraciné est un arbre dans lequel un sommet a été désigné comme étant la racine et chaque arête est dirigée à partir de la racine

2.1.2 Présentation de la structure d'arbre

Avant d'entrer dans la partie Machine Learning de cette section, il est nécessaire de présenter ce que l'on entend par "arbre de décision" dans le domaine de l'informatique en général. Il s'agit d'une structure de

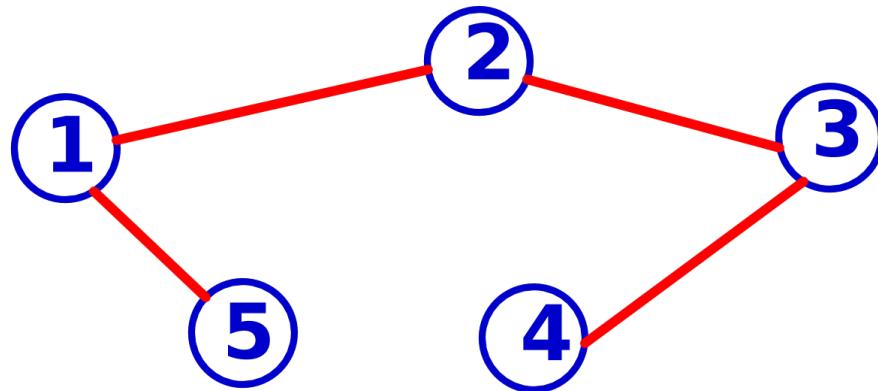


Figure 2.2: Exemple d'un arbre avec 5 sommets et 4 arrêtes.

données qui se peut se décrire de deux façons. On peut tout d'abord adopter un point de vue récursif et dire qu'un arbre est soit vide, soit il s'agit d'un nœud racine et de sous-arbres étant eux-même des arbres. Sinon on peut dire qu'un arbre est composé de nœuds qui possèdent chacun un unique nœud parent (ou nœud père) à l'exception de la racine qui n'en a pas. A contrario un nœud peut avoir plusieurs enfants.

On peut définir d'autres éléments d'un arbre :

- Une **feuille** est un nœud qui n'a pas d'enfant.
- Une **branche** est un chemin allant de la racine à une feuille.
- Une **arête** est un segment qui relie deux nœuds.

L'avantage des arbres, au-delà de leur structure permettant facilement la mise en place d'algorithmes récursifs, réside dans leur représentation intuitive et visuelle :

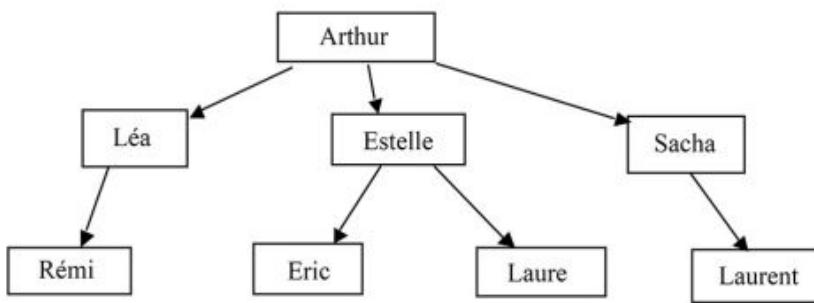


Figure 2.3: Exemple d'arbre généalogique (les flèches signifient "a pour enfant") [11]

2.1.3 Principe général

Le modèle des arbres de décision vise à décomposer une décision globale en une succession de décisions locales plus faciles à traiter. On représente cette décomposition par un arbre dans lequel chaque noeud correspond à une question simple, une décision locale. En fonction de la réponse à cette question on va continuer avec un des noeud fils si d'autres informations sont nécessaires ou on arrivera sur une feuille de l'arbre qui donnera une réponse à la question globale.

Les questions posées au sein d'un tel arbre doivent être fermées, c'est-à-dire que le nombre de réponses possibles doit être fini. Dans le cas où une question nécessite une réponse numérique, comme l'estimation d'une durée par exemple, il faut décomposer l'espace des réponses en un nombre fini de sous-espaces pour pouvoir décrire la suite de l'arbre.

2.1.4 Un premier exemple

Pour se familiariser avec les arbres de décisions, étudions un premier exemple qui traite d'une situation connue de tous : faut-il attendre qu'une table se libère dans un restaurant plein ?

Il faut commencer par déterminer quels sont les facteurs importants à prendre en compte lors de cette décision et les formuler en questions les plus simples possibles : Est-ce que nous avons vraiment faim ? Est-ce qu'il y a d'autres restaurant à proximité ? Quel est le temps d'attente estimé ? Est-ce qu'il pleut ? Est-ce qu'il y a un bar dans le restaurant pour pouvoir patienter ?...

On classe alors ces questions par ordre de crucialité puis on construit l'arbre de décision en plaçant la question la plus importante à la racine (ici ce serait "le restaurant accueille-t-il encore des clients ?" par exemple) puis, en fonction de la réponse, soit on a déjà suffisamment d'informations pour prendre la décision globale (si la réponse est "non" alors il faut changer de restaurant), soit d'autres questions se posent pour pouvoir trancher et dans ce cas on pose la question restante la plus importante et on itère ce procédé jusqu'à ce que tous les cas de figure aient été couverts par l'arbre.

Pour le problème du restaurant on peut par exemple parvenir à un arbre de ce genre :

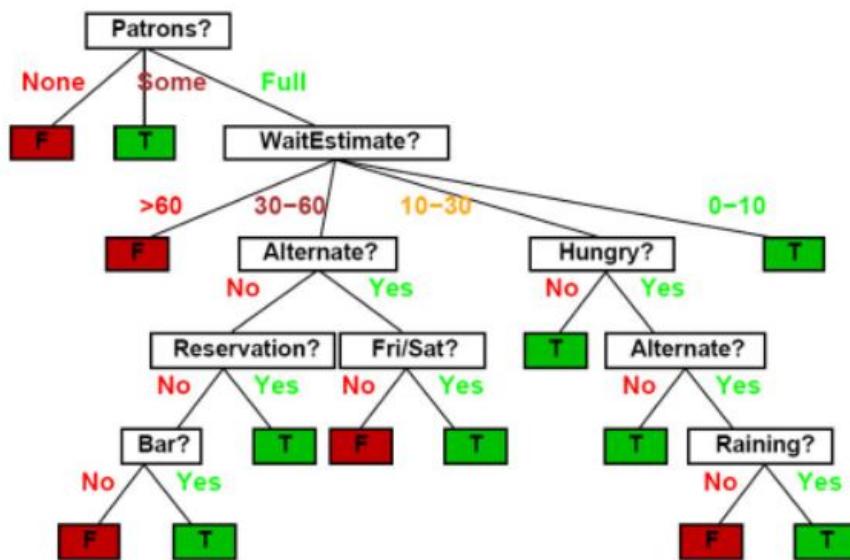


Figure 2.4: Exemple d'arbre de décision pour le problème du restaurant [12]

2.1.5 Les différentes questions possibles

Les questions utilisées dans la création d'arbres de décision peuvent être de natures très différentes. Cette caractéristique donne une très grande flexibilité à ce modèle. Néanmoins il faut bien garder à l'esprit que ces questions doivent rester très simples à traiter pour le programme afin de pouvoir lui assurer une certaine efficacité. Les questions peuvent porter sur la valeur d'un attribut caractéristique, sur la véracité d'une clause logique, sur l'appartenance à un intervalle ou un sous-ensemble, sur des attributs discrets ou numériques... Les possibilités de questions sont infinies et il faut trouver un équilibre entre le nombre d'informations dont on souhaite disposer et le temps de calcul que nécessitera la prédiction globale.

2.1.6 Formalisation de la structure d'un arbre de décision

Maintenant que nous avons une vision un peu plus claire et pratique de ce qu'est un arbre de décision, voyons comment nous pouvons le formaliser dans le cadre d'une application à l'apprentissage automatique : Les données seront codées comme étant un ensemble d'attributs permettant de répondre facilement aux différentes questions qui auront été retenues lors de la création de l'arbre.

Un nœud de décision sera associé à un test, d'appartenance ou de véracité selon les cas, ou à une question qui portera toujours sur un et un seul attribut des données pour que la réponse soit facile d'accès et donc la plus rapide possible.

Les branches de l'arbre représenteront les différentes valeurs possibles de l'attribut testé ou des réponses possibles à la question posées par le nœud de décision père.

Enfin un nœud terminal, que l'on pourra aussi appeler feuille, désignera la classe qui servira de prédiction pour la donnée entrée dans l'algorithme.

2.1.7 Apprentissage hiérarchique

Nous n'avons vu jusqu'à maintenant que des techniques de classification ne nécessitant qu'une seule opération pour étiqueter une certaine observation x . Ces modèles utilisent le même ensemble de variables pour l'ensemble des classes en accordant la même importance à chacune de ces variables. Ainsi ils ne sont pas adaptés aux situations où les classes sont déterminées par différentes variables dans différentes régions de l'espace. Par exemple une pomme peut être décrite comme rouge et grande ou comme jaune et ronde, c'est ce que l'on appelle une classe **multi-modale**.

Les arbres de décision nécessitent, pour leur part, de multiples tests conditionnels qui vont permettre de classifier les données, chacun de ces tests dépendant de ses antécédents. Cette série de tests place les arbres de décision dans la catégorie des *modèles hiérarchiques*. On en arrive alors à la définition suivante d'un arbre de décision :

On appelle **arbre de décision** un modèle de prédiction qui peut être représenté sous la forme d'un arbre. Chaque noeud de l'arbre teste une condition sur une variable et chacun de ses enfants correspond à une réponse possible à cette condition. Les feuilles de l'arbre correspondent à une étiquette. Pour prédire l'étiquette d'une observation, on "suit" les réponses aux tests depuis la racine de l'arbre, et on retourne l'étiquette de la feuille à laquelle on arrive.[13]

Position du problème d'apprentissage

Soit $\mathcal{D} = \{(x_i, y_i)\}_{i \leq n}$ l'ensemble des données décrites chacunes par un ensemble d'attributs $x_i = [A_i^1, A_i^2, \dots, A_i^M]$, avec $\{A_i^j\}_{j \in [|1, M|]}$ à valeurs numériques ou symboliques, M le nombre d'attributs d'une donnée et y_i la prédiction associée, qui est ici connue puisque \mathcal{D} est notre base de données initiale.

À partir de cette base de données on va donc chercher à déterminer le plus petit arbre de décision compatible avec notre problème. Malheureusement, la recherche d'un tel arbre est impossible puisqu'il s'agit d'un problème NP-complet à l'instar du problème du voyageur de commerce par exemple. Dans un tel problème il est possible de vérifier qu'une solution proposée est optimale, cependant on ne sait pas en trouver une efficacement. On peut seulement se conformer à une heuristique assurant la création d'un arbre ayant une certaine cohérence avec les données.

Dans ce but on peut utiliser des principes comme celui du rasoir d'Occam qui est un principe de raisonnement philosophique selon lequel il faut "éliminer des explications improbables d'un phénomène". Aujourd'hui on l'utilise plus sous l'énoncé suivant : "les hypothèses suffisantes les plus simples doivent être préférées", ce qui revient pour notre problème à trouver l'hypothèse la plus simple possible compatible avec les données. Un autre principe important est celui du *Minimum Description Length* basé sur le fait que toute régularité dans un ensemble de données peut être utilisée afin de compresser l'information. Ce principe consiste en la recherche de l'hypothèse qui produit le plus petit nombre d'opérations grâce à la compression des données.

2.1.8 Algorithme de construction d'un arbre de décisions

Dans cette partie nous allons voir comment construire un arbre de décision grâce à un algorithme élémentaire. On se place dans l'optique de construire l'arbre de façon incrémentale dont chaque étape sera décomposée en deux phases. Tout d'abord on commence par décider si le nœud traité lors de l'étape est un nœud terminal, si c'est le cas alors on lui associe une prédiction et, dans le cas contraire, on choisit alors un attribut parmi ceux des données, un test à effectuer sur l'attribut sélectionné ainsi que les branches pouvant être empruntées après ce test.

La structure d'arbre encourage fortement l'utilisation d'un algorithme récursif. En effet un arbre se décompose toujours comme étant un nœud racine et les différents fils de ce nœud ne sont que des sous-arbres auxquels on peut appliquer la fonction de construction. On pourrait présenter une première version de l'algorithme avec le pseudo-code suivant :

Fonction Construire-arbre(\mathcal{D}):

SI tous les points de \mathcal{D} sont de même classe :

Créer une feuille associée à cette classe

SINON :

Choisir la meilleure paire (A^j, test) pour créer un nœud

Ce test sépare \mathcal{D} en q parties $\mathcal{D}_1, \dots, \mathcal{D}_q$

Construire-arbre(\mathcal{D}_k) pour $k \in \{1, \dots, p\}$

Ce pseudo-code correspond à l'algorithme appelé **Iterative Dichotomiser 3** (ou **ID3**) le plus basique possible. Toutefois cette première version n'est pas vraiment optimale puisqu'il faut attendre que toutes les données de l'échantillon \mathcal{D} aient la même classe pour pouvoir créer une feuille. Cette vision risquerait de conduire à une situation de sur-apprentissage puisque, comme on peut le voir sur l'exemple de la Figure 2.2 avec certains points associés à des classes différentes de tous leurs voisins les plus proches, les données de la base initiale peuvent être bruitées. Ainsi partitionner l'espace des attributs de telle sorte que chaque sous-espace ne contienne qu'une seule classe rendrait le programme trop proche des données et entraînerait un phénomène d'overfitting.

Pour palier à ce problème il existe plusieurs critères permettant de mesurer l'hétérogénéité (ou la pureté) d'une base de données, et donc de déterminer à quel moment il serait judicieux d'arrêter de découper l'espace des attributs. Pour pouvoir définir de tels critères nous allons commencer par introduire la notation $P_c(R)$ qui correspond à la proportion d'exemples d'entraînement de la région R qui appartiennent à la classe c :

$$P_c(R) = \frac{1}{|R|} \sum_{i/x_i \in R} \delta(y_i, c)$$

Un premier critère, utilisé pour mesurer l'information et que l'on intègre souvent à l'algorithme ID3, est l'**entropie** :

$$H(R) = - \sum_{c=1}^C P_c(R) \log_2(P_c(R))$$

On peut noter que ce critère est également utilisé régulièrement dans l'algorithme **C4.5**, qui découle de ID3 mais qui calcule à chaque étape et pour chaque attribut le ratio de gain d'information¹ et sélectionne l'attribut pour lequel il est maximal.

Dans une variante de l'algorithme ID3, l'algorithme **Classification And Regression Tree** (ou **CART**), on se restreint aux arbres binaires (un nœud ne peut avoir que deux fils) et on utilise le critère connu sous l'appellation d'indice de Gini, et qui sert à mesurer les inégalités entre les différentes classes :

$$I_G(R) = \sum_{c=1}^C P_c(R)(1 - P_c(R)) = 1 - \sum_{c=1}^C P_c(R)^2$$

¹La formule est détaillée sur la page suivante.

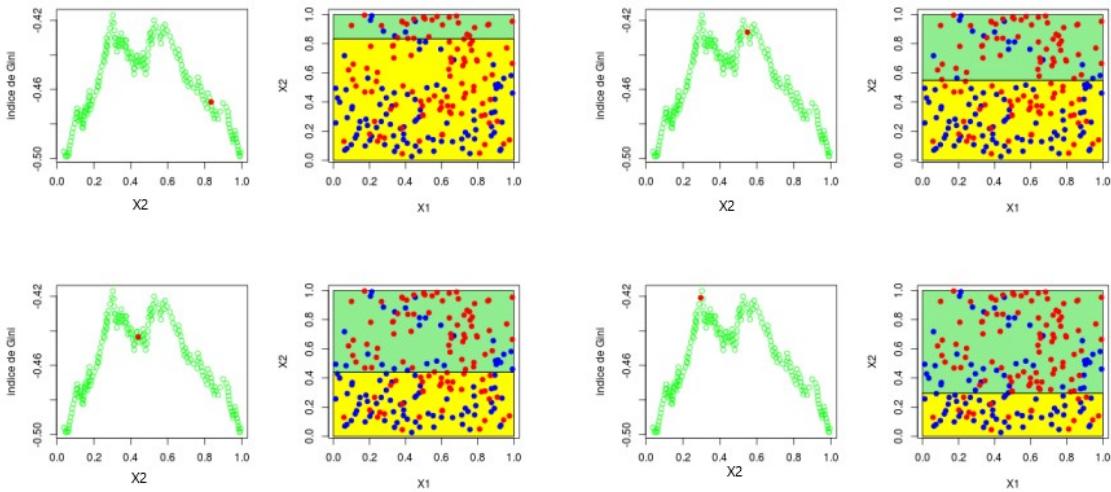


Figure 2.5: Exemple de partition de l'ensemble en fonction de différentes valeurs de l'indice de Gini

Enfin on peut citer l'indice d'erreur qui peut permettre de connaître la probabilité qu'une donnée n'appartienne pas à la classe prédominante de l'échantillon :

$$I_{err}(R) = 1 - \max_{c \in [|1, C|]} P_c(R)$$

Pour le choix du test, on cherche à maximiser le gain d'homogénéité apporté par un test T pour séparer une région R en sous-régions R_j :

$$Gain(R, T) = H(R) - \sum_j P(R_j)H(R_j)$$

Avec $P(R_j)$ la proportion d'exemples d'entraînement dans la région R qui sont dans R_j . Avec cette notion on obtient la formule du ratio de gain d'information évoqué précédemment :

$$IGR(R, T) = \frac{Gain(R, T)}{H(R)}$$

En pratique, on utilise une approche empirique pour tout les attributs A^j puis on effectue un tri des valeurs de gains par ordre croissant avant de tirer un test selon une approche dichotomique.

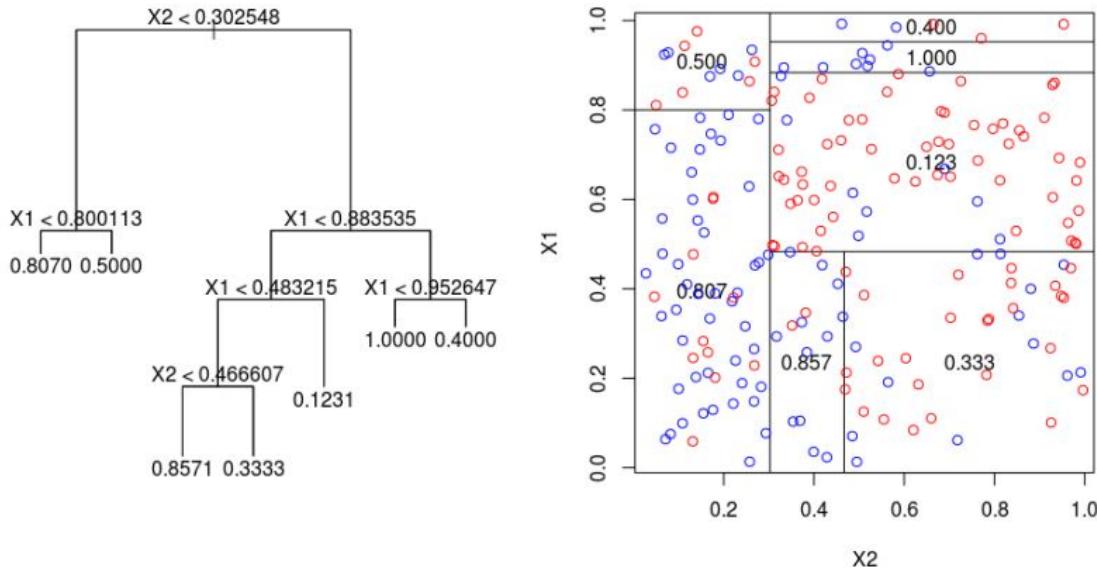


Figure 2.6: Exemple d'arbre pouvant être obtenu en utilisant un critère de type Gini[14]

Comme dit précédemment, il faut à tout prix essayer d'éviter le sur-apprentissage lors de la construction d'un l'arbre de décision puisque, non seulement il risquerait de mal généraliser le problème, mais en plus il risquerait d'être mal équilibré, ce qui augmenterait fortement le temps de traitement de nouvelles données. Si on atteint une telle situation il est recommandé d'utiliser des techniques d'élagage (ou pruning en anglais) pour améliorer la qualité de l'arbre a posteriori.

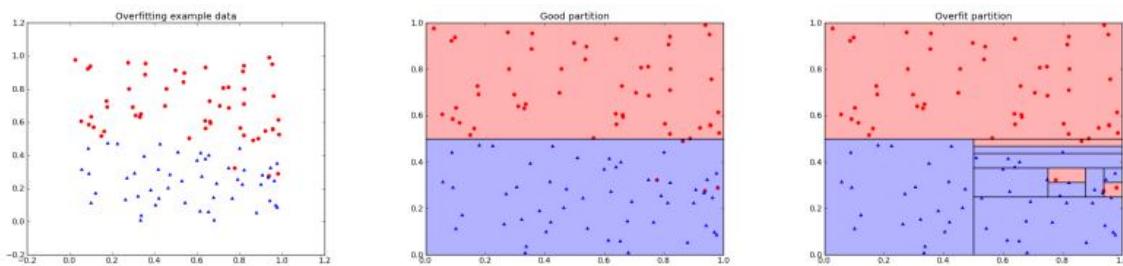


Figure 2.7: Exemple d'espace partitionné en suivant trop les données

Complexité de la construction d'un arbre

La construction d'un arbre de décision peut vite devenir un problème nécessitant un temps déraisonnable pour de multiples raisons (nombreuses classes possibles, base de données de grande taille, etc...). Cette complexité peut néanmoins être contrôlée grâce à plusieurs restrictions que l'on peut imposer : limitation de la profondeur de l'arbre, minoration du gain en homogénéité, ajout d'une pénalisation de complexité dans le coût ou encore en garantissant une bonne estimation des coûts en imposant un nombre minimal d'échantillons par nœud.

2.1.9 Résumé concernant les arbres de décision

Nous avons vu dans cette partie que les arbres de décision constituaient une approche extrêmement facile à interpréter d'un point de vue visuel et logique. En effet la représentation graphique d'un arbre permet de rester dans un cadre intuitif où il est facile de se retrouver. De plus, au-delà de la phase de construction d'un arbre de prédiction, cette méthode offre un apprentissage et une classification rapides et efficaces pour peu que l'on se restreigne à des tests simples. Cette affirmation reste vraie en grande dimension, contrairement aux méthodes abordées dans le premier chapitre.

Toutefois cette approche, comme toutes les autres, a ses défauts. Le plus gênant pour la méthode des arbres de décision est sa tendance au sur-apprentissage mais nous avons vu qu'il existe cependant des moyens de contrôler la complexité de l'arbre créé. De plus cette tendance provoque une certaine sensibilité au bruit et aux points aberrants que pourrait contenir la base de données initiale.

Enfin il faut retenir que les arbres de décision ne s'utilisent que dans le cadre de problèmes de classification, voire de régression, et qu'ils sont capables de traiter des données aussi bien numériques que symboliques.

2.2 Introduction des méthodes ensemblistes

Commençons tout d'abord par définir ce que l'on entend par "méthodes ensemblistes". Il s'agit de méthodes agrégeant des *ensembles* de classificateurs et dont l'objectif est de produire une variété de classificateurs en échantillonnant différemment les données, en modifiant les structures des classificateurs... Cette production va permettre de déterminer une classe finale en utilisant la fusion des prédictions données par les différents classificateurs.

Le principe est donc de tirer parti de plusieurs classificateurs peu performants (que l'on appelle aussi faibles) afin de construire une fusion de ces classificateurs qui sera performante (classificateur "fort"). Cette fusion va permettre de réduire la variance d'apprentissage et de moyenner les erreurs des classificateurs. En d'autre terme on adopte la philosophie de *l'union fait la force!*

Il existe deux grandes approches pour ce type de méthode : le bagging, qui développe plusieurs modèles en parallèle, et le boosting, qui est une amélioration séquentielle du modèle.

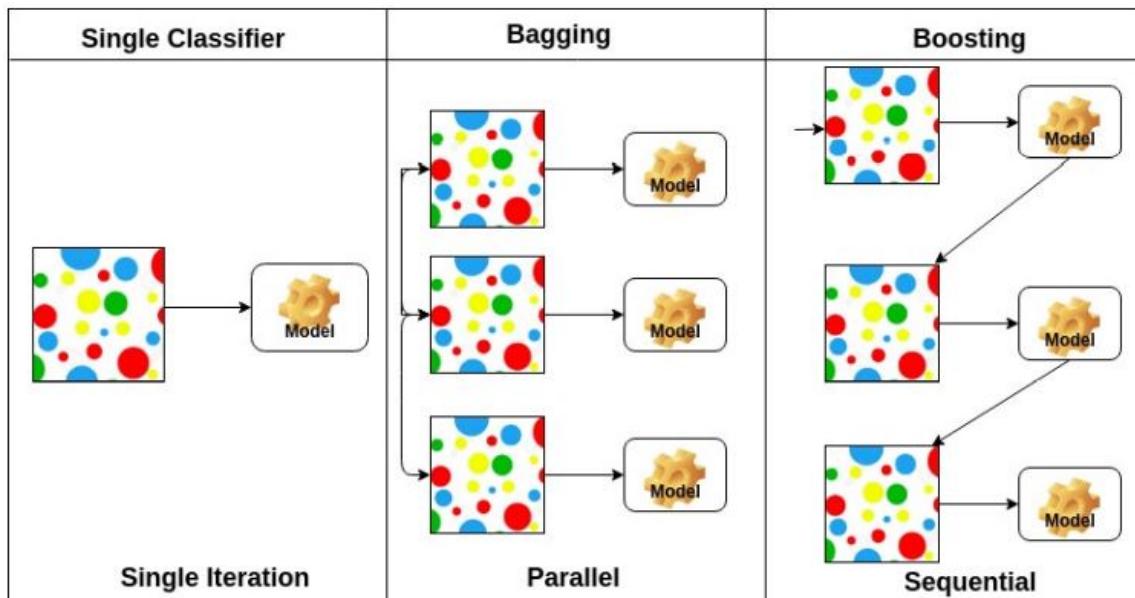


Figure 2.8: Schéma représentant les deux approches des méthodes ensemblistes [15]

2.3 Bagging

La technique du bagging consiste en la génération de multiples jeux de données à partir de la base initiale puis d'entraîner le programme sur chacun des jeux créés. Ici on ne se focalise pas sur le contenu de l'algorithme en lui-même mais sur la méthode d'entraînement de celui-ci. Ainsi le bagging peut s'appliquer à n'importe quel modèle d'apprentissage automatique : un arbre de décision, un réseau de neurone, une SVM, etc...

On commence donc par construire des jeux de données $\tilde{\mathcal{D}}_1, \tilde{\mathcal{D}}_2, \dots, \tilde{\mathcal{D}}_K$ en effectuant un tirage avec remise dans la base de données initiale \mathcal{D} . Cette méthode nous permet d'obtenir des jeux de données $\tilde{\mathcal{D}}_k$ similaires mais qui ont chacun leurs spécificités. En effet la probabilité pour qu'un exemple de la base ne soit pas sélectionné lors de la création d'un sous-échantillon se traduit par :

$$P = \left(1 - \frac{1}{n}\right)^n$$

$$\lim_{N \rightarrow \infty} P = \frac{1}{e} \approx 0.3679$$

Une fois ces K jeux de données générés, on entraîne le même algorithme sur chaque $\tilde{\mathcal{D}}_k$ ce qui nous permettra d'obtenir K classificateurs f_k qui seront tous différents à cause de cet apprentissage sur des bases différentes malgré un code identique en tout autre point. Une fois cela fait on agrège ces K classificateurs soit par vote majoritaire soit par moyenne :

$$f_{maj}(\mathbf{x}) = \arg \max_y \sum_{k=1}^K \delta(f_k(\mathbf{x}), y)$$

$$f_{moy}(\mathbf{x}) = \frac{1}{K} \sum_{k=1}^K f_k(\mathbf{x})$$

La conséquence d'un tel agrégat est que chaque classifieur commet des erreurs différentes à cause des sous-échantillon $\tilde{\mathcal{D}}_k$ qui sont différents et moins riches en information que \mathcal{D} . Ces différences entre les erreurs commises par les f_k vont assurer une variance d'apprentissage plus faible au classifieur agrégé. Ainsi la méthode du bagging permet de régulariser le processus de prédiction.

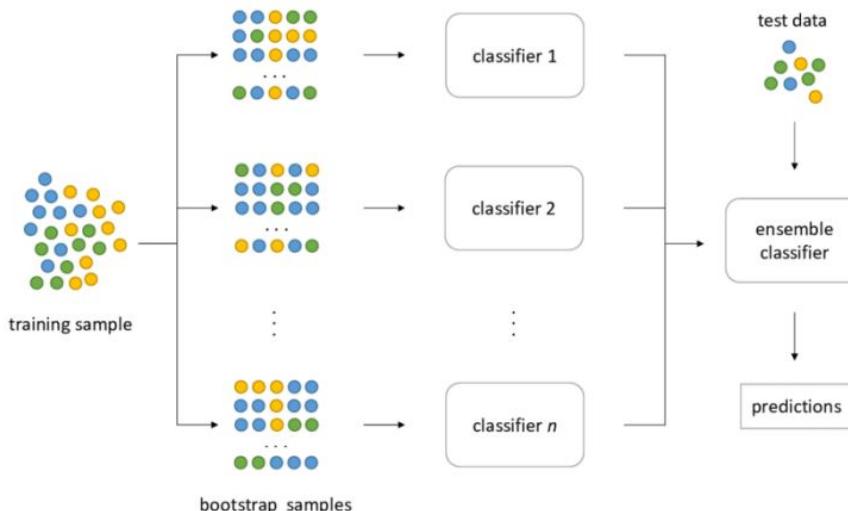


Figure 2.9: Schéma de principe du Bagging [16])

2.4 Les forêts aléatoires (ou Random Forests)

Les forêts aléatoires ont pour principe de construire plusieurs arbres, d'où le nom "forêt", se basant non seulement sur des échantillons différents, à l'instar du bagging, mais aussi sur des variables différentes. Si, par exemple, on se trouve dans un problème à p variables, on commence par sélectionner $q < p$ variables aléatoirement puis on effectue les tests sur ces variables pour construire des arbres de décision. En pratique on utilise $q \approx \sqrt{p}$ pour des problèmes de classification et $q \approx \frac{p}{3}$ en régression.

La structure de ces arbres comprend les paramètres de contrôle de la complexité que sont la profondeur maximale, le nombre d'échantillons par nœud, le critère de pureté des nœud (seuil à partir duquel on estime qu'il s'agit d'un nœud terminal) etc... À cela s'ajoute une part d'aléatoire sur les attributs, les données et les tests.

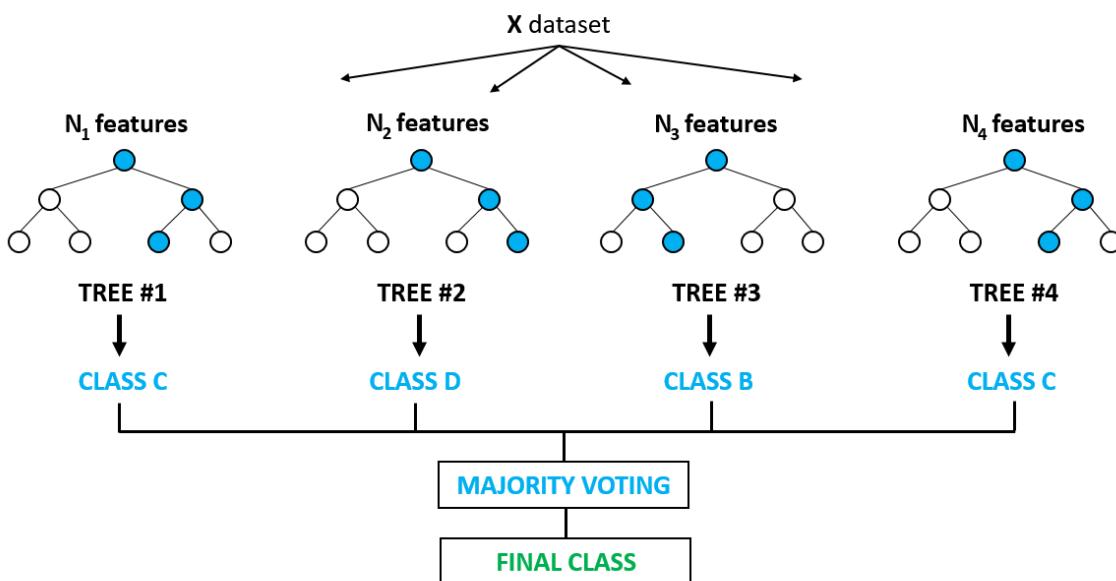


Figure 2.10: Schéma de principe d'une forêt aléatoire [17]

2.4.1 Algorithme de construction d'une forêt aléatoire

POUR $k = 1 \dots K$:

Tirage de $\tilde{\mathcal{D}}_k$ de même taille que \mathcal{D} (Bagging)

Tirage avec remise de q attributs A^j parmi les M possibles

Construction de l'arbre G_k avec des seuils aléatoires

Construction de f_k la fonction de décision de G_k dont les feuilles sont remplies avec $\tilde{\mathcal{D}}_k$

Aggrégation :

$$f_{moy}(\mathbf{x}) = \frac{1}{K} \sum_{k=1}^K f_k(\mathbf{x}) \text{ (régression)}$$

$$f_{maj}(\mathbf{x}) = \arg \max_y \sum_{k=1}^K \delta(f_k(\mathbf{x}), y) \text{ (classification)}$$

2.4.2 Biais et variance

Pour évoquer les notions de biais et de variance, prenons l'exemple de la régression :

$$y = f(\mathbf{x}) + \epsilon$$

Dans un problème comme celui-ci on peut identifier deux grandes sources d'aléatoires : le bruit ϵ auquel sont soumises les données ainsi que l'échantillonnage de ces données d'apprentissage D . Pour un prédicteur appris $\hat{f}_D(x)$ on définit les trois notions suivantes :

L'**erreur** est l'écart quadratique moyen entre la prédiction et la valeur idéale (qui on le rappelle est connue pour les données d'apprentissage).

Le **biais** est l'erreur de la prédiction moyenne par rapport à la valeur idéale.

La **variance** est l'écart quadratique moyen entre une prédiction et la prédiction moyenne.

L'erreur pour un x donné dans un tel problème de régression peut se décomposer en trois termes distincts :

$$\begin{aligned} Err(\mathbf{x}) &= E_D[(y - \hat{f}_D(\mathbf{x}))^2] \\ &= \underbrace{\epsilon^2}_{\text{bruit}^2} + \underbrace{(E_D[\hat{f}_D(\mathbf{x})] - y)^2}_{\text{biais}^2} + \underbrace{E_D[(E_D[\hat{f}_D(\mathbf{x})] - \hat{f}_D(\mathbf{x}))^2]}_{\text{variance}} \end{aligned}$$

L'origine de l'erreur de généralisation est double, cependant les deux termes sont difficiles à contrôler individuellement. Une telle décomposition est plus difficile à obtenir dans le cadre d'un problème de classification car les sources d'aléatoire peuvent être plus compliquées à déterminer, les comportements sont néanmoins comparables.

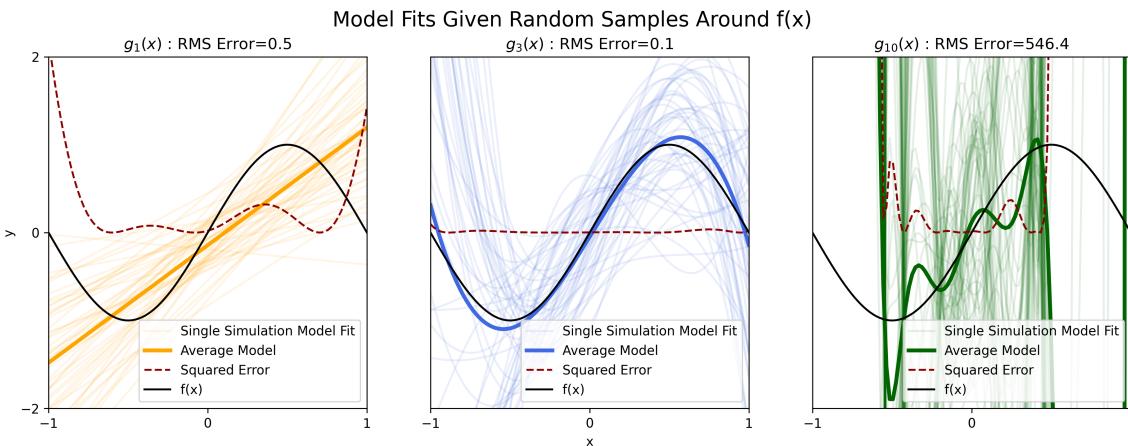


Figure 2.11: Simulation d'une régression pour 50 échantillons et des polynômes de degrés 1, 3 et 10 [18]

Sur la Figure 2.11 on peut observer que pour un polynôme de degré 1 on a une variance faible (les différentes courbes approximées sont proches les unes des autres) mais on a un biais important (la prédiction moyenne est loin d'être une bonne approximation de la fonction recherchée). Pour le polynôme de degré 3 on a une variance et un biais qui sont relativement faibles. Enfin pour le polynôme de degré 10 on a une variance très importante et un biais assez faible.

L'intérêt des approches ensemblistes est que l'on introduit une source d'aléatoire supplémentaire: choix des sous-échantillons, du sous-ensemble de variables, etc...

Les K arbres nous fournissent les prédictions $\hat{f}_1(\mathbf{x}), \dots, \hat{f}_K(\mathbf{x})$ qui sont des variables aléatoires identiquement distribuées de variance σ^2 et de corrélation $\rho = \text{Corr}(\hat{f}_i(\mathbf{x}), \hat{f}_j(\mathbf{x})), \forall i \neq j$. Il en découle

que la variance du prédicteur ensembliste $\hat{f}(\mathbf{x}) = \frac{1}{K} \sum_{k=1}^K \hat{f}_k(\mathbf{x})$ est :

$$\text{var}(\hat{f}(\mathbf{x})) = \frac{1}{K^2} \sum_{i=1}^K \sum_{j=1}^K \text{Cov}(\hat{f}_i(\mathbf{x}), \hat{f}_j(\mathbf{x}))$$

$$\text{var}(\hat{f}(\mathbf{x})) = \frac{1}{K^2} \left(\sum_{i=1}^K \text{var}(\hat{f}_i(\mathbf{x})) + \sum_{i \neq j} \text{Cov}(\hat{f}_i(\mathbf{x}), \hat{f}_j(\mathbf{x})) \right)$$

$$\text{var}(\hat{f}(\mathbf{x})) = \frac{1}{K^2} (K\sigma^2 + K(K-1)\rho\sigma^2)$$

$$\text{var}(\hat{f}(\mathbf{x})) = \rho\sigma^2 + \frac{1-\rho}{K}\sigma^2$$

On déduit de cette formule que l'on a intérêt à construire des prédicteurs individuels indépendants, c'est-à-dire avoir $\rho \approx 0$, et avoir un grand nombre K de ces prédicteurs pour minimiser le rapport.

2.4.3 Résumé concernant les Random Forests

Les forêts aléatoires constituent l'une des approches les plus préformantes et les plus simples à mettre en place en pratique. En effet cette méthode permet d'obtenir des arbres plus décorrélés que par un simple bagging ce qui lui octroie une robustesse accrue et, comme elle utilise des arbres de décision, reste pertinente même en grande dimension. Le seul bémol restant est le temps d'entraînement à cause de la construction de multiples arbres, mais cet obstacle peut être contourné assez aisément grâce à la facilité avec laquelle il est possible de paralléliser le code.

Les Random Forests peuvent être utilisées dans la résolution de problèmes de classification et de régression à l'instar des arbres de décision, puisque ces forêts ne sont en réalité qu'un ensemble d'arbres. Elles peuvent traiter des données aussi bien numériques que symboliques. On choisit en général d'utiliser les Random Forests avec une faible profondeur d'arbre (généralement entre 2 et 5), les autres hyper-paramètres sont à estimer par validation croisée mais elles ont l'avantage de ne pas dépendre crucialement de ces paramètres.

2.5 Boosting

2.5.1 Principe de la méthode du boosting

Pour pouvoir utiliser le boosting il faut disposer d'un ensemble de données $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$ pour lequel on a $\forall i \in [|1, n|], y_i \in \{-1, 1\}$. On suppose que l'on dispose d'un ensemble (ou famille) \mathcal{H} de classifieurs f_k à valeur dans $\{-1, 1\}$. Ces classifieurs ne sont pas forcément performants et seront appelés *weak learners*.

L'objectif du boosting est de construire un classifieur performant à partir de ces *weak learners* en utilisant une moyenne pondérée de ces classifieurs :

$$f(\mathbf{x}) = \sum_{k=1}^K \alpha_k f_k(\mathbf{x})$$

Pour construire ce classifieur appelé *strong learner*, il va falloir déterminer quels poids associer à chacun des *weak learners*. Pour cela on va utiliser l'algorithme Adaboost (ou Adaptive boosting algorithm).

2.5.2 L'algorithme Adaboost

L'Adaptive boosting algorithm vise à la minimisation de l'erreur globale de f de manière itérative. Pour cela l'algorithme va, à chaque itération k , modifier f^k (le classifieur obtenu après la k -ième itération) de manière à *donner plus de poids aux données difficiles*, c'est-à-dire aux données qui ont été mal classées précédemment, ce qui va permettre de corriger les erreurs commises par f^{k-1} :

L'algorithme Adaboost dans le cas $y \in \{-1, 1\}$:

Initialiser les poids liés aux données :

$$\mathbf{d}^0 \leftarrow (\frac{1}{K}, \frac{1}{K}, \dots, \frac{1}{K})$$

POUR $k = 1, \dots, K$:

Entraîner f_k sur les données \mathcal{D} pondérées par \mathbf{d}^{k-1} $\left(f_k = \arg \min_f \sum_i d_i^{k-1} (1 - \delta(y_i, f(\mathbf{x}_i))) \right)$

Prédire $\hat{y}_i = f_k(\mathbf{x}_i) \forall i$

Calculer l'erreur pondérée $\epsilon^k = \sum_i d_i^{k-1} (1 - \delta(y_i, \hat{y}_i))$

Calculer les paramètres adaptatifs $\alpha^k = \frac{1}{2} \log \left(\frac{1-\epsilon^k}{\epsilon^k} \right)$

Re-pondérer les données \mathbf{d}^k avec $\forall i, d_i^k = d_i^{k-1} \exp(-\alpha^k y_i \hat{y}_i)$

Classifieur pondéré final : $f(\mathbf{x}) = \operatorname{sgn} \left(\sum_{k=1}^K \alpha^k f_k(\mathbf{x}) \right)$

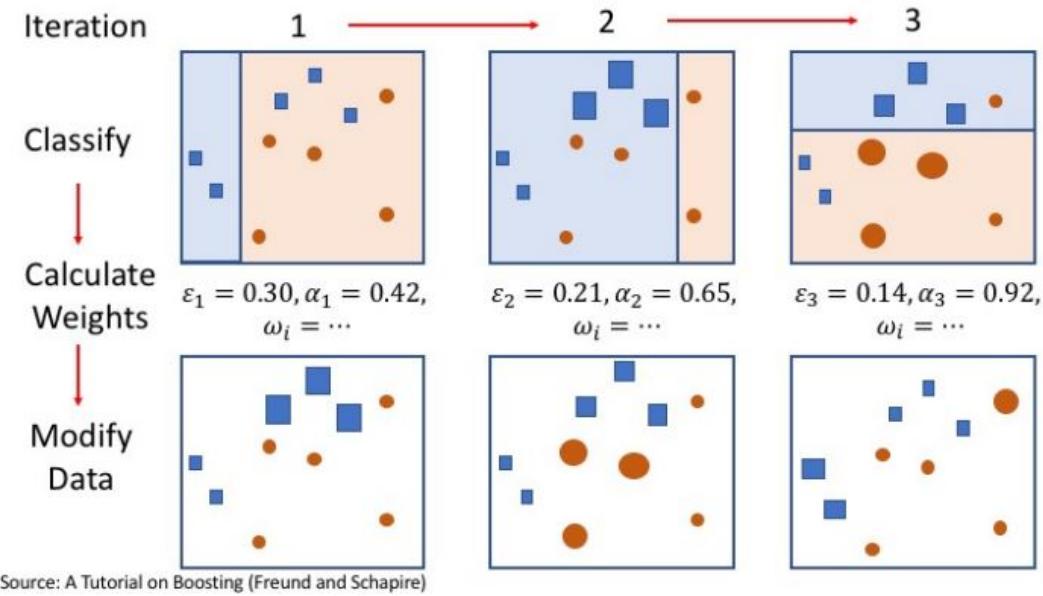


Figure 2.12: Apprentissage séquentiel des classificateurs et des pondérations par l'algorithme Adaboost [19]

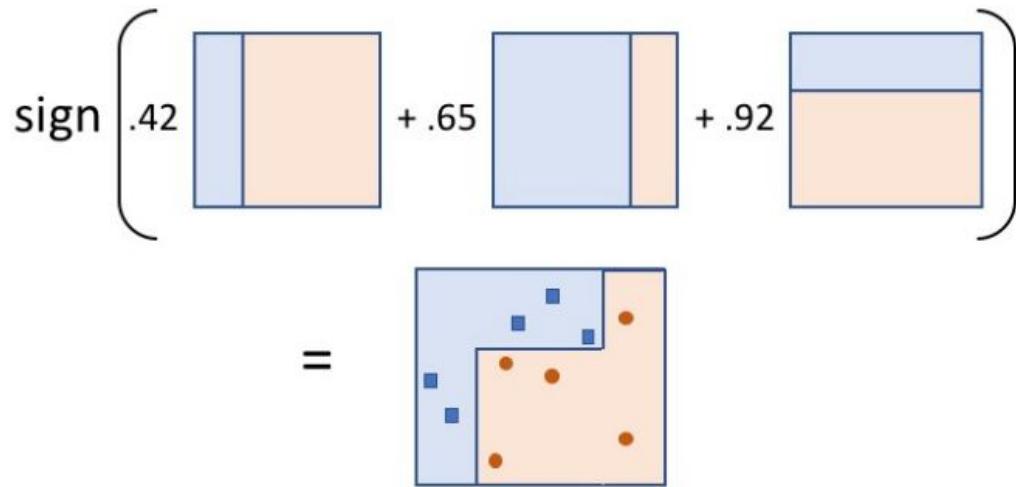


Figure 2.13: Classifieur final obtenu après l'apprentissage présenté en Figure 2.12 [19]

2.5.3 Gradient Boosting

Dans cette section nous allons aborder une variante de boosting : le Gradient Boosting. On suppose toujours que l'on dispose d'un ensemble de données $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$ où $y_i \in \{-1, 1\}$ ainsi que d'un ensemble \mathcal{H} de classificateurs f_k qui ne sont pas nécessairement performants. Cette fois-ci le principe est de construire pas-à-pas un classificateur performant en ajoutant un à un les *weak learners* pondérés. Ainsi l'objectif est de construire itérativement un *strong learner* :

$$f^T(\mathbf{x}) = \sum_{t=1}^T \alpha^t f_t(\mathbf{x}) = f^{T-1}(\mathbf{x}) + \alpha^T f^T(\mathbf{x})$$

Pour construire ce classifieur on va, à chaque étape, chercher à minimiser le risque empirique :

$$\hat{\mathcal{R}}_n(f^T) = \frac{1}{n} \sum_{i=1}^n L(y_i, f^T(\mathbf{x}_i))$$

Si on revient sur l'algorithme Adaboost, il s'agit en réalité d'un programme utilisant le gradient boosting avec une fonction de coût $L(y, f(\mathbf{x})) = \exp(-y.f(\mathbf{x}))$. En effet Adaboost peut être vu comme une construction itérative d'un classifieur optimal par minimisation du risque empirique à chaque itération. Il existe de nombreuses autres versions utilisant différentes pénalités :

- LogitBoost : $L(y, f(\mathbf{x})) = \log_2(1 + \exp(-2y.f(\mathbf{x})))$
- L_2 Boost : $L(y, f(\mathbf{x})) = \frac{(y-f(\mathbf{x}))^2}{2}$
- Doomll : $L(y, f(\mathbf{x})) = 1 - \tanh(y.f(\mathbf{x}))$
- Savage : $L(y, f(\mathbf{x})) = \frac{1}{(1+\exp(2y.f(\mathbf{x})))^2}$

On peut se demander pourquoi cette méthode est appelée *Gradient* Boosting, cela vient du fait que l'on cherche à minimiser le risque empirique à chaque étape et que l'on pourrait donc percevoir $\alpha^T f_T(\mathbf{x})$ comme le *weak learner* qui approxime le mieux le pas d'une descente de gradient dans l'espace des fonctions de classification. Une version exacte de la descente de gradient donne les Gradient Boosting Models :

$$f^T(\mathbf{x}) = f^{T-1}(\mathbf{x}) + \alpha^T \sum_{i=1}^n \nabla_{f^{T-1}} L(y_i, f^{T-1}(\mathbf{x}_i))$$

2.5.4 Résumé concernant le Boosting

Il faut retenir que le boosting consiste en une agrégation adaptative de classificateurs moyens. Cette méthode très efficace permet d'améliorer n'importe quel ensemble de classificateurs et est assez facile à mettre en place, cependant le gradient boosting est un peu plus complexe. Le boosting est toutefois assez sensible aux données aberrantes et atteint facilement un état de sur-apprentissage. C'est pourquoi les *weak learners* ne doivent pas être trop performants sous peine d'overfitting. Au-delà de ces problèmes il faut choisir une pénalité qui sera adaptée au bruit des données. Les variantes du boosting peuvent être utilisées pour résoudre des problèmes de classification ou de régression.

Chapter 3

Régularisation / SVM

3.1 Régularisation

Lorsque les variables du problème sont fortement corrélées, ou que leur nombre dépasse celui des observations, la matrice $\Phi \in \mathcal{M}_{n,M}$ (définie en section 1.5.1) représentant nos données ne peut pas être de rang M , c'est-à-dire que la famille des colonnes de cette matrice n'est pas libre. Cela entraîne que la matrice $\Phi^T \Phi$ n'est pas inversible et qu'il existe plusieurs solutions à un problème de régression linéaire par minimisation des moindres carrés. L'absence d'unicité de la solution présente un fort risque de sur-apprentissage : comment pourrait-on garantir que le modèle que l'on a sélectionné est celui qui généralise le mieux ?

Afin de limiter ce risque d'overfitting, nous allons ajouter un contrôle des valeurs des coefficients de régression affectés aux variables. En effet si on reprend l'exemple de la section 1.5.1 :

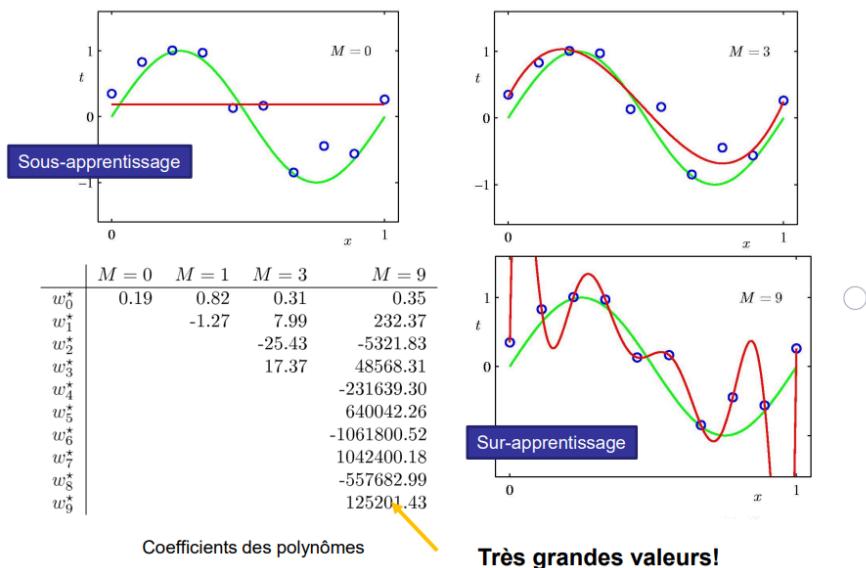


Figure 3.1: Coefficient des polynômes (tiré de *Pattern Recognition and Machine Learning* par Christopher M. Bishop)

On voit que, dans le cas du sur-apprentissage, les coefficients du polynôme sont grands. On peut donc penser que contrôler les valeurs de ces coefficients pourrait améliorer la qualité de la prédiction.

3.1.1 Définition de la régularisation

On appelle **régularisation** le fait d'apprendre un modèle en minimisant la somme du risque empirique sur les données d'entraînement et d'un terme de contrainte Ω sur les solutions possibles :

$$f = \arg \min_{f'} \frac{1}{n} \sum_{i=1}^n L(f'(\mathbf{x}_i, \boldsymbol{\omega}), y_i) + \lambda \Omega(f')$$

où le coefficient de régularisation $\lambda \in \mathbb{R}_+$ contrôle l'importance relative de chacun des termes.[13]

3.1.2 Modélisation mathématique de la régularisation

Pour revenir à notre exemple, en considérant que l'on utilise toujours la fonction de coût des moindres carrés, on peut utiliser la régression dite *ridge* en utilisant le terme de contrainte suivant:

$$\Omega(\boldsymbol{\omega}) = \|\boldsymbol{\omega}\|_2^2 = \sum_{j=0}^M \omega_j^2$$

Ainsi la **régression ridge** est le modèle $f : \mathbf{x} \mapsto \boldsymbol{\omega}^T \mathbf{x}$ avec :

$$\boldsymbol{\omega} = \arg \min_{\boldsymbol{\omega}' \in \mathbb{R}^{p+1}} \|\mathbf{y} - \Phi \boldsymbol{\omega}'\|_2^2 + \lambda \|\boldsymbol{\omega}'\|_2^2$$

On peut noter que cette régularisation est également utilisée dans le domaine des réseaux de neurones, où elle est appelée *weight decay*. Attelons-nous maintenant à la résolution de ce problème. Il s'agit d'un problème d'optimisation convexe puisqu'il s'agit de la minimisation d'une forme quadratique. Ainsi pour obtenir la solution il suffit d'annuler le gradient en $\boldsymbol{\omega}$ de la fonction :

$$\nabla_{\boldsymbol{\omega}} (\|\mathbf{y} - \Phi \boldsymbol{\omega}\|_2^2 + \lambda \|\boldsymbol{\omega}\|_2^2) = 0$$

Si l'on note $I_M \in \mathcal{M}_M(\mathbb{R})$ la matrice identité en dimension M , la solution s'écrit alors :

$$\boldsymbol{\omega} = (\lambda I_M + \Phi^T \Phi)^{-1} \Phi^T \mathbf{y}$$

La matrice $\lambda I_M + \Phi^T \Phi$ est bien inversible car $\lambda > 0$. On en déduit qu'il existe toujours une unique solution explicite au problème : la régularisation a transformé un problème potentiellement mal posé en problème bien posé. Ainsi si on pénalise les grandes valeurs des coefficients du polynôme, on obtient une fonction aux variations moins prononcées qui est susceptible de mieux répondre au problème posé.

3.1.3 Résultats de la régularisation

Voyons quels résultats peuvent être obtenus en modifiant la valeur du paramètre λ :

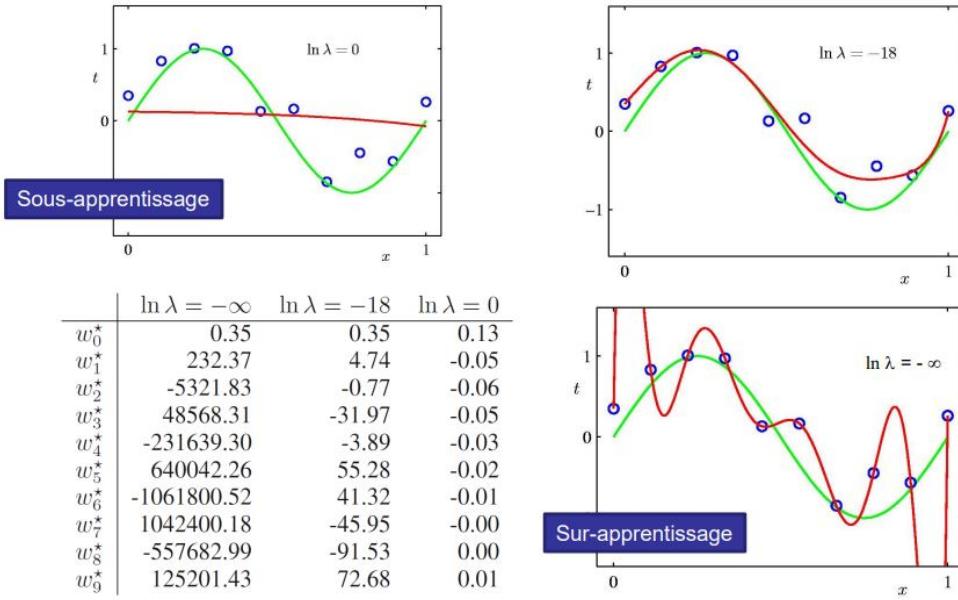


Figure 3.2: Coefficient des polynômes en fonction de la valeur de $\ln \lambda$ (tiré de *Pattern Recognition and Machine Learning* par Christopher M. Bishop)

Ce qui est présenté sur la Figure 3.2, à savoir la variation des valeurs des coefficients de régression en fonction de la valeur du coefficient de régularisation λ , constitue ce que l'on appelle le **chemin de régularisation**. Ce chemin permet de comprendre l'effet de la régularisation sur les valeurs des coefficients du polynôme représenté par ω :

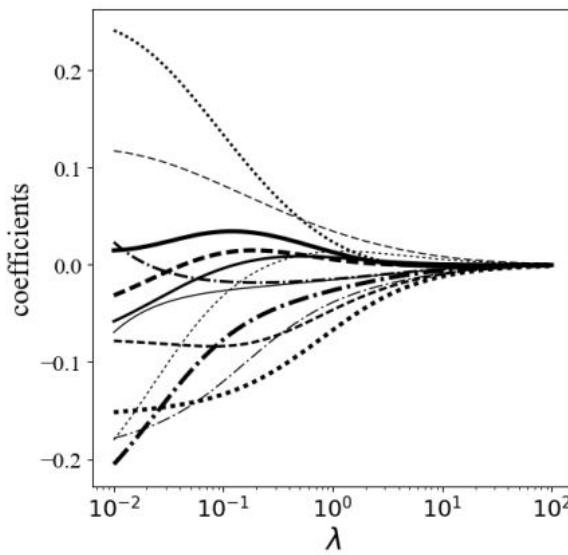


Figure 3.3: Chemin de régularisation de la régression ridge pour des données à 12 variables, chaque ligne représente l'évolution d'un coefficient [13]

À partir des valeurs de la Figure 3.2 on peut tracer la courbe de l'erreur quadratique en fonction de $\ln \lambda$ pour un polynôme de degré 9 :

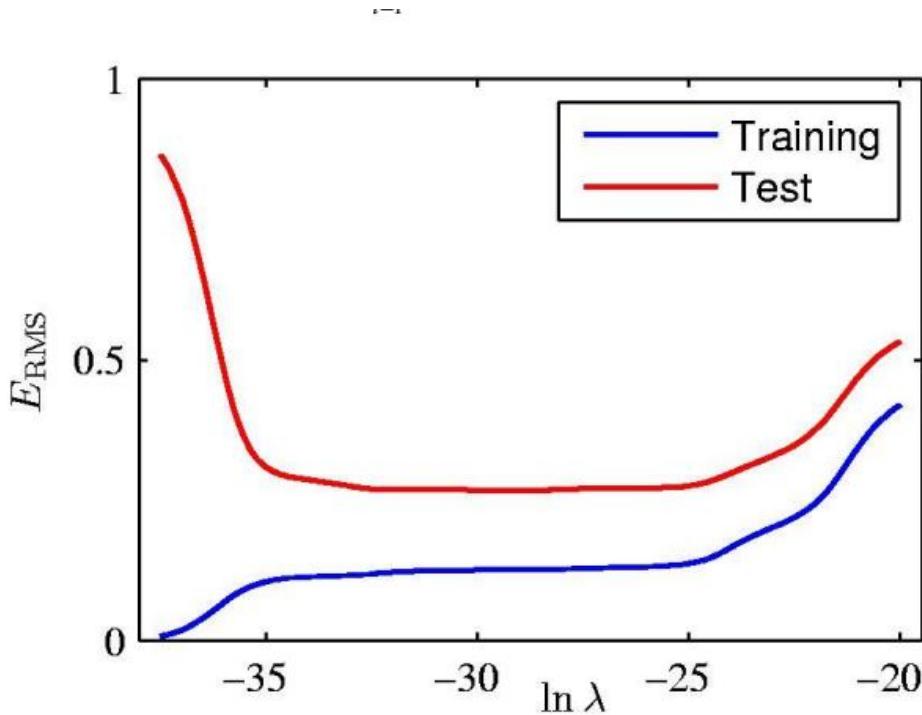


Figure 3.4: Erreur quadratique en fonction de la valeur de $\ln \lambda$ (tiré de *Pattern Recognition and Machine Learning* par Christopher M. Bishop)

Si on compare ces valeurs à celles obtenues en Figure 1.10 en section 1.5.2 on voit que l'erreur sur les données de test est bien inférieure (≈ 0.30 avec la régularisation contre presque 1 sans).

3.2 Validation croisée

3.2.1 Qu'est-ce que la validation croisée ?

La validation croisée est une méthode visant à améliorer la phase de validation d'un modèle. En effet la séparation de la base de données \mathcal{D} en un ensemble d'entraînement et un ensemble de test contient nécessairement une part d'aléatoire. Il n'est pas impossible de créer des jeux de données qui ne sont pas représentatifs du problème, notamment lorsque la base \mathcal{D} est de faible taille. La validation croisée permet d'estimer l'erreur de généralisation à partir des données d'apprentissage grâce à une petite astuce.

La procédure à suivre dans le cadre de la validation croisée est la suivante :

Partitionner la base \mathcal{D} en K échantillons de tailles sensiblement égales $\mathcal{D}_1, \mathcal{D}_2, \dots, \mathcal{D}_K$
Pour chaque valeur de $k = 1, \dots, K$:

Entraîner un modèle f_k sur $\bigcup_{l \neq k} \mathcal{D}_l$

Évaluer ce modèle sur \mathcal{D}_k en mesurant l'erreur empirique $\widehat{R}_n^k(f_k)$

Chaque partition de \mathcal{D} en \mathcal{D}_k et $\bigcup_{l \neq k} \mathcal{D}_l$ est appelée un **fold** de la validation croisée. Il serait possible d'objecter que l'on ne résout pas le problème de la création arbitraire de sous-ensembles de la base de données \mathcal{D} . Toutefois ces multiples découpes vont permettre de moyenner les effets aléatoires de la partition. Une fois que l'on a parcouru tous les folds, on peut calculer l'erreur estimée finale en moyennant les erreurs mesurées :

$$\widehat{R}_n(f) = \frac{1}{n} \sum_{k=1}^n \widehat{R}_n^k(f)$$

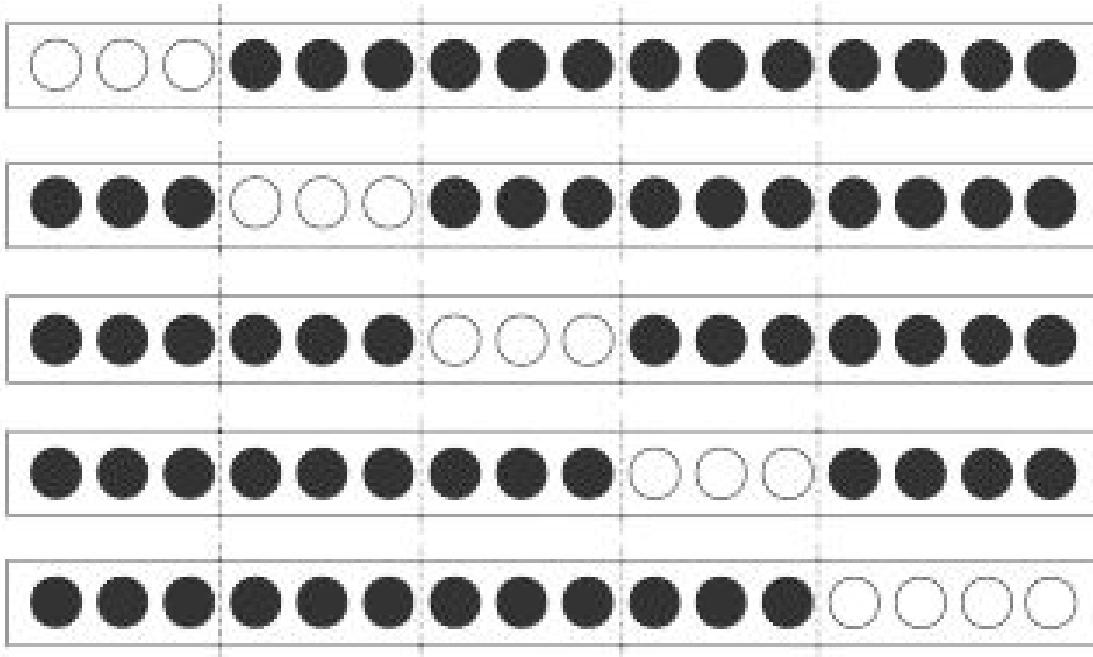


Figure 3.5: Exemple de validation croisée en 5 folds, les jeux de validation sont en blanc et ceux d'entraînement sont en noir [13]

3.2.2 Les intérêts de la validation croisée

La validation croisée possède de multiples utilités, la première d'entre elles étant de permettre l'estimation de la "vraie" erreur de prédiction. De plus elle permet d'estimer la variance d'apprentissage, cependant le biais reste lui inconnu. Enfin elle peut permettre le réglage des "hyper-paramètres", comme le coefficient de régularisation par exemple.

Néanmoins il y a d'autres sources d'aléatoire qui ne relèvent pas de la validation croisée comme par exemple les choix des attributs servant à construire des arbres dans les random forests, la découpe de la base de données dans un modèle utilisant le bagging ou encore l'initialisation et l'optimisation des réseaux de neurones.

3.2.3 La validation croisée stratifiée

Une validation croisée est dite **stratifiée** si la moyenne des étiquettes est sensiblement identique dans chacun des \mathcal{D}_k :

$$\frac{1}{|\mathcal{D}_1|} \sum_{i \in \mathcal{D}_1} y_i \approx \frac{1}{|\mathcal{D}_2|} \sum_{i \in \mathcal{D}_2} y_i \approx \dots \approx \frac{1}{|\mathcal{D}_K|} \sum_{i \in \mathcal{D}_K} y_i \approx \frac{1}{|\mathcal{D}|} \sum_{i \in \mathcal{D}} y_i$$

L'objectif de cette procédure est d'éviter d'avoir la malchance d'avoir un échantillon \mathcal{D}_k ne contenant que des exemples de la même classe dans son jeu d'entraînement, ce qui le pousserait inévitablement à prédire cette classe dans toutes les situations, ce qui serait très mauvais.

3.2.4 La validation croisée *leave-one-out*

Une autre stratégie possible pour la création des \mathcal{D}_k est tout simplement de prendre $K = n$, c'est-à-dire que chaque échantillon \mathcal{D}_k ne contient qu'une seule observation (\mathbf{x}_k, y_k) . Ainsi chaque fold est composé de $n - 1$ données d'entraînement et d'un jeu de test de taille 1, c'est ce que l'on appelle la validation croisée **leave-one-out** : on met de côté un unique exemple pour chaque fold.

Cette méthode présente tout de même deux inconvénients majeurs. En premier lieu elle nécessite un grand temps de calcul puisqu'il est nécessaire d'entraîner n modèles au lieu d'en entraîner entre 5 et 10 en temps normal. À cela s'ajoute le fait que les échantillons créés sont très similaires entre eux et que les modèles qui en découleront le seront tout autant.

3.3 Les "Support Vector Machines" (ou SVM)

Il existe deux grands types d'approches en Machine Learning : les approches génératives et les approches discriminatives. Les premières ont pour objectif principal de modéliser les distributions de données pour ensuite pouvoir les exploiter, c'est ce type d'approche que nous avons étudiés dans le chapitre 1. Dans cette section nous allons nous intéresser à la seconde catégorie qui vise à la construction des meilleures frontières possibles entre les données.

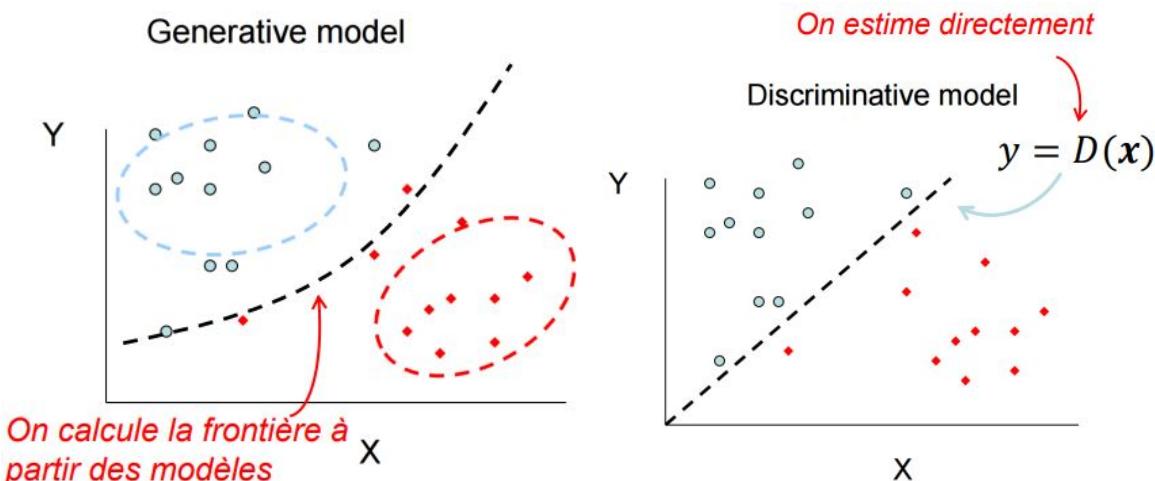


Figure 3.6: Présentation des approches génératives et discriminatives

3.3.1 Position du problème

Dans cette section on se placera dans le cadre d'un problème de classification binaire, avec $\forall i \in [|1, n|], y_i \in \{-1, 1\}$, sur des données dans \mathbb{R}^p . De plus ici on fera l'hypothèse que les données sont *linéairement séparables*, c'est-à-dire qu'il existe un hyperplan de \mathbb{R}^p tel que tous les points \mathbf{x}_i positifs ($y_i = 1$) soient d'un côté de cet hyperplan et tous les points négatifs de l'autre. Un tel hyperplan est dit *séparateur*.

L'équation d'un hyperplan séparateur est toujours de la forme :

$$b + \boldsymbol{\omega}^T \cdot \mathbf{x} = 0$$

Le classifieur est défini par l'expression :

$$f(\mathbf{x}, \boldsymbol{\omega}) = \text{sgn}(b + \boldsymbol{\omega}^T \cdot \mathbf{x})$$

Ainsi pour un problème à deux classes:

$$\begin{cases} y = +1 & \text{si } b + \boldsymbol{\omega}^T \cdot \mathbf{x} > 0 \\ y = -1 & \text{si } b + \boldsymbol{\omega}^T \cdot \mathbf{x} < 0 \end{cases}$$

Attribuer la classe de y revient à savoir où se situe la donnée x par rapport à l'hyperplan H . Ce qui nous donne l'erreur :

$$\hat{R}_n(f) = \frac{1}{n} \sum_{i=1}^n \delta(y_i, f(\mathbf{x}, \boldsymbol{\omega}), -1)$$

3.3.2 Choix de l'hyperplan

On a fait l'hypothèse que les données étaient linéairement séparables, cependant cette hypothèse n'implique pas l'existence d'un, mais d'une infinité d'hyperplans séparateurs comme représenté en Figure 3.7. Cette situation amène donc un problème : quel hyperplan devrait-on choisir ?

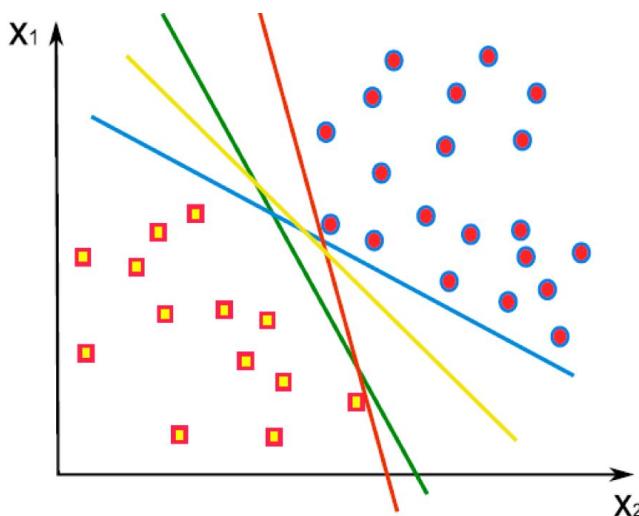


Figure 3.7: Exemple de données linéairement séparables dans \mathbb{R}^2 [20]

Pour pouvoir répondre à cette question, commençons par introduire la notion de marge. La marge γ d'un hyperplan séparateur est la distance de cet hyperplan à l'observation du jeu d'entraînement la plus proche.

On va donc chercher à déterminer l'hyperplan séparateur H pour lequel la marge est maximale, c'est ce que l'on appelle le classifieur à **marge large**. Il y a alors au moins une observation positive et une observation négative situées à une distance γ de l'hyperplan séparateur, dans le cas contraire on pourrait rapprocher H des observations situées à une distance supérieure pour augmenter γ .

Supposons que nous disposons de l'hyperplan séparateur H , on peut alors définir les deux hyperplans H_+ et H_- qui sont parallèles et situés à une distance γ de H . H_+ (respectivement H_-) contient alors au moins une observation positive (respectivement négative). On appelle **vecteurs de support** les observations situées à une distance γ de H , elles appartiennent soit à H_+ soit à H_- et on dit qu'elles "soutiennent" ces deux hyperplans.

C'est de là que vient le nom de la méthode *Support Vector Machines*, en français *machine à vecteurs de support* ou encore *séparatrice à vaste marge* pour conserver l'acronyme SVM.

La zone située entre H_+ et H_- est appelée **zone d'indécision**, elle ne contient aucune observation.

3.3.3 Formulation de la SVM à marge rigide

Nous allons maintenant chercher à déterminer l'équation de l'hyperplan séparateur H . Nous savons que cette équation sera de la forme $b + \omega^T \cdot x = 0$. Ainsi nous savons que les hyperplans H_+ et H_- sont définis par des équations de la forme $b + \omega^T \cdot x = \pm C$ avec C une constante. On peut fixer $C = 1$ sans perdre en généralité car le fait de multiplier ω et b ne modifie pas l'équation de H .

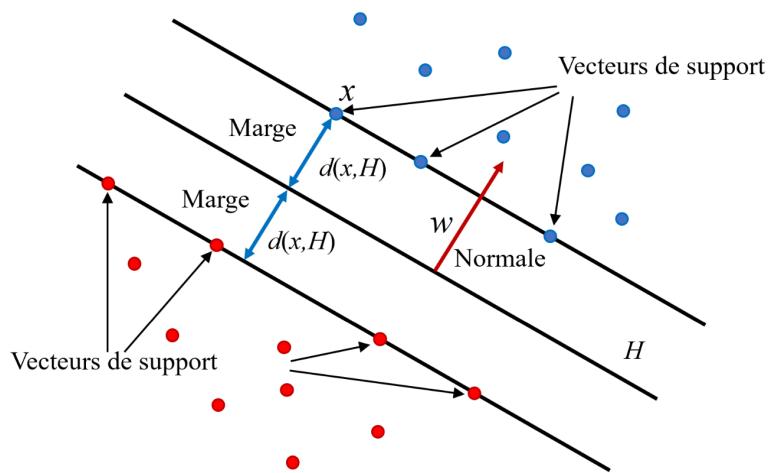


Figure 3.8: Représentation graphique de la recherche de H [21]

On a donc $\forall i$ tel que $y_i = 1 : b + \omega^T \cdot \mathbf{x}_i \geq 1$ et $\forall i$ tel que $y_i = -1 : b + \omega^T \cdot \mathbf{x}_i \leq -1$ avec égalité pour les vecteurs de support.

Déterminons maintenant quelle est la formule permettant d'obtenir la distance entre l'hyperplan séparateur et un point. Pour ce faire nous allons nous placer dans le cas $\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \in \mathbb{R}^2$ et $\omega \in \mathbb{R}^2$.

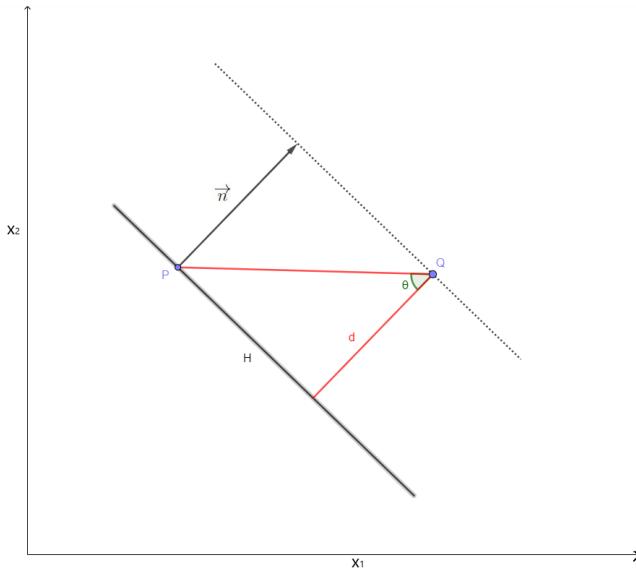


Figure 3.9

Comme illustré sur la Figure 3.9, on dispose d'un point P de coordonnées $\mathbf{x}_P = \begin{pmatrix} x_{1P} \\ x_{2P} \end{pmatrix}$ que l'on supposera dans l'hyperplan séparateur et d'un point Q de coordonnées $\mathbf{x}_Q = \begin{pmatrix} x_{1Q} \\ x_{2Q} \end{pmatrix}$ que l'on supposera dans la marge. On note d la distance entre Q et H .

On a que :

$$\cos(\overrightarrow{PQ}, \vec{n}) = \cos \theta = \frac{d}{\|\overrightarrow{PQ}\|}$$

D'où :

$$d = \|\overrightarrow{PQ}\| \cos \theta$$

De plus on a :

$$\langle \overrightarrow{PQ}, \vec{n} \rangle = \|\overrightarrow{PQ}\| \|\vec{n}\| \cos \theta$$

On sait que $d > 0$, il découle alors de la relation précédente que :

$$d = \frac{|\langle \overrightarrow{PQ}, \vec{n} \rangle|}{\|\vec{n}\|}$$

À cela s'ajoute que, par définition, $\vec{n} = \omega$, ce qui conduit à :

$$d = \frac{|\omega^T \cdot \mathbf{x}_Q - \omega^T \cdot \mathbf{x}_P|}{\|\omega\|}$$

Or $P \in H$ donc on a $b + \omega^T \mathbf{x}_P = 0$ et donc $\omega^T \mathbf{x}_P = -b$ ce qui donne :

$$d = \frac{|b + \omega^T \cdot \mathbf{x}_Q|}{\|\omega\|}$$

Ainsi la distance entre un point et H s'obtient avec :

$$d_i = \frac{|b + \omega^T \cdot \mathbf{x}_i|}{\|\omega\|}$$

Ainsi en prenant des vecteurs de support on obtient que la marge normalisée, ou *marge géométrique* (distance en bleu sur la Figure 3.8) vaut $1/\|\omega\|$. Dans ce problème on cherche à maximiser la marge, ce qui revient au problème d'optimisation suivant :

$$\begin{cases} \text{Trouver} & \arg \min_{\omega \in \mathbb{R}^p, b \in \mathbb{R}} \|\omega\|^2 \\ \text{Sous contraintes} & y_i(b + \omega^T \cdot \mathbf{x}_i) \geq 1 \forall i \end{cases}$$

Il s'agit d'un problème quadratique classique, mais qui est soumis à beaucoup de contraintes puisqu'il y en a autant que d'exemples d'apprentissage.

3.3.4 Formulation de la SVM à marge souple

En règle générale les données ne sont pas linéairement séparables, ainsi certains points seront mal classifiés quel que soit l'hyperplan choisi. D'autres seront classifiés correctement mais se situeront dans la zone d'indécision.

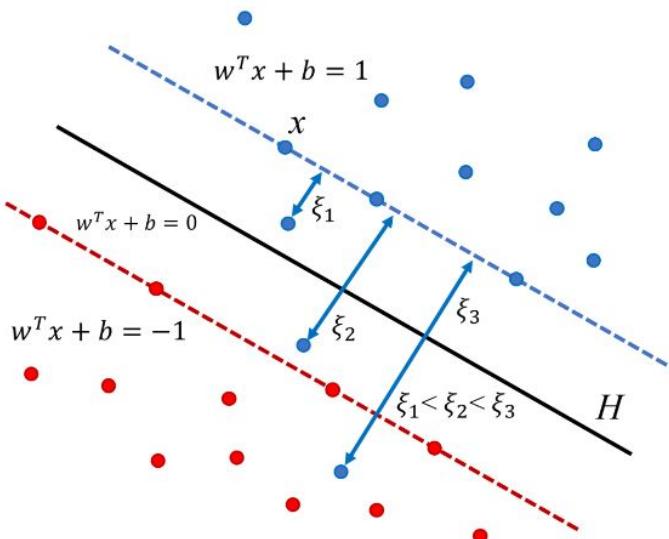


Figure 3.10: Exemple de données non linéairement séparables [21]

Dans le but de pouvoir faire face à ce genre de situation, on cherche maintenant à obtenir une séparation robuste à quelques données non séparées. Pour pouvoir définir une telle séparation on va introduire la notion de **variables de relâchement** (ou **Slack variables** en anglais), que l'on note $\xi_i, \forall i \in [|1, n|]$. Ces variables vont nous permettre de relâcher la contrainte de séparabilité pour chaque exemple :

$$\begin{cases} \text{Trouver} & \arg \min_{\omega \in \mathbb{R}^p, b \in \mathbb{R}} \|\omega\|^2 + C \sum_{i=1}^n \xi_i \\ \text{Sous contraintes} & y_i(b + \omega^T \cdot \mathbf{x}_i) \geq 1 - \xi_i \quad \forall i \\ & \xi_i \geq 0 \end{cases}$$

L'idée est que les ξ_i modélisent l'erreur de classification pour quelques données. En effet dans les cas où il n'y a pas d'erreur on a $y_i(b + \omega^T \cdot \mathbf{x}_i) \geq 1$ et donc $\xi_i = 0$. Par contre en cas d'erreur on a $y_i(b + \omega^T \cdot \mathbf{x}_i) < 1$ et donc $\xi_i = 1 - y_i(b + \omega^T \cdot \mathbf{x}_i) \geq 0$. On impose donc comme contrainte :

$$\forall i, \quad \xi_i = \max(0, 1 - y_i(b + \omega^T \cdot \mathbf{x}_i))$$

L'hyperparamètre de coût C est une constante qui permet de trouver un compromis entre la marge et le relâchement de la contrainte. La marge se voit alors conférée une certaine souplesse, d'où le nom se *SVM à marge souple*. En règle générale les slack variables prennent la forme d'une fonction de coût $L(b + \omega^T \cdot \mathbf{x}_i, y_i)$.

Le problème se résume alors à :

$$\arg \min_{\omega \in \mathbb{R}^p, b \in \mathbb{R}} \|\omega\|^2 + C \sum_{i=1}^n \max(0, 1 - y_i(b + \omega^T \cdot \mathbf{x}_i))$$

Sous cette forme le problème d'optimisation est contraint et l'on peut utiliser d'autres méthodes de résolution, comme une descente de gradient par exemple.

On peut interpréter le SVM comme un cas particulier du formalisme combinant l'erreur empirique combinée à la régularisation en définissant la fonction de pénalisation :

$$Loss(\omega, \mathcal{D}) = \frac{1}{n} \sum_{i=1}^n L(f(\mathbf{x}_i, \omega), y_i) + r(\omega)$$

Avec :

$$\begin{aligned} r(\omega) &= \frac{1}{C} \|\omega\|^2 \\ L(f(\mathbf{x}_i, \omega), y_i) &= \max(0, 1 - y_i(b + \omega^T \cdot \mathbf{x}_i)) \end{aligned}$$

On peut également envisager l'utilisation d'autres fonctions de coût (on pose $y' = f(\mathbf{x}_i, \omega) = b + \omega^T \cdot \mathbf{x}_i$):

$$0/1 \text{ loss : } L(y', y) = \begin{cases} 1 \text{ si } yy' \leq 0 \\ 0 \text{ sinon} \end{cases}$$

$$\text{Squared loss : } L(y', y) = (y - y')^2$$

$$\text{Hinge : } L(y', y) = \max(0, 1 - yy')$$

$$\text{Exponential : } L(y', y) = \exp(-yy')$$

Ces différentes fonctions de coût ont les variations suivantes en fonction de yy' (en fonction de $y - y'$ pour Squared loss) :

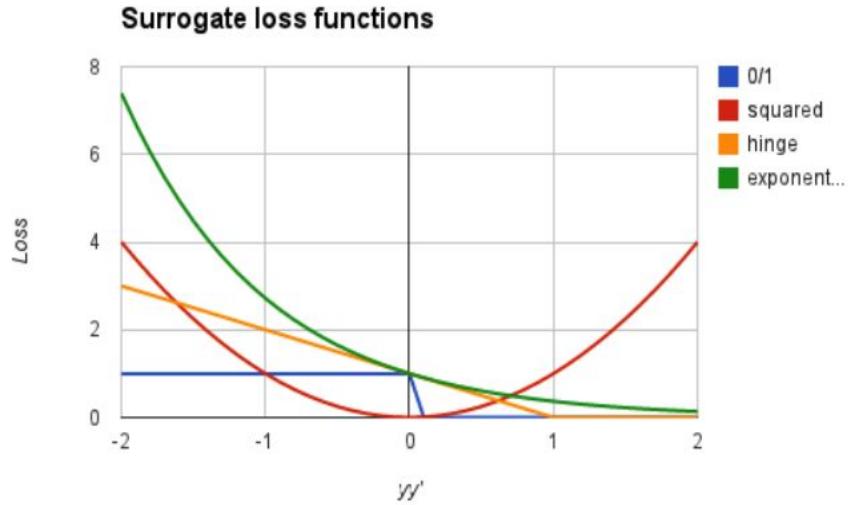


Figure 3.11: Courbe de variation des différentes fonctions de coût [22]

3.3.5 Forme duale du SVM à marge souple

Montrons maintenant que notre problème d'optimisation:

$$\arg \min_{\omega \in \mathbb{R}^p, b \in \mathbb{R}, \xi \in \mathbb{R}^n} \frac{\|\omega\|^2}{2} + C \sum_{i=1}^n \xi_i \quad (3.1)$$

$$\begin{aligned} \text{tel que } y_i(b + \omega^T \cdot \mathbf{x}_i) &\geq 1 - \xi_i \forall i \\ \xi_i &\geq 0 \forall i \end{aligned}$$

Est équivalent au problème :

$$\begin{aligned} \arg \max_{\alpha \in \mathbb{R}^n} & \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \cdot \mathbf{x}_j \\ \text{tel que } & \sum_{i=1}^n \alpha_i y_i = 0; 0 \leq \alpha_i \leq C \forall i \end{aligned} \quad (3.2)$$

Pour ce faire on introduit $2n$ multiplicateurs de Lagrange $\{\alpha_i, \beta_i\}_{i=1,\dots,n}$ et on écrit le lagrangien:

$$\begin{aligned} L : \mathbb{R}^p \times \mathbb{R} \times \mathbb{R}^n \times \mathbb{R}_+^n \times \mathbb{R}_+^n &\rightarrow \mathbb{R} \\ \omega, b, \xi, \alpha, \beta &\mapsto \frac{\|\omega\|^2}{2} + C \sum_{i=1}^n \xi_i - \sum_{i=1}^n \alpha_i (y_i(b + \omega^T \cdot \mathbf{x}_i) - 1 + \xi_i) - \sum_{i=1}^n \beta_i \xi_i \end{aligned} \quad (3.3)$$

La *fonction duale de Lagrange* est donc la fonction :

$$q : \mathbb{R}_+^n \times \mathbb{R}_+^n \rightarrow \mathbb{R}$$

$$\alpha, \beta \mapsto \inf_{\omega \in \mathbb{R}^p, b \in \mathbb{R}, \xi \in \mathbb{R}^n} L(\omega, b, \xi, \alpha, \beta)$$

Ainsi le problème dual équivalent à l'équation 3.1 est :

$$\max_{\alpha \in \mathbb{R}_+^n, \beta \in \mathbb{R}_+^n} \inf_{\omega \in \mathbb{R}^p, b \in \mathbb{R}, \xi \in \mathbb{R}^n} \frac{\|\omega\|^2}{2} + C \sum_{i=1}^n \xi_i - \sum_{i=1}^n \alpha_i (y_i(b + \omega^T \cdot \mathbf{x}_i) - 1 + \xi_i) - \sum_{i=1}^n \beta_i \xi_i \quad (3.4)$$

Le lagrangien est convexe en ω , il est donc minimal quand son gradient en ω est nul, c'est-à-dire si

$$\omega = \sum_{i=1}^n \alpha_i y_i \mathbf{x}_i \quad (3.5)$$

De plus le lagrangien est affine en b , son infimum est donc $-\infty$ sauf si son gradient en b est nul, c'est-à-dire si

$$\sum_{i=1}^n \alpha_i y_i = 0 \quad (3.6)$$

Il est également affine en ξ , de même son infimum est donc $-\infty$ sauf si son gradient en ξ est nul, c'est-à-dire si

$$\forall i, \beta_i = C - \alpha_i \quad (3.7)$$

La fonction duale q est donc maximisée quand les relations 3.6 et 3.7 sont vérifiées. En remplaçant ω grâce à la relation 3.5 dans l'expression de la fonction duale, on peut réécrire 3.4 ainsi :

$$\max_{\alpha \in \mathbb{R}^n} -\frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \cdot \mathbf{x}_j + \sum_{i=1}^n \alpha_i + C \sum_{i=1}^n \xi_i - \sum_{i=1}^n (C - \alpha_i) \xi_i - \sum_{i=1}^n \alpha_i \xi_i$$

tel que $\sum_{i=1}^n \alpha_i y_i = 0, \quad 0 \leq \alpha_i \leq C \quad \forall i$

On retrouve ainsi le problème 3.2.[13]

Afin de pouvoir interpréter ce résultat, introduisons maintenant les conditions de Karush-Kuhn-Tucker. Soit (\mathcal{P}) un problème de la forme :

$$(\mathcal{P}) : \begin{cases} \min_{\mathbf{u} \in \mathcal{U}} & f(\mathbf{u}) \text{ sous les contraintes} \\ & g_i(\mathbf{u}) \leq 0 \quad \forall i \in [|1, m|] \\ & h_j(\mathbf{u}) = 0 \quad \forall j \in [|1, r|] \end{cases}$$

Avec f, g_i, h_j à valeurs réelles de classe \mathcal{C}^1 . On note (\mathcal{Q}) son problème dual.

Soit un triplet $(\mathbf{u}, \boldsymbol{\alpha}, \boldsymbol{\beta})$ de $\mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^r$ qui vérifie les conditions suivantes (α_i est la i -ème composante de $\boldsymbol{\alpha}$, idem pour β_i):

- Admissibilité primale : $\forall i, g_i(\mathbf{u}) \leq 0$ et $\forall j, h_j(\mathbf{u}) = 0$
- Admissibilité duale : $\forall i, \alpha_i \geq 0$
- Complémentarité des contraintes : $\forall i, \alpha_i g_i(\mathbf{u}) = 0$
- Stationnarité : $\nabla f(\mathbf{u}) + \sum_{i=1}^m \alpha_i \nabla g_i(\mathbf{u}) + \sum_{j=1}^r \beta_j \nabla h_j(\mathbf{u}) = 0$

Alors \mathbf{u} est un point de minimisation du primal (\mathcal{P}) et $(\boldsymbol{\alpha}, \boldsymbol{\beta})$ est un point de maximisation du dual (\mathcal{Q}) .

On peut utiliser les conditions de Karush-Kuhn-Tucker sur le problème 3.1, dont le problème dual est le problème 3.2, qui nous donnent comme conditions d'écart complémentaires :

$$\begin{aligned} \forall i, \quad \alpha_i(y_i(\boldsymbol{\omega}^T \cdot \mathbf{x}_i + b) - 1) &= 0 \\ \forall i, \quad (C - \alpha_i) \max(0, (1 - y_i(\boldsymbol{\omega}^T \cdot \mathbf{x}_i + b))) &= 0 \end{aligned}$$

Ainsi on a trois cas de figure :

- $\alpha_i = 0$: dans ce cas la contrainte est satisfaite et l'observation \mathbf{x}_i est à l'extérieur de la zone d'indécision.
- $0 < \alpha_i < C$: dans ce cas \mathbf{x}_i est un vecteur de support.
- $\beta_i = 0$: dans ce cas $\alpha_i = C$ et $\max(0, (1 - y_i(\boldsymbol{\omega}^T \cdot \mathbf{x}_i + b))) > 0$ ce qui signifie que \mathbf{x}_i est du mauvais côté de la zone d'indécision.

Avec cette interprétation on peut définir les vecteurs de support comme étant les observations telles que $\alpha_i > 0$:

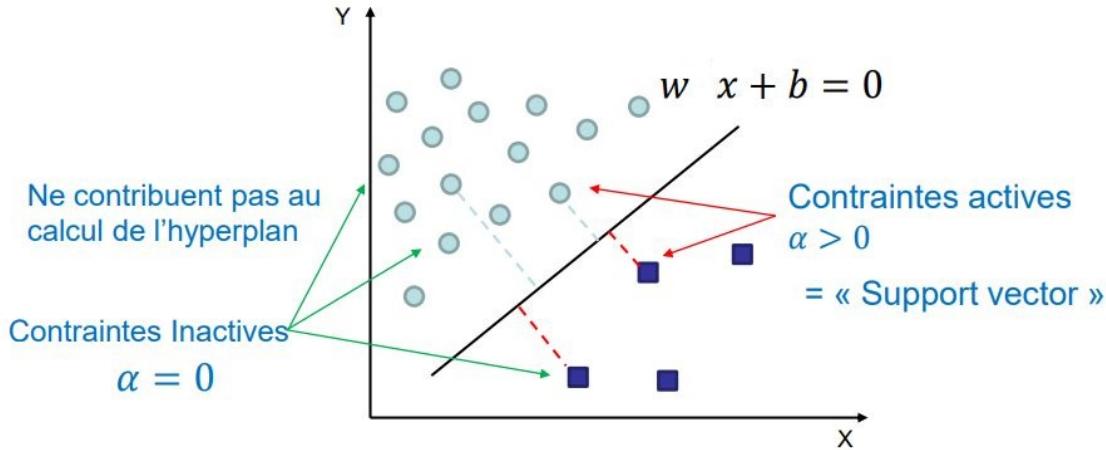


Figure 3.12: Interprétation des α_i dans la construction de l'hyperplan

3.3.6 Données non linéairement séparables

Nous avons vu comment séparer des données qui ne sont pas linéairement séparables à cause de quelques observations. Toutefois ce n'est pas la seule raison qui peut rendre des données non linéairement séparables. En effet il n'est pas rare de se retrouver dans un cas où séparer linéairement les données n'est pas possible à cause de leur forme. Dans de tels cas on peut alors modifier les données grâce à une transformation polynomiale ou polaire afin de pouvoir les séparer :

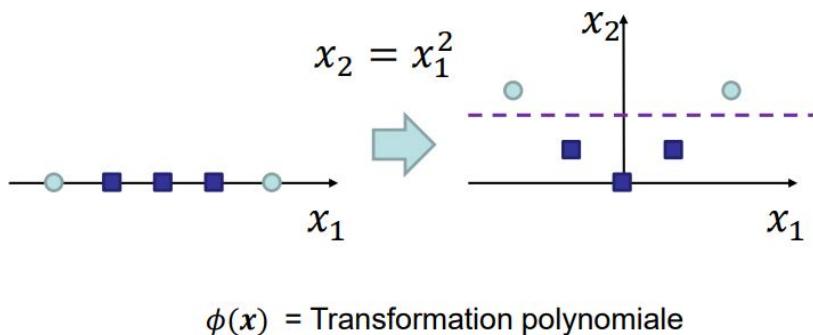


Figure 3.13: Exemple de transformation polynomiale

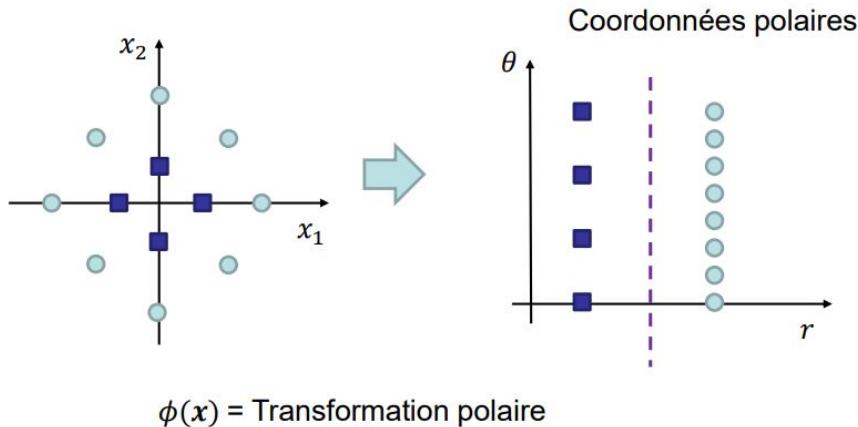


Figure 3.14: Exemple de transformation polaire

3.3.7 Les noyaux dans les SVM

Si on revient sur la formulation duale 3.2 du SVM, on peut utiliser ce que l'on appelle le **Kernel trick** pour pouvoir l'adapter aux données transformées, le problème devient alors :

$$\begin{aligned} & \arg \max_{\alpha \in \mathbb{R}} \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j) \\ & \text{tel que } \forall i, \quad 0 \leq \alpha_i \leq C \end{aligned}$$

Le noyau K est un produit scalaire dans l'espace transformé :

$$K(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j)$$

Il est uniquement nécessaire de connaître la similarité entre les données pour introduire la non linéarité dans le problème. L'utilisation de noyaux dans les SVM permet notamment d'introduire des mesures de similarités propres au domaine étudié et sans avoir à gérer la complexité de la transformation. De plus elle définit la fonction de classification à partir de noyaux "centrés" sur les vecteurs de support :

$$f(\mathbf{x}_i, \boldsymbol{\omega}) = b + \sum_{i=1}^n \alpha_i y_i K(\mathbf{x}_i, \mathbf{x}_j)$$

Voici quelques exemples de noyaux couramment utilisés :

Polynôme de degrés supérieurs à d : $K(\mathbf{x}, \mathbf{y}) = (\mathbf{x} \cdot \mathbf{y} + 1)^d$

Noyau gaussien : $K(\mathbf{x}, \mathbf{y}) = \exp\left(-\frac{(\mathbf{x} - \mathbf{y})^T(\mathbf{x} - \mathbf{y})}{2\sigma^2}\right)$

Intersection d'histogrammes : $K(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^n \min(\mathbf{x}_i, \mathbf{y}_i)$

3.3.8 Résumé concernant les SVM

Il s'agit d'obtenir une formulation optimale quadratique du problème de classification binaire. Le primal se présente sous la forme d'une optimisation d'un critère empirique à laquelle on ajoute la régularisation. Le dual permet quant-à-lui d'introduire la parcimonie et le kernel trick.

Les solutions s'expriment comme des combinaisons linéaires éparses de noyaux :

$$f(\mathbf{x}) = b + \sum_{i=1}^n \alpha_i y_i K(\mathbf{x}_i, \mathbf{x})$$

où $\alpha_i > 0$ seulement pour les vecteurs de support, sinon $\alpha_i = 0$.

En pratique, il faut régler :

- Le coefficient de régularisation C .
- Le type de noyau et ses paramètres (d pour noyau polynomial, σ pour un noyau gaussien etc...).
- Les paramètres de l'optimiseur.

3.4 Multiclasse

On se place désormais dans le cas d'un problème où il existe plus de deux classes différentes. Ici il est nécessaire d'effectuer de multiples hypothèses car il est possible que des classes soient plus rares que d'autres (on dit que les classes ne sont pas équilibrées) et le coût d'une erreur de classification peut être différent en fonction des classes. Par exemple, dans le cadre de classification d'animaux, on peut considérer qu'il est plus grave de classer un lion comme étant une gazelle que comme un tigre.

Face à des problèmes de ce type il existe deux stratégies majeures. La première consiste à optimiser un critère multi-hypothèse dans l'apprentissage, comme l'entropie dans les arbres de décision par exemple. Ou alors on peut décomposer le problème multiclasse en plusieurs problèmes binaires et ainsi utiliser un ensemble de classificateurs binaires.

Pour passer d'une classification binaire à N classes on dispose de plusieurs techniques, nous en aborderons ici deux : le *One vs One* et le *One vs Rest*.

3.4.1 Le One vs One

On apprend autant de classificateurs que de paires de classes, qui sont au nombre de $N(N - 1)/2$. Le choix de la classe se porte alors sur celle ayant le plus de votes, c'est-à-dire celle qui est donnée par le plus grand nombre de classificateurs. Cependant cette méthode a ses limites et, dans certains cas, des égalités surviennent et il est alors impossible de déterminer quelle classe choisir.

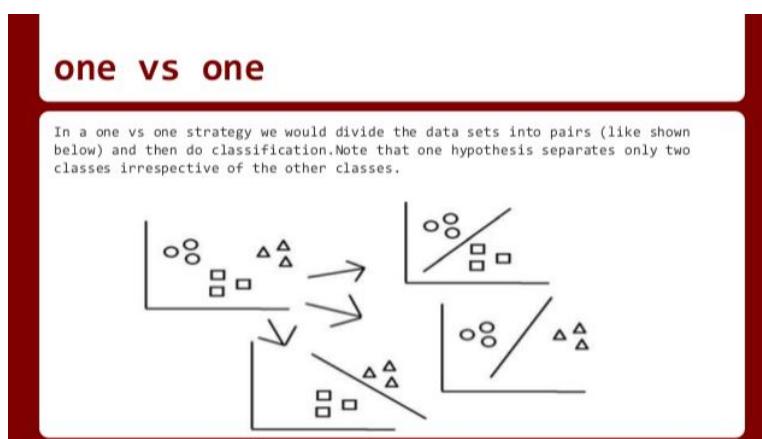


Figure 3.15: Illustration du principe *One vs One* [23]

3.4.2 Le One vs Rest

Avec cette approche on cherche à apprendre un classifieur par classe. Ainsi pour la classification on choisit la classe ayant le meilleur score. Toutefois on obtient un déséquilibre des données entre la classe cible et les autres.

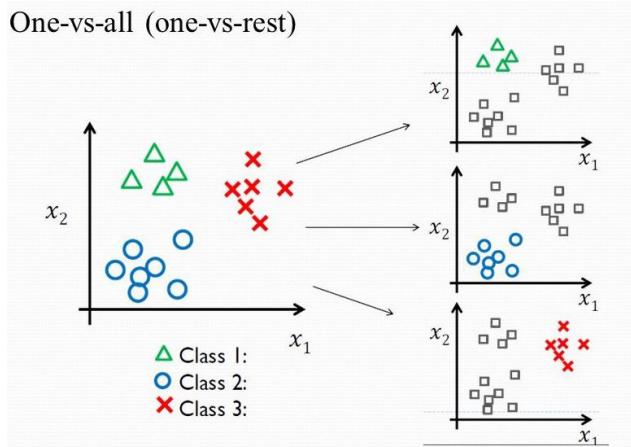


Figure 3.16: Illustration du principe *One vs Rest* [24]

3.4.3 Evaluation du multiclasse

Il existe plusieurs critères permettant d'estimer la qualité d'un classifieur multiclasse. Le taux d'erreur moyen et la matrice de confusion sont présentés en 1.3.3. Mais on peut définir un critère plus fin prenant en compte des coûts différents en fonction de l'erreur commise : le coût moyen.

$$\hat{\mathcal{R}} = \sum_{i=1}^N \sum_{j=1}^N \lambda(i, j) \text{conf}(i, j) p(j)$$

Avec $\lambda(i, j)$ le coût de décider i lorsque la vraie classe est j , $\text{conf}(i, j)$ la probabilité de décider i lorsque la vraie classe est j et enfin $p(j)$ la proportion d'exemple de la classe j dans les données de validation.

Chapter 4

Introduction aux réseaux de neurones

4.1 Le modèle du neurone

4.1.1 Motivation du modèle

La biologie moderne a montré que le cerveau humain était constitué d'environ 100 milliards de petites unités appelées neurones. Ces neurones assurent le fonctionnement du cerveau en s'envoyant mutuellement des signaux électriques. L'apprentissage automatique vise à reproduire la reflexion humaine en l'améliorant, ainsi il n'est pas incongru d'imaginer un modèle reprenant l'architecture d'un cerveau humain.

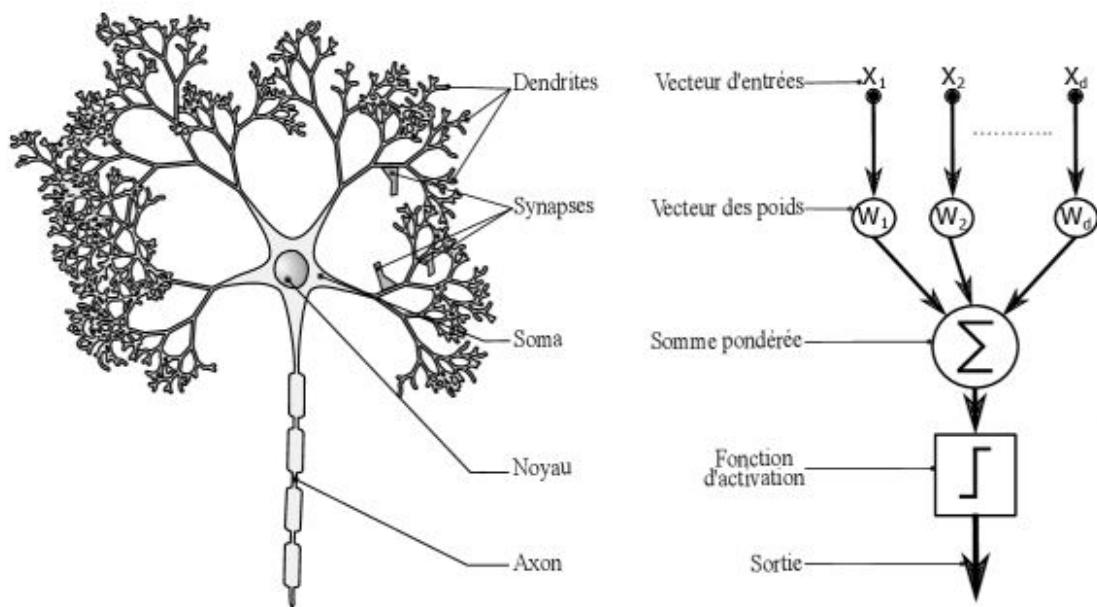


Figure 4.1: Illustration du parallèle entre le cerveau biologique et le modèle du neurone [25][26]

4.1.2 Passage du neurone au réseau

Bien sûr, d'un point de vue biologique, l'excitation du neurone n'est pas une simple somme pondérée de ses signaux entrants, la réalité est bien plus complexe et elle ne sert que d'inspiration au modèle informatique. Nous considérons un modèle simplifié où les neurones ont deux états. Le neurone est soit activé soit désactivé. Chaque neurone peut alors être assimilé à un hyperplan séparateur de l'espace des données.

On peut alors se demander quelle est la différence avec les SVM si on assimile un neurone à un hyperplan permettant de séparer les données. Cette distinction vient du fait que, dans certains cas, on ne peut pas séparer les données avec un seul hyperplan.

En pratique, dans le cas d'une classification binaire $y \in \{-1, 1\}$, pour savoir si un neurone est "activé" (i.e. s'il prédit l'exemple comme positif ou négatif), on entre une donnée $\mathbf{x} = [x_1, \dots, x_d]$ qui arrive en entrée du neurone, qui contient les poids $\omega_1, \dots, \omega_d$. Le neurone calcule alors la somme:

$$a = \sum_{i=1}^d \omega_i x_i = \boldsymbol{\omega}^T \cdot \mathbf{x}$$

La valeur de cette somme va déterminer si le neurone s'active ou non. Si $a > 0$ alors il prédit $y = 1$, sinon $y = -1$. On peut aisément interpréter les différents poids utilisés par le neurone. En effet une dimension i de la donnée telle que $\omega_i = 0$ indique que cet élément n'est pas pris en compte pendant la prédiction tandis que celles avec un poids positif sont décisives dans l'activation du neurone.

Il est aussi possible de choisir un **seuil d'activation** θ non nul, ce qui n'activerait le neurone que pour une valeur $a > \theta$. Pour ce faire on introduit un terme de biais b dans le calcul effectué par le neurone :

$$a = \boldsymbol{\omega}^T \cdot \mathbf{x} + b$$

Ainsi on peut définir formellement un neurone n :

$$\begin{aligned} n_{\boldsymbol{\omega}, b} : \quad \mathbb{R}^d &\rightarrow \mathbb{R} \\ \mathbf{x} &\mapsto \boldsymbol{\omega}^T \cdot \mathbf{x} + b \end{aligned}$$

Une fois cette définition posée on peut introduire la notion de **couche de ψ neurones** (notée c) qui est une séquence de ψ neurones prenant la même entrée et dont les ψ sorties sont regroupées en un vecteur :

$$\begin{aligned} c_{\Omega, \mathbf{b}} : \quad \mathbb{R}^d &\rightarrow \mathbb{R}^\psi \\ \mathbf{x} &\mapsto \begin{bmatrix} n_{\omega_1, b_1} \\ \dots \\ n_{\omega_\psi, b_\psi} \end{bmatrix} \end{aligned}$$

$\Omega \in \mathcal{M}_{\psi, d}(\mathbb{R})$ est la matrice de poids de l'ensemble des ψ neurones et $\mathbf{b} \in \mathbb{R}^\psi$ en est le vecteur de biais. On peut noter que la couche est linéaire : $c_{\Omega, \mathbf{b}}(\mathbf{x}) = \Omega \mathbf{x} + \mathbf{b}$

Cette linéarité permet de montrer que l'empilement de 2 couches de neurones revient à utiliser une seule couche :

$$\Omega'(\Omega \mathbf{x} + \mathbf{b}) + \mathbf{b}' = (\Omega' \Omega) \mathbf{x} + (\Omega' \mathbf{b} + \mathbf{b}')$$

Cependant, dans le modèle du neurone, la notion d'activation est une source de non linéarité :

$$\Omega' \text{activation}(\Omega \mathbf{x} + \mathbf{b}) + \mathbf{b}' \neq \text{activation}((\Omega' \Omega) \mathbf{x} + (\Omega' \mathbf{b} + \mathbf{b}'))$$

Il existe de nombreuses fonctions d'activation, la plus ancienne étant la fonction *signe*. Une condition sur les fonctions d'activation est qu'elles doivent être non linéaires.

C'est sous cette forme que sont définis les réseaux de neurones : il s'agit d'un empilement de couches de neurones en prenant l'activation en compte. La fonction d'activation la plus utilisée de nos jours est appelée *relu* et s'exprime ainsi :

$$\text{relu}(\mathbf{x}) = \max(\mathbf{x}, 0)$$

Avec ces notions on peut maintenant définir ce qu'est un réseau de neurones entièrement connecté **MLP** (multi layer perceptron en anglais) de profondeur Q est un empilement de Q couches de neurones :

$$\begin{aligned} \text{reseau}_\Omega : \quad \mathbb{R}^d &\rightarrow \mathbb{R}^{d'} \\ \mathbf{x} &\mapsto \Omega_Q(\text{relu}(\Omega_{Q-1}\text{relu}(\dots(\text{relu}(\Omega_1 \mathbf{x}))))) \end{aligned}$$

Ici $\Omega = \{\Omega_1, \dots, \Omega_Q\}$ représente Q matrices qui ont pour conditions que $\Omega_1 \in \mathcal{M}_{n, d}(\mathbb{R})$ avec n quelconque, $\Omega_Q \in \mathcal{M}_{d', n}(\mathbb{R})$ et $\Omega_q \in \mathcal{M}_{n, n}(\mathbb{R}) \forall q \in [2, Q-1]$. Les dimensions de ces matrices doivent néanmoins être cohérentes entre elles afin de pouvoir effectuer les calculs matriciels.

4.1.3 Les réseaux de neurones et les SVM

Les principes de base des réseaux de neurones sont semblables à ceux des SVM. En effet dans les deux cas on applique une fonction $f(\mathbf{x}, \boldsymbol{\omega})$ à la donnée et le signe de cette fonction donne la prédiction associée à cette donnée. La seule différence est l'expression de cette fonction f qui pour les SVM est de la forme :

$$f(\mathbf{x}, \boldsymbol{\omega}) = b + \boldsymbol{\omega}^T \cdot \mathbf{x}$$

Tandis que pour les MLP on a :

$$f(\mathbf{x}, \boldsymbol{\Omega}) = \Omega_Q(\text{activation}(\Omega_{Q-1}\text{activation}(\dots(\text{activation}(\Omega_1\mathbf{x})))))$$

4.1.4 Preuves de l'universalité des réseaux *relu*

La valeur absolue

Les réseaux de neurones *relu* sont assez universels et peuvent par exemple servir à calculer la valeur absolue pour $x \in \mathbb{R}$:

$$|x| = \text{relu}(x) + \text{relu}(-x) = \mathbf{1}_2^T \cdot \text{relu} \left(x \begin{bmatrix} 1 \\ -1 \end{bmatrix} \right)$$

Avec $\mathbf{1}_2 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$. Ici on utilise un réseau à deux couches dont la première contient deux neurones et la seconde un seul. Ainsi trois neurones permettent de calculer la valeur absolue d'un réel x .

La norme 1 pour $x \in \mathbb{R}^2$

De la même façon on peut appliquer les réseaux de neurones au calcul de la norme 1 pour $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ dans \mathbb{R}^2 :

$$\begin{aligned} \|\mathbf{x}\|_1 &= |x_1| + |x_2| = |[1, 0] \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}| + |[0, 1] \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}| \\ \|\mathbf{x}\|_1 &= \text{relu}([1, 0] \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}) + \text{relu}([-1, 0] \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}) + \text{relu}([0, 1] \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}) + \text{relu}([0, -1] \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}) \\ \|\mathbf{x}\|_1 &= \mathbf{1}_4^T \cdot \text{relu} \left(\begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix} \mathbf{x} \right) \end{aligned}$$

Avec $\mathbf{1}_4 = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$. Ainsi on utilise un réseau de neurone de profondeur deux avec 4 neurones dans la première couche et 1 dans la seconde.

La norme 1 pour $x \in \mathbb{R}^d$

On peut généraliser le résultat précédent en dimension d en suivant le même cheminement et en réorganisant les lignes de la matrices, ce que l'on peut faire sans soucis puisqu'une seule composante de la matrice est non nulle par ligne et que le produit matriciel final s'effectue avec le vecteur **1** de dimension d :

$$\|x\|_1 = \mathbf{1}_d^T \cdot \text{relu} \left(\begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots \\ \dots & \dots & \dots & \dots \\ 0 & \dots & 0 & 1 \\ -1 & 0 & \dots & 0 \\ 0 & -1 & 0 & \dots \\ \dots & \dots & \dots & \dots \\ 0 & \dots & 0 & -1 \end{pmatrix} x \right)$$

$$\|x\|_1 = \mathbf{1}_d^T \cdot \text{relu} \left(\begin{bmatrix} \mathbf{I}_d \\ -\mathbf{I}_d \end{bmatrix} x \right)$$

Où \mathbf{I}_d est la matrice identité de dimension d . On a donc toujours un réseau de profondeurs 2 avec $2d$ neurones dans la première couche et un dans la seconde.

La norme 1 avec biais

On peut même coder la distance en norme 1 à un point quelconque q en rajoutant simplement un terme de biais dans la première couche :

$$\|x - q\|_1 = |(1, 0) \cdot (x - q)| + |(0, 1) \cdot (x - q)|$$

$$\|x\|_1 = |(1, 0) \cdot x| + |(0, 1) \cdot x| + |(-1, 0) \cdot q| + |(0, -1) \cdot q|$$

$$\|x - q\|_1 = \mathbf{1}_d^T \cdot \text{relu} \left(\begin{bmatrix} \mathbf{I}_d \\ -\mathbf{I}_d \end{bmatrix} x + \begin{bmatrix} -q \\ q \end{bmatrix} \right)$$

Une fois encore on utilise une simple couche de $2d$ neurones en plus d'un neurone pour sommer.

Le pseudo-dirac

Pour montrer un peu plus la diversité des applications des réseaux de neurones *relu*, voici comment on peut définir une pseudo fonction de Dirac :

$$\delta_{\mathbf{q}}(\mathbf{x}) \approx \delta'_{\mathbf{q}}(\mathbf{x}) = \mathbf{1}_d^T \cdot \text{relu}(1 - \|\mathbf{x} - \mathbf{q}\|_1)$$

Cette fonction vérifie :

$$\begin{cases} \delta'_{\mathbf{q}}(\mathbf{q}) & > 0 \quad (\text{ici } \delta_{\mathbf{q}}(\mathbf{q}) = 1) \\ \delta'_{\mathbf{q}}(\mathbf{x}) & = 0 \quad \forall \mathbf{x} / \|\mathbf{x} - \mathbf{q}\|_1 > 1 \end{cases}$$

4.2 Apprentissage des réseaux *relu*

4.2.1 Une erreur à éviter : apprendre la base de données par cœur

Si l'on dispose d'une base de données $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i \in [|1, n|]}$ telle que $\forall i \in [|1, n|]$, $\mathbf{x} \in \mathbb{R}^d$ et $y_i \in \{-1, 1\}$, alors il suffit de $2 \times n \times d + n + 1$ neurones pour apprendre la base de données "par cœur" avec :

$$f(\mathbf{x}, \Omega) = \sum_{i=1}^n y_i \text{relu}(1 - \|\mathbf{x} - \mathbf{x}_i\|_1)$$

$$f(\mathbf{x}, \Omega) = \begin{bmatrix} y_1 \\ \dots \\ y_n \end{bmatrix}^T \text{relu}(-\mathbf{I} \text{relu} \left(\begin{bmatrix} \mathbf{I} \\ \dots \\ \mathbf{I} \\ -\mathbf{I} \\ \dots \\ -\mathbf{I} \end{bmatrix} \mathbf{x} + \begin{bmatrix} -\mathbf{x}_1 \\ \dots \\ -\mathbf{x}_n \\ \mathbf{x}_1 \\ \dots \\ \mathbf{x}_n \end{bmatrix} \right) + \mathbf{1})$$

Si l'on se trouve dans un problème de classification de points colorés par exemple, utiliser un seul neurone reviendrait à chercher un hyperplan séparateur $f(\mathbf{x}, \omega) = b + \omega^T \cdot \mathbf{x}$ mais il se peut qu'il n'existe pas de séparation.

Avec un réseau de neurones on utilise $f(\mathbf{x}, \Omega) = \Omega_Q \text{relu}(\Omega_{Q-1} \text{relu}(\dots \text{relu}(\Omega_1 \mathbf{x})))$ et on sait qu'avec $O(ND)$ neurones et 3 couches, on peut tout apprendre. Cependant, nous avons déjà vu dans les chapitres précédents que tout apprendre n'était pas nécessairement une bonne chose et que, dans la plupart des cas, il s'agissait même de quelque chose de négatif.

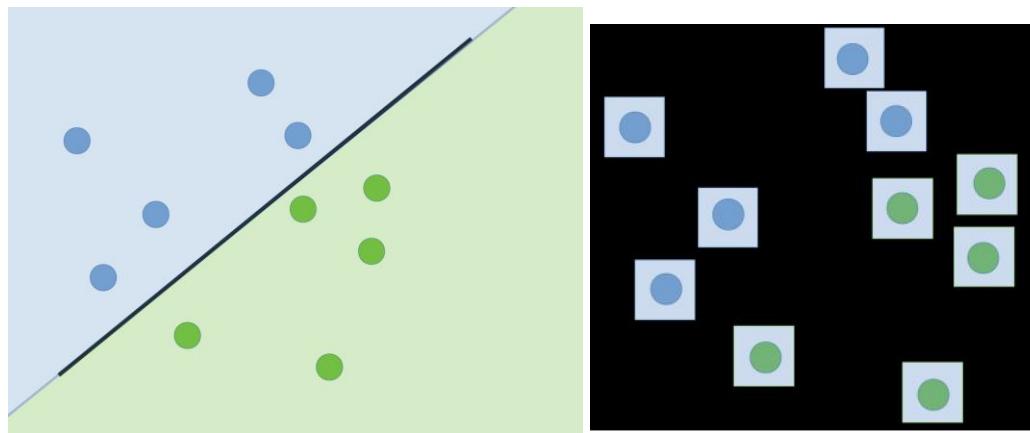


Figure 4.2: Comparaison des prédicteurs SVM et d'un réseau de neurones qui aurait appris la base d'apprentissage

On voit sur la Figure 4.2 qu'un modèle de réseau de neurones ayant appris la base d'apprentissage par cœur ne sert à rien puisqu'il ne prend aucune décision en dehors de cette base. Il ne fait qu'encoder les données dont nous disposons déjà et ne donne aucune information sur les autres points de l'espace des données. Or la partie qui importe pour un programme de Machine Learning est sa capacité à traiter des données inconnues.

4.2.2 L'algorithme perceptron

Pour éviter de rencontrer le problème précédent, on va changer la méthode d'entraînement du modèle des réseaux de neurones en utilisant l'algorithme du perceptron.

Il s'agit d'un algorithme classique d'apprentissage pour le modèle des **réseaux de neurones simple composés d'une seule couche de perceptron**. En pratique, nous utilisons pour les réseaux de neurones plus profonds la descente de gradient stochastique (présentée en section 4.3). L'algorithme perceptron est un programme dans le même style que l'approche des k plus proches voisins puisqu'il s'agit d'un algorithme certes simple, mais qui obtient de très bons résultats pour certains types de problèmes.

Algorithm 5 PERCEPTRONTRAIN(\mathbf{D} , $MaxIter$)

```

1:  $w_d \leftarrow 0$ , for all  $d = 1 \dots D$                                 // initialize weights
2:  $b \leftarrow 0$                                                        // initialize bias
3: for  $iter = 1 \dots MaxIter$  do
4:   for all  $(x,y) \in \mathbf{D}$  do
5:      $a \leftarrow \sum_{d=1}^D w_d x_d + b$                                // compute activation for this example
6:     if  $ya \leq 0$  then
7:        $w_d \leftarrow w_d + yx_d$ , for all  $d = 1 \dots D$            // update weights
8:        $b \leftarrow b + y$                                          // update bias
9:     end if
10:   end for
11: end for
12: return  $w_0, w_1, \dots, w_D, b$ 

```

Algorithm 6 PERCEPTRONTEST($w_0, w_1, \dots, w_D, b, \hat{\mathbf{x}}$)

```

1:  $a \leftarrow \sum_{d=1}^D w_d \hat{x}_d + b$                                // compute activation for the test example
2: return  $\text{SIGN}(a)$ 

```

Figure 4.3: L'algorithme du perceptron [27]

Avant d'aller plus loin dans l'interprétation de l'algorithme, commençons par introduire toutes les notations utilisées. Tout d'abord les w_i correspondent aux poids qui seront affectés aux composantes des données, b correspond au biais, a est le résultat de la prédiction et y est la classe réelle de la donnée \mathbf{x} , il prend en entrée la base d'entraînement ici notée \mathbf{D} ainsi que $MaxIter$ qui correspond au nombre de parcours de la base de données, plus ce nombre sera élevé, meilleurs seront les résultats du modèle **sur les données d'entraînement**. A priori il ne suffit donc pas de prendre $MaxIter$ très grand pour obtenir un excellent prédicteur.

Arrêtons-nous maintenant sur le fonctionnement de cet algorithme. Il s'agit d'un algorithme **d'apprentissage incrémental** (ou apprentissage **online** en anglais), ce qui signifie qu'il effectue une ou plusieurs opérations à chaque nouvelle observation qui lui est fournie, à mettre en opposition d'un apprentissage dit **hors-ligne** qui opère sur toute la base d'entraînement en même temps.

De plus cet algorithme est dirigé par ses erreurs, c'est-à-dire que, tant que les résultats obtenus sont corrects, il ne modifie pas ses paramètres. Ces derniers correspondent aux poids w_1, w_2, \dots, w_D des différentes composantes des données et le biais b qui permet de "recentrer" la prédiction. L'algorithme parcourt la base de données \mathbf{D} $MaxIter$ fois et, pour chaque donnée \mathbf{x} , calcule sa prédiction a avant de vérifier que celle-ci est du même signe (et donc correcte puisqu'on se trouve dans un problème de classification binaire) que la véritable classe y de \mathbf{x} . Si la prédiction est bonne alors on ne change rien, si au contraire elle est erronée alors on modifie les poids et le biais afin que la prédiction a soit plus proche de y .

En réalité la Figure 4.3 ne présente qu'un cas particulier d'un algorithme plus général qui s'appuie sur la descente de gradient. En effet, on cherche en réalité à minimiser le risque empirique et son gradient nous indique la direction à prendre pour atteindre ce but. Ainsi pour chaque observation $(\mathbf{x}_i, y_i) \in \mathcal{D}$ on actualise les poids en utilisant ce gradient :

$$\begin{aligned} \forall j \in [|1, d|], \quad \omega_j &\leftarrow \omega_j - \eta \frac{\partial L(f(\mathbf{x}_i, \boldsymbol{\omega}), y_i)}{\partial \omega_j} \\ b &\leftarrow b - \eta \frac{\partial L(f(\mathbf{x}_i, \boldsymbol{\omega}), y_i)}{\partial b} \end{aligned} \tag{4.1}$$

Ici $\eta > 0$ est un hyperparamètre qui est le pas de l'algorithme du gradient est appelé **vitesse d'apprentissage**. Toutefois il faut le choisir soigneusement car si cet hyperparamètre est trop grand, l'algorithme pourrait ne pas réussir à affiner sa solution tandis que s'il est trop petit alors la convergence sera très lente. Dans le cas de la classification binaire on utilise souvent la fonction de coût connue sous le nom de **critère du perceptron** :

$$L(f(\mathbf{x}_i, \boldsymbol{\omega}), y_i) = \max(0, -y_i(\boldsymbol{\omega}^T \cdot \mathbf{x}_i))$$

Avec cette fonction de coût la règle d'actualisation 4.1 devient :

$$\frac{\partial L}{\partial \omega_j} \leftarrow \begin{cases} 0 & \text{si } y_i(\boldsymbol{\omega}^T \cdot \mathbf{x}_i) > 0 \\ -y_i(\mathbf{x}_i)_j & \text{sinon} \end{cases}$$

La convergence de l'algorithme du perceptron a été démontrée en 1962 par l'ukrainien Albert Novikoff sous la forme d'un théorème stipulant que, pour un jeu de n observations $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i \in [|1, n|]}$ et $D, \gamma \in \mathbb{R}_+^*$, si on a :

- $\forall i \in [|1, n|], \|\mathbf{x}_i\|_2 \leq D$
- il existe $\mathbf{u} \in \mathbb{R}^d$ tel que
 - $\|\mathbf{u}\|_2 = 1$
 - $\forall i \in [|1, n|], y_i(\mathbf{u}^T \cdot \mathbf{x}_i + b) \geq \gamma$

alors l'algorithme du perceptron converge en au plus $\left(\frac{D}{\gamma}\right)^2$ étapes.

On peut noter qu'il est aussi possible d'utiliser l'algorithme du perceptron dans le cadre d'un problème de régression, en général avec la fonction de coût :

$$L(f(\mathbf{x}_i, \boldsymbol{\omega}), y_i) = \frac{1}{2}(y_i - f(\mathbf{x}_i, \boldsymbol{\omega}))^2$$

Ainsi la règle d'actualisation 4.1 devient :

$$\omega_j \leftarrow \omega_j - \eta(f(\mathbf{x}_i, \boldsymbol{\omega}) - y_i)(\mathbf{x}_i)_j$$

4.2.3 Un nouveau paradigme

Nous avons déjà vu précédemment que la qualité d'un algorithme d'apprentissage automatique était évaluée sur un jeu de données de validation. De plus nous avons vu que pour obtenir de bonnes performances sur ces données il fallait que le modèle n'ait ni trop peu ni trop de paramètres. En effet dans le premier cas on ne disposera pas de ressources suffisantes pour pouvoir apprendre correctement tandis que dans le second on ne ferait qu'apprendre la base d'apprentissage sans pouvoir généraliser aux autres données inconnues.

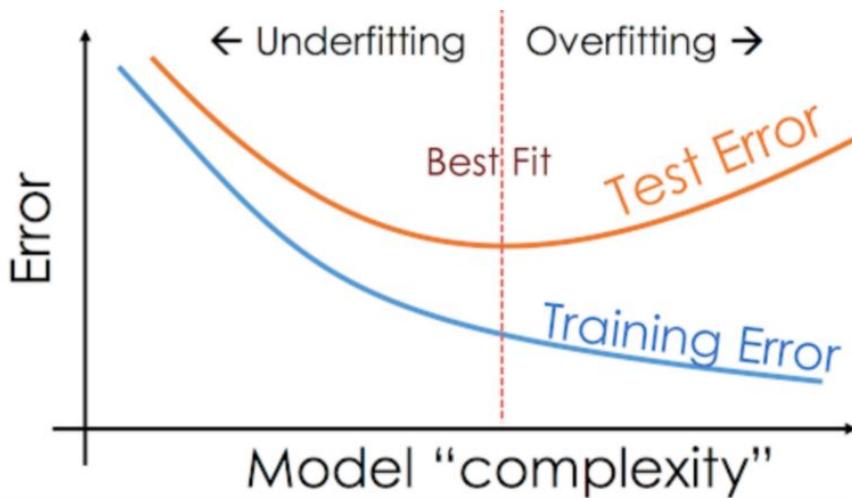


Figure 4.4: L'ancien paradigme des réseaux de neurones [28]

Toutefois un phénomène de "double descente" a été découvert tout récemment par Mikhail Belkin et al. dans une publication qui date seulement de décembre 2018.[29] Ce phénomène est illustré par la Figure 4.5 :

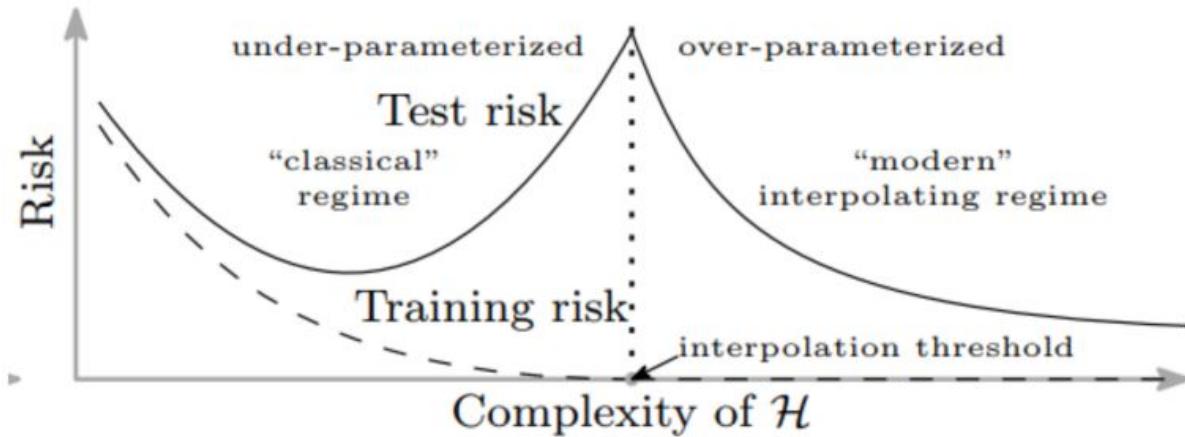


Figure 4.5: Le nouveau paradigme des réseaux de neurones [29]

Cette "double descente" montre qu'il est toujours préférable de prendre le plus gros réseau possible pour obtenir de meilleurs résultats. L'aspect positif de cette découverte est que l'on sait maintenant que l'on peut obtenir des modèles de plus en plus performants. Cependant elle montre aussi que la puissance de calcul dont on dispose prime sur l'ingéniosité et l'élegance du modèle créé, ce qui peut paraître injuste.

L'origine de la double descente

La question qui vient après cette découverte est : pourquoi observe-t-on une double descente ? Il semblerait qu'en réalité, quand on apprend un réseau, on apprendrait en réalité un ensemble de sous-réseaux dont la complexité s'adapterait au problème.

Si on se place dans un cas où l'on utilise la méthode du bagging développée au chapitre 2, c'est-à-dire qu'à partir de $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i \in [|1, n|]}$ on forme K prédicteurs f_1, \dots, f_K . Pour la phase de test il est possible de fusionner les différents modèles et de tendre vers les performances d'un modèle moyen f^* qui, attention, n'est pas forcément optimal.

La complexité n'augmente pas avec le nombre de modèles mais seulement avec leur capacité. Ainsi créer plein de modèles équivalents n'augmente pas l'overfitting.

Dans l'ancien paradigme on cherchait à créer K réseaux de Q neurones, ce qui n'est pas la même chose que de créer un seul réseau de KQ neurones. Ici K ne crée pas d'overfitting et la qualité du modèle reposait grandement sur la valeur sélectionnée pour Q : si ce dernier était trop petit, on apprenait pas, et s'il était trop grand on entrait dans un état de sur-apprentissage.

Avec la découverte de la double descente est apparu un nouveau paradigme qui assure qu'un réseau de H neurones va se décomposer naturellement en K réseaux de $\frac{H}{K}$ neurones. Cette taille de réseau serait en plus adaptée au problème traité.

Il faut néanmoins garder à l'esprit qu'il s'agit d'un phénomène que l'on *observe* dans un certain nombre d'expériences dans un domaine borné. Il ne s'agit pas d'une vérité générale. Cette spécificité vient de la façon d'entraîner les réseaux de neurones : la descente de gradient stochastique.

4.3 La descente de gradient stochastique

4.3.1 La descente de gradient

Soit F une fonction dérivable de \mathbb{R}^d dans \mathbb{R} , le gradient de F en un point \mathbf{u} noté $\nabla F(\mathbf{u})$ est la fonction de \mathbb{R}^d dans \mathbb{R}^d telle que :

$$\forall \mathbf{u}, \mathbf{h} \in \mathbb{R}^d, F(\mathbf{u} + \mathbf{h}) = F(\mathbf{u}) + (\nabla F(\mathbf{u}))^T \cdot \mathbf{h} + o(\mathbf{h})$$

Il ressort de cette définition que si $\nabla F(\mathbf{u}) \neq 0$ alors il existe $\lambda > 0$ tel que $F(\mathbf{u} - \lambda \nabla F(\mathbf{u})) < F(\mathbf{u})$ (il suffit de remplacer \mathbf{h} par $-\lambda \nabla F(\mathbf{u})$). À partir de ce constat on peut définir l'algorithme de la descente de gradient qui permet de trouver un point \mathbf{u}^* tel que $\nabla F(\mathbf{u}^*) = 0$ et d'ainsi trouver un minimum local de la fonction F , voire d'un minimum global si F est convexe. Étant donnés un pas $\lambda > 0$ et une tolérance $\epsilon > 0$, cet algorithme se résume à :

Choisir $\mathbf{u} \in \mathbb{R}^d$

Tant que $\|\nabla F(\mathbf{u})\|_2^2 \geq \epsilon$:

$$\mathbf{u} \leftarrow \mathbf{u} - \lambda \nabla F(\mathbf{u})$$

Retourner \mathbf{u}

L'hyperparamètre λ a une importance capitale dans cet algorithme car c'est lui qui va déterminer la vitesse de convergence de l'algorithme. S'il est choisi trop petit alors l'algorithme convergera très lentement, cependant il ne suffit pas de le prendre trop grand non plus car l'actualisation de \mathbf{u} ne serait alors pas assez fine pour converger vers le minimum et l'algorithme pourrait même diverger.

Pour palier à ce problème on peut utiliser un pas adaptatif qui sera de plus en plus petit à mesure que l'on s'approche du minimum. Une deuxième version de l'algorithme du gradient peut alors être :

Input : $F, \mathbf{u}_0, \epsilon$

1. $\mathbf{u} = \mathbf{u}_0$
2. Calculer $\nabla F(\mathbf{u})$
3. Si $\|\nabla F(\mathbf{u})\|_2^2 < \epsilon$ alors retourner \mathbf{u}
4. $\lambda = 1$
5. Tant que $F(\mathbf{u} - \lambda \nabla F(\mathbf{u})) \geq F(\mathbf{u})$ faire $\lambda = 0.5\lambda$
6. $\mathbf{u} = \mathbf{u} - \lambda \nabla F(\mathbf{u})$
7. Retour à l'étape 2

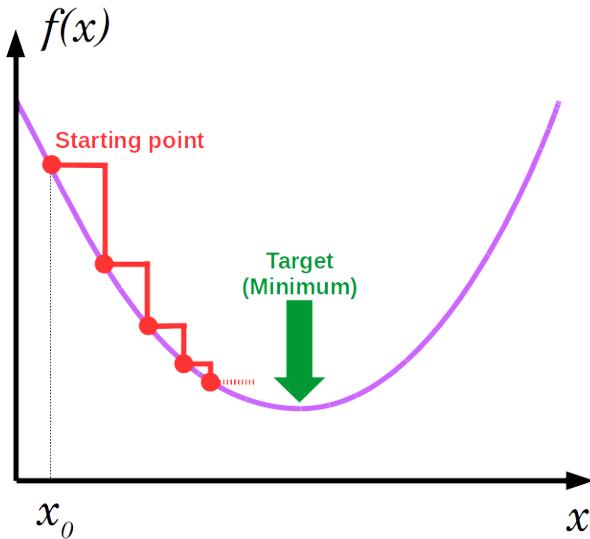


Figure 4.6: Principe de l'algorithme de la descente de gradient [30]

Dans l'application de cet algorithme à l'apprentissage on remplace la variable \mathbf{u} par les poids ω du réseau. En ce qui concerne la fonctionnelle F on la remplace par une fonction de coût $L(f(\omega))$ de la forme :

$$L(f(\omega)) = \sum_{i=1}^n l(f(\mathbf{x}_i, \omega), y_i)$$

Pour que l'algorithme de la descente de gradient puisse fonctionner, l doit être une fonction assez lisse. De plus l doit avoir une valeur proche de 0 si $y_i f(\mathbf{x}_i, \omega)$ est grand (les deux facteurs sont de même signe donc la prédiction est bonne) et au contraire doit fortement pénaliser les cas où $y_i f(\mathbf{x}_i, \omega)$ est petit. On peut par exemple utiliser la fonction *hinge loss* :

$$l(f(\mathbf{x}_i, \omega), y_i) = \text{relu}(1 - y_i f(\mathbf{x}_i, \omega))$$

4.3.2 Les limites de la descente de gradient

Le problème majeur de l'algorithme de la descente de gradient est la quantité de calculs nécessaires à sa réalisation. En effet si l'on dispose d'une base de données de grande taille, comme $n = 1000000$ par exemple, alors il est nécessaire d'appliquer f , elle-même constituée de plusieurs couches, à 1000000 de points et ce à chaque itération de l'algorithme. Ainsi cette première version, bien que valable mathématiquement et efficace pour des bases de données de tailles modestes, nécessite une puissance de calcul bien trop importante pour être utilisée telle quelle.

4.3.3 L'algorithme du gradient stochastique

Il s'agit d'une variante de l'algorithme du gradient visant à résoudre le problème de coût calculatoire prohibitif sur les bases de données conséquentes. Toutefois cette amélioration n'a pu être montrée mathématiquement que dans le cas où $L(f(\omega)) = \sum_{i=1}^n l_i(f(\omega))$ est convexe (où $l_i(f(\omega)) = l(f(\mathbf{x}_i, \omega), y_i)$). Pour ce faire on va bien sûr reprendre le principe d'une descente de gradient mais sur une sous somme des l_i tirée aléatoirement. Voici le pseudo-code de l'algorithme du gradient stochastique avec un pas adaptatif λ et une tolérance ϵ :

Choisir $\mathbf{u} \in \mathbb{R}^d$

Tant que $\|\nabla F(\mathbf{u})\|_2^2 \geq \epsilon$:

 Choisir i aléatoirement parmi $\{1, \dots, n\}$

$\lambda = 1$

 Tant que $F(\mathbf{u} - \lambda \nabla F(\mathbf{u})) \geq F(\mathbf{u})$ faire $\lambda = 0.5\lambda$

$\mathbf{u} \leftarrow \mathbf{u} - \lambda \nabla F(\mathbf{u})$

Retourner \mathbf{u}

4.3.4 La méthode mini Batch SGD

En pratique, en Python, l'algorithme présenté ci-dessus n'est qu'un cas particulier de la méthode appelée **mini Batch SGD** dans laquelle la base de données est découpée en sous-ensembles appelés *Batch* sur lesquels on calcule la fonction de coût partielle *partial_L*. On appelle une **époque** une période pendant laquelle toute la base de données a été utilisée dans les calculs de *partial_L*. Une fois une époque passée, on change les Batchs afin d'ajouter de la diversité dans chacun d'eux. L'algorithme du gradient stochastique n'est en fait qu'un algorithme mini Batch SGD dans lequel les Batchs sont de taille 1. L'actualisation des poids dans le cas de la fonction *hinge loss* devient alors :

$$\begin{aligned} \text{partial_L}(f(\omega)) &= \sum_{i \in \text{Batch}} \text{relu}(1 - y_i f(\mathbf{x}_i, \omega)) \\ \omega &= \omega - \lambda_{iter} \nabla \text{partial_L}(f(\omega)) \end{aligned}$$

Où λ_{iter} est le pas adaptatif de l'algorithme.

4.4 Le réseau neuronal convolutif

Dans cette section nous allons traiter d'une adaptation des réseaux de neurones au deep learning grâce au modèle du réseau neuronal convolutif.

Nous avons jusqu'à présent traité des données vectorielles $\mathbf{x} \in \mathbb{R}^d$. Toutefois il est possible d'appliquer les réseaux de neurones à tout type de données à condition de pouvoir les transformer en vecteurs.

Par exemple on pourrait être intéressé par des données sous la forme d'images. On considère que les images sont composées de trois matrices de dimensions $H \times L$ (hauteur \times largeur) dans le cas général : $X_{r,g,b} = [X_r, X_g, X_b]$ avec $X_r \in \mathcal{M}_{H,L}(\mathbb{R})$, $X_g \in \mathcal{M}_{H,L}(\mathbb{R})$ et $X_b \in \mathcal{M}_{H,L}(\mathbb{R})$. On considère alors les données à traiter sous la forme $\mathbf{x} \in \mathbb{R}^{3 \times H \times L}$.

On peut généraliser ce formalisme en posant que $X \in \mathbb{R}^{C \times H \times L}$ avec $C = 3$ si l'image est en format RGB (en couleur) ou encore $C = 1$ si elle est en niveau de gris. Le deep learning permet d'automatiser toutes les étapes de l'apprentissage automatique :

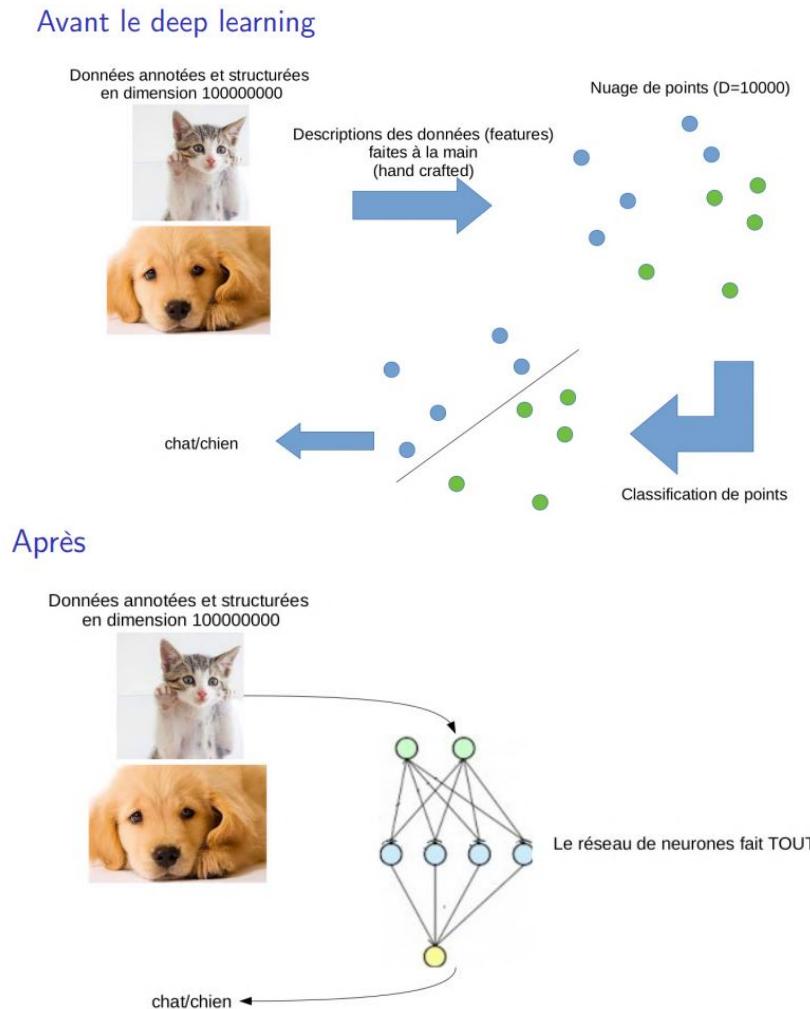


Figure 4.7: Comparaison de la démarche du machine learning avant et après le deep learning

4.4.1 Le neurone convolutif

Soit une image entrée $I \in \mathbb{R}^{C \times H \times L}$. On peut considérer la convolution de I avec un noyau $K \in \mathbb{R}^{C \times (2\delta_H+1) \times (2\delta_L+1)}$, les dimensions de K doivent être impaires pour que le noyau ait une case "centrale" comme nous l'expliquerons après. Cette convolution sera notée $I \star K \in \mathbb{R}^{H \times L}$ et définie par :

$$(I \star K)_{h \in [|1, H|], l \in [|1, L|]} = \sum_{c=1}^C \sum_{\alpha=0}^{2\delta_H} \sum_{\beta=0}^{2\delta_L} I_{c, h-\delta_H+\alpha, l-\delta_L+\beta} \times K_{c, \alpha, \beta}$$

Comme pour les neurones "basiques", on pourra considérer un groupe de Q neurones convolutifs K_1, \dots, K_Q dont on regroupe les sorties en une nouvelle image dans $\mathbb{R}^{Q \times H \times L}$. Pour illustrer cette définition mathématique qui peut sembler obscure, voici un schéma permettant de comprendre le principe de manière plus visuelle :

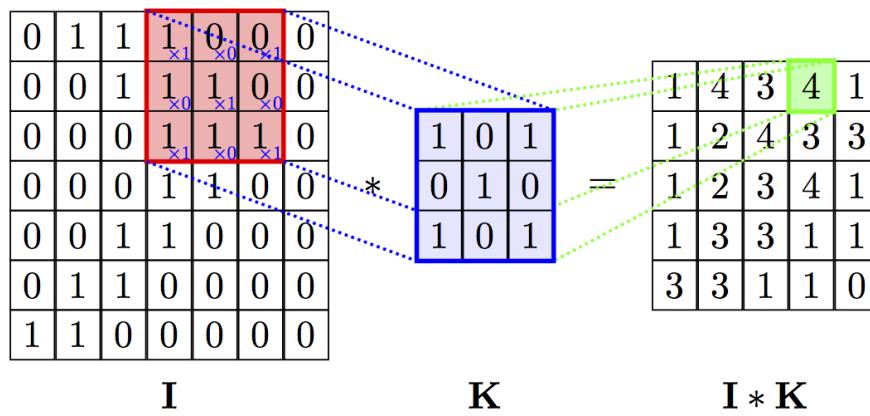


Figure 4.8: Schéma représentant le principe du neurone convolutif [39]

Pour chaque pixel (h, l) de l'image I , on va "placer" la case centrale du noyau sur ce pixel puis pour chaque case de K on va faire le produit de la valeur de cette case avec celle qui lui est superposée dans I puis en faire la somme. Une fois ce résultat obtenu on le place dans la case (h, l) du résultat de la convolution. On peut se dire que cette définition pose problème pour les pixels situés aux extrémités de l'image, mais on se contente de considérer que les cases du noyau qui sont "en dehors" de l'image sont superposées à du noir et donc à la valeur 0.

Comparons maintenant cette méthode avec les réseaux de neurones classiques. En effet si l'on considère une valeur de $I \star K$, typiquement :

$$(I \star K)_{h \in [|1, H|], l \in [|1, L|]} = \sum_{c=1}^C \sum_{\alpha=0}^{2\delta_H} \sum_{\beta=0}^{2\delta_L} I_{c, h-\delta_H+\alpha, l-\delta_L+\beta} \times K_{c, \alpha, \beta}$$

On voit que cette valeur peut tout à fait se coder avec un neurone classique $I \cdot K$ avec :

$$\mathcal{K}_{i,j} = \begin{cases} K_{i-h+\delta_H, j-l+\delta_L} & \text{si } h - \delta_H \leq i \leq h + \delta_H \\ & \quad l - \delta_L \leq j \leq l + \delta_L \\ 0 & \text{sinon} \end{cases}$$

Cependant il existe une différence majeure entre ces deux approches : le nombre de paramètres nécessaires à la génération de $I \star K$. Pour la convolution on a $O(\delta_H \times \delta_L)$ paramètres contre $O(H^2 \times L^2)$ en utilisant une couche classique (perceptron). Pour donner un exemple concret on cela donne 9 paramètres contre 4294967296 pour une image de taille 256x256 et un noyau 3x3.

Un autre point essentiel est qu'une couche de convolution fait ressortir l'information locale.

Nous pouvons considérer qu'un réseau convolutionnel est similaire à un réseau Fully connected (Perceptron), mais avec un apriori fort sur ses poids. Cette apriori consiste à dire que les poids d'une unité couche caché doivent être identiques aux poids de son voisin, mais décalés dans l'espace. L'apriori indique également que les poids doivent être nuls, sauf dans le petit champ proche du pixel d'intérêt. Globalement, on peut considérer que l'utilisation de la convolution introduit une distribution de probabilité sur les paramètres d'une couche. Ce priors dit que la couche ne doit apprendre que des interactions locales et est équivariante à la translation.

Bien sûr, la mise en œuvre d'un réseau convolutif comme un réseau entièrement codé par cette apriori serait extrêmement coûteuse en termes de calcul, c'est pourquoi on utilise des corrélations croisé pour coder ces couches de convolution.

Une autre idée est que nous ne devrions comparer les modèles convolutifs à d'autres modèles non convolutifs tel que les fully connected (Pereceptron). Les modèles qui n'utilisent pas la convolution seraient capables d'apprendre même si l'on mélange tous les pixels de l'image. Les modèles convolutifs quant à eux travaillent de manière local et base leur décision sur l'information locale.

4.4.2 Le pooling

On prend toujours une image entrée $I \in \mathbb{R}^{C \times H \times L}$. Le pooling consiste en la création de $pool(I) \in \mathbb{R}^{C \times \frac{H}{2} \times \frac{L}{2}}$ tel que :

$$pool(I)_{c \in [|1, C|], h \in [|1, H|], l \in [|1, L|]} = \max_{\alpha \in \{2h, 2h+1\}, \beta \in \{2l, 2l+1\}} I_{c, \alpha, \beta}$$

Ici l'idée est de réduire la taille de l'image par 4 en sélectionnant pour chaque pixel et chaque canal de l'image résultat la valeur maximale parmi 4 pixels adjacents.

4.4.3 Les réseaux

Les réseaux de neurones convolutifs sont quant à eux issue de la concaténation d'opération de convolution suivie de non-linéarité et d'opération de pooling qui concentre l'information. Une fois la représentation convolutif fini, vient souvent des fully connected qui permette de transformer l'image en un vecteur de probabilité.

L'acte de créer une architecture consiste à décider quelle couche mettre ou cette tâche est assez dure et sera vue plus en détails en troisième année. Nous allons expliquer LeNet-5 qui est l'un des réseaux les plus simples.

L'architecture de LeNet-5 (Figure 4.10) se compose de deux couches convolutives et qui sont suivie chacune d'une fonction d'activation et d'une couche de pooling, après ces deux couches il y a trois couches fully connected (Perceptron) suivie chacune d'une fonction d'activation enfin d'un softmax.

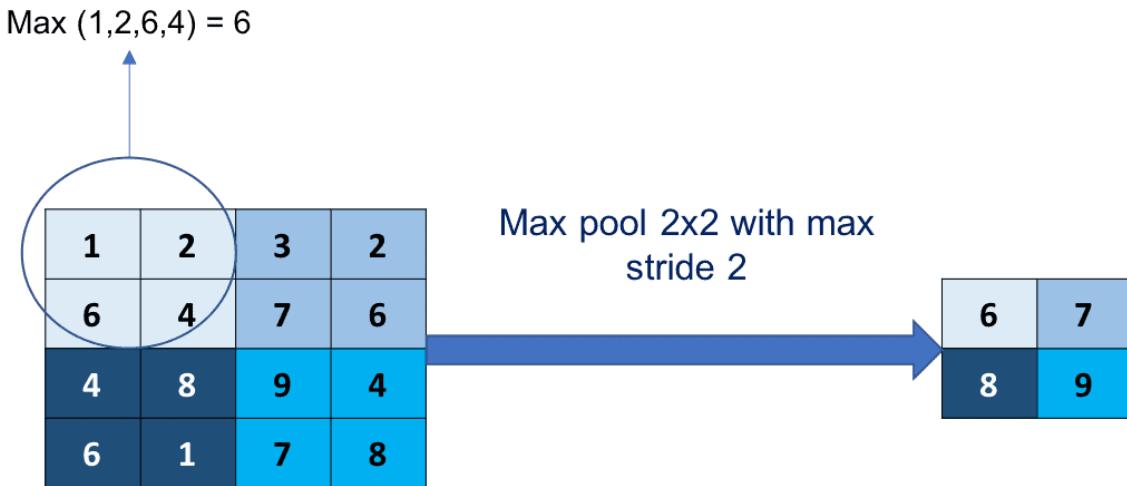


Figure 4.9: Schéma de principe du pooling [38]

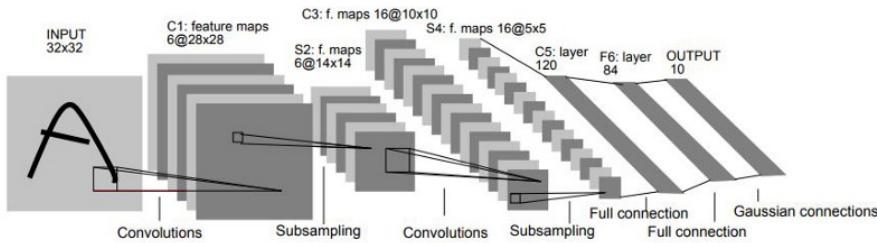


Figure 4.10: Schéma de LeNet-5 [37]

L'entrée de LeNet-5 est une image en niveaux de gris 32×32 (de la base de donnée MNIST) qui passe par la première couche (**C1**) convulsive avec 6 cartes de caractéristiques (feature maps) ou filtres ayant une taille de 5×5 et un pas de un. Les dimensions de l'image passent de $32 \times 32 \times 1$ à $28 \times 28 \times 6$. Ainsi à la fin de la première couche une image est représentée par 6 images.

Ensuite, le LeNet-5 applique une couche de mise de pooling par la moyenne (**S2**) avec une taille de filtre 2×2 et un stride de deux. Les dimensions de l'image résultante seront réduites à $14 \times 14 \times 6$.

Ensuite, il y a une deuxième couche convulsive (**C3**) avec 16 cartes de caractéristiques et un filtre de taille 5×5 et un stride de 1. Les dimensions de l'image résultante seront de $10 \times 10 \times 16$.

La quatrième couche (**S4**) est à nouveau une couche de pooling par la moyenne avec une taille de filtre 2×2 et un stride de 2. Cette couche est la même que la deuxième couche sauf qu'elle a 16 cartes de caractéristiques, donc la sortie sera réduite à $5 \times 5 \times 16$. Enfin, on applique un reshape pour transformer ce tenseur de $5 \times 5 \times 16$ en vecteur de taille 400.

Puis, une cinquième couche (**C5**) qui est une fully connected avec 120 cartes neurones de sortie est appliquée. Chacune des 120 neurones de **C5** est connectée à l'ensemble des 400 neurones ($5 \times 5 \times 16$) de la quatrième couche **S4**.

La sixième et septième couche sont des respectivement des fully connected avec 84 et 10 cartes neurones de sortie

Une couche de softmax est appliquée sur les 10 derniers neurones qu'on nomme logit permettant de prédire les classes entre 0 et 9 des images d'entrée.

Chapter 5

Unsupervised Learning

Dans les chapitres précédents, nous nous sommes concentrés sur l'apprentissage supervisé, c'est-à-dire que l'on disposait d'une étiquette y_i pour chaque observation \mathbf{x}_i . A contrario dans ce chapitre nous allons aborder l'apprentissage non supervisé. La différence majeure entre ces deux branches de l'apprentissage automatique est que les données traitées en unsupervised learning ne sont pas étiquetées, le but va alors être d'apprendre une fonction sur l'espace des données qui vérifie certaines propriétés. Nous traiterons de deux grands types de problèmes d'apprentissage non supervisé : la réduction de dimension et le clustering.

5.1 Réduction de dimension

La réduction de dimension vise, comme son nom l'indique, à trouver une représentation des données dans un espace de dimension plus faible que celle de l'espace d'origine. Cette réduction de dimension permet entre autre de réduire les temps de calcul ainsi que l'espace mémoire nécessaire au traitement du problème en question. On peut utiliser un tel algorithme pour prétraiter un problème d'apprentissage supervisé puisque la réduction de la dimension des données améliore très souvent les performances d'un tel algorithme.

Ainsi, d'un point de vue formel, on peut définir un problème de réduction de dimension comme la recherche d'un certain $d' < d$ afin de transformer les observations $\mathbf{x}_1, \dots, \mathbf{x}_n \in \mathcal{X}$ de dimension d en des observations "réduites" $\mathbf{z}_1, \dots, \mathbf{z}_n \in \mathcal{Z}$ de dimension d' qui doivent vérifier certaines propriétés propres au problème traité.

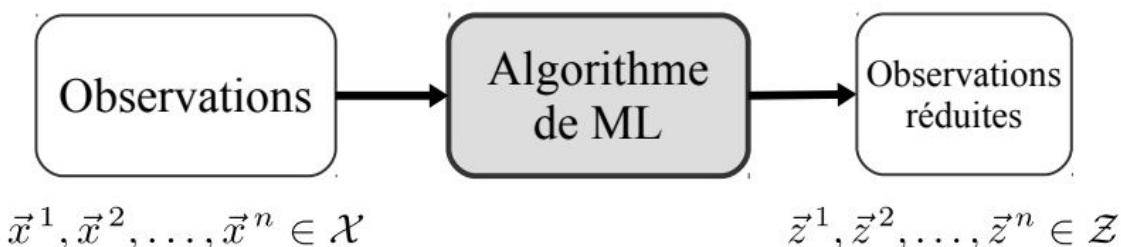


Figure 5.1: Principe de la réduction de dimension [13]

5.1.1 La malédiction des grandes dimensions

Avant d'aller plus loin et de détailler des méthodes permettant de réduire la dimension des données, intéressons-nous à une des principales motivations qui nous pousse à vouloir le faire. Nous avons déjà évoqué ce concept en 1.7.4 dans le cadre de la méthode des k plus proches voisins. Il s'agit d'un phénomène qui apparaît lorsque l'on traite d'un problème en grande dimension et qui rend les classes d'un problème de plus en plus difficiles à séparer à mesure que la dimension augmente.

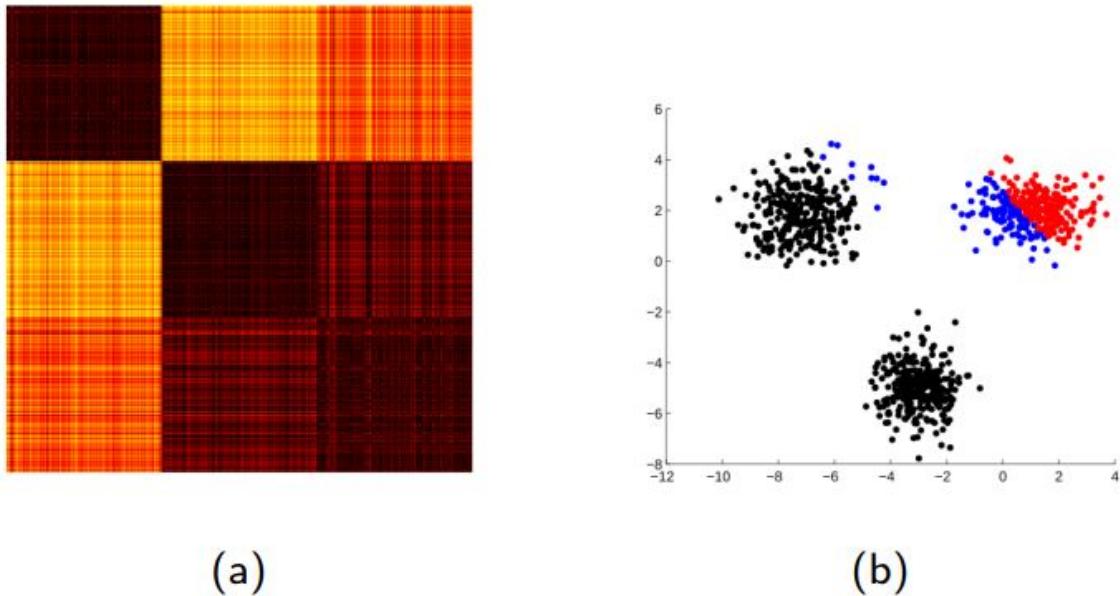


Figure 5.2: Illustration de la malédiction des grandes dimensions pour $d = 10$

La Figure 5.2 (a) représente la matrice de Gram des données définie telle que $\forall i, j \in [|1, n|]$, la composante située à la i -ème ligne et à la j -ème colonne correspond au produit scalaire $\mathbf{x}_i^T \cdot \mathbf{x}_j$. Ici cette matrice montre que les classes sont difficiles à séparer, dans le cas contraire on aurait une diagonale de blocs sombres carrés et le reste de la matrice serait claire.

Cette difficulté à séparer les données se traduit par la Figure 5.2 (b) où l'on voit que les trois groupes de données ne sont pas bien clusterisés. Tous ces soucis sont dûs à la malédiction des grandes dimensions, d'où l'envie de créer des méthodes de réduction de dimension pour éviter ces problèmes.

Pour expliquer ce phénomène on peut étudier le rapport entre la distance maximale entre deux observations et la distance minimale entre deux observations :

$$R = \frac{\max_{(i,j)} \|\mathbf{x}_i - \mathbf{x}_j\|_2}{\min_{(i,j)} \|\mathbf{x}_i - \mathbf{x}_j\|_2}$$

On remarque que ce rapport tend vers 1 lorsque la dimension des données devient grande, comme illustré en Figure 5.3. C'est pourquoi il devient difficile pour l'algorithme de déterminer à quelle classe appartient une observation, puisque ces dernières sont toutes situées à la même distance les unes des autres.

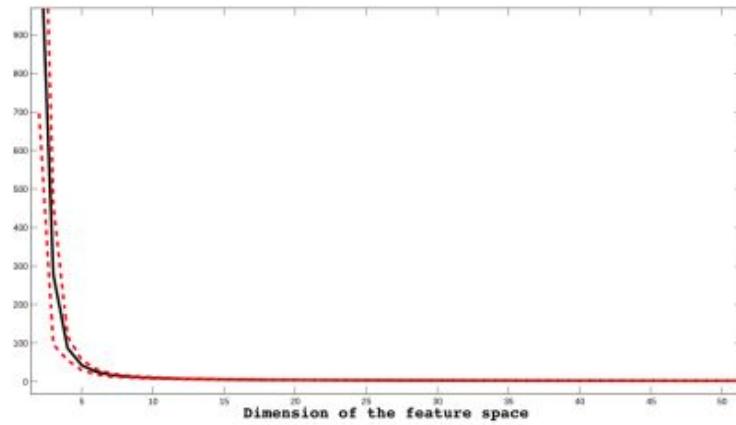


Figure 5.3: Evolution de la valeur de R en fonction de d en sélectionnant aléatoirement 500 observations $\mathbf{x}_i \in [0, 1]^d$

5.1.2 Analyse en composantes principales (ou *Principal Component Analysis*)

L'analyse en composantes principales, que nous noterons par la suite **PCA**, consiste à réduire la dimension des données via un changement de base faisant en sorte que la première composante des données sous leur nouvelle forme soit celle ayant la plus grande variance, que la seconde composante ait la seconde plus grande variance etc...

Pour ce faire on part d'un jeu de n points $\mathcal{D} = \{\mathbf{x}_i\}_{i=1}^n \in \mathbb{R}^d$ pour le transformer en $\mathcal{D}' = \{\mathbf{x}'_i\}_{i=1}^n \in \mathbb{R}^{d'}$ avec $d' \ll d$.

On note $\{\omega_j\}_{j=1}^n \in \mathbb{R}^d$ les j composantes principales. Le but de la méthode PCA va être de trouver ce jeu de vecteurs $\{\omega_j\}_{j=1}^n$ tel que :

$$\arg \min_{\omega_j} \left[\frac{1}{n} \sum_{i=1}^n \left\| \mathbf{x}_i - \mathbf{x}_i^T \cdot \omega_j \frac{\omega_j}{\|\omega_j\|} \right\|^2 \right], \quad \forall j \in [|1, d|] \quad (5.1)$$

On développe

$$\left\| \mathbf{x}_i - \mathbf{x}_i^T \cdot \omega_j \frac{\omega_j}{\|\omega_j\|} \right\|^2 = 1 - 2 \frac{(\mathbf{x}_i^T \cdot \omega_j)^2}{\|\omega_j\|} + (\mathbf{x}_i^T \cdot \omega_j)^2$$

En ajoutant la contrainte $\|\omega_j\|^2 = 1$ et en réinjectant dans (5.1) on obtient la nouvelle fonction objectif :

$$\arg \max_{\omega_j, \|\omega_j\|^2=1} \frac{1}{n} \sum_{i=1}^n (\mathbf{x}_i^T \cdot \omega_j)^2, \quad \forall j \in [|1, d|]$$

On peut montrer que cela revient à chercher les composantes qui maximisent la variance :

$$var(\mathbf{x}_i^T \cdot \omega_j) = \frac{1}{n} \sum_{i=1}^n (\mathbf{x}_i^T \cdot \omega_j)^2 - \left(\frac{1}{n} \sum_{i=1}^n \mathbf{x}_i^T \cdot \omega_j \right)^2$$

En effet, si on considère que les données sont centrées, c'est-à-dire que $\sum_{i=1}^n \mathbf{x}_i = 0$, alors on peut simplifier :

$$\text{var}(\mathbf{x}_i^T \cdot \boldsymbol{\omega}_j) = \frac{1}{n} \sum_{i=1}^n (\mathbf{x}_i^T \cdot \boldsymbol{\omega}_j)^2$$

On peut réécrire le problème sous forme matricielle en notant $X \in \mathcal{M}_{n,d}(\mathbb{R})$ la matrice contenant les observations :

$$\begin{aligned} \frac{1}{n} \sum_{i=1}^n (\mathbf{x}_i^T \cdot \boldsymbol{\omega}_j)^2 &= \frac{1}{n} (X \boldsymbol{\omega}_j)^T (X \boldsymbol{\omega}_j) \\ &= \boldsymbol{\omega}_j^T \left(\frac{1}{n} X^T X \right) \boldsymbol{\omega}_j \\ &= \boldsymbol{\omega}_j^T V \boldsymbol{\omega}_j \end{aligned}$$

avec $V = \frac{1}{n} X^T X, V \in \mathcal{M}_{d,d}(\mathbb{R})$ la covariance de X . Ainsi le problème se réécrit :

$$\arg \max_{\boldsymbol{\omega}_j, \|\boldsymbol{\omega}\|^2=1} \boldsymbol{\omega}_j^T V \boldsymbol{\omega}_j, \quad \forall j \in [1, d] \quad (5.2)$$

Pour pouvoir résoudre ce problème, il est nécessaire d'introduire quelques notions qui nous permettront à terme d'utiliser le multiplicateur de Lagrange. Commençons par la notion d'extremum local sous contrainte. Soient f et g de fonctions de deux variables définies respectivement sur D_f et D_g . Soit $P_0 = (x_0, y_0)$ un point appartenant à $D_f \cap D_g$ tel que $g(x_0, y_0) = 0$. P_0 est un maximum (respectivement minimum) local de f sur $D = \{(x, y) | g(x, y) = 0\}$ s'il existe un voisinage \mathcal{V} de P_0 tel que pour tout $(x, y) \in \mathcal{V}$ tel que $g(x, y) = 0, f(x, y) \leq f(x_0, y_0)$ (respectivement $f(x, y) \geq f(x_0, y_0)$).

Après cela on peut énoncer le théorème de la condition nécessaire sur l'extremum local. On a toujours f et g deux fonctions de deux variables mais qui sont cette fois-ci de classe C^1 ainsi que $P_0 = (x_0, y_0) \in D_f \cap D_g$ avec $g(x_0, y_0) = 0$. Si P_0 est un extremum local de f sur $D = \{(x, y) | g(x, y) = 0\}$ et $\nabla g(x_0, y_0) \neq 0$ alors il existe $\lambda_0 \in \mathbb{R}$ tel que :

$$\nabla f(x_0, y_0) = \lambda_0 \nabla g(x_0, y_0)$$

On dit alors que P_0 est un point stationnaire de f sur D et λ_0 est appelé multiplicateur de Lagrange associé. Ce multiplicateur de Lagrange est utilisé pour trouver les maximums et minimums locaux d'une fonction soumise à des contraintes.

Une propriété fondamentale du multiplicateur de Lagrange est la suivante. P_0 est un extrema de f sur D associé au multiplicateur de Lagrange $\lambda_0 \in \mathbb{R}$ si et seulement si (x_0, y_0, λ_0) est solution de :

$$\begin{cases} \frac{\partial \mathcal{L}}{\partial x}(x, y, \lambda) = 0 \\ \frac{\partial \mathcal{L}}{\partial y}(x, y, \lambda) = 0 \\ \frac{\partial \mathcal{L}}{\partial \lambda}(x, y, \lambda) = 0 \end{cases}$$

avec $\mathcal{L} = f + \lambda g$.

Une fois ces notions introduites on peut les appliquer au problème (5.2) en réécrivant la fonction objectif :

$$\mathcal{L}(\omega_j, \lambda) = \omega_j^T V \omega_j - \lambda(\omega_j^T \omega_j - 1)$$

On veut maximiser cette fonction, il faut donc la dériver et trouver le point en lequel cette dérivée est nulle :

$$\frac{\partial \mathcal{L}}{\partial \omega_j}(\omega_j, \lambda) = 2V\omega_j - 2\lambda\omega_j = 0$$

On obtient alors la solution :

$$V\omega_j = \lambda\omega$$

Ainsi la composante principale qui satisfait la fonction souhaitée est un vecteur propre de la matrice de covariance V . De plus celle qui maximise $\mathcal{L}(\omega, \lambda)$ est celle ayant la plus grande valeur propre. On peut donc trouver tous les ω_j en décomposant V en valeurs singulières (ou **SVD** en anglais).

Écrivons maintenant la matrice de covariance V . Pour ce faire on notera $x_{i,j}$ la j -ème composante de \mathbf{x}_i :

$$V = \begin{pmatrix} Var(x_{.,1}) & Covar(x_{.,1}, x_{.,2}) & \cdots & Covar(x_{.,1}, x_{.,d}) \\ Covar(x_{.,2}, x_{.,1}) & Var(x_{.,2}) & \cdots & Covar(x_{.,2}, x_{.,d}) \\ \vdots & \ddots & \vdots & \vdots \\ Covar(x_{.,d-1}, x_{.,1}) & \cdots & Var(x_{.,d-1}) & Covar(x_{.,d-1}, x_{.,d}) \\ Covar(x_{.,d}, x_{.,1}) & \cdots & Covar(x_{.,d}, x_{.,d-1}) & Var(x_{.,d}) \end{pmatrix}$$

Avec

$$Var(x_{.,j}) = \frac{1}{n} \sum_{i=1}^n x_{i,j}^2 - \left(\frac{1}{n} \sum_{i=1}^n x_{i,j} \right)^2$$

Et

$$Covar(x_{.,j_1}, x_{.,j_2}) = \frac{1}{n} \sum_{i=1}^n x_{i,j_1} x_{i,j_2} - \left(\frac{1}{n} \sum_{i=1}^n x_{i,j_1} \right) \left(\frac{1}{n} \sum_{i=1}^n x_{i,j_2} \right)$$

Il existe plusieurs approches pour choisir la dimension réduite d' , qui correspond à la composante principale que l'on conserve. Une méthode courante s'appuie sur le constat suivant : si on conserve des données de dimension d' , alors les $d - d'$ valeurs propres qui ne sont pas retenues doivent être relativement petites par rapport au problème. On utilise alors comme critère le rapport :

$$Prop = \frac{\sum_{i=1}^{d'} \lambda_i}{\sum_{i=1}^d \lambda_i}$$

Ce rapport donne la proportion de variance que l'on conserve en réduisant la dimension des données. On écrit $\Omega_{d'} \in \mathcal{M}_d(\mathbb{R})$ la matrice carrée de taille d qui contient les d' vecteurs propres correspondant aux valeurs propres les plus élevées et dont les $d - d'$ colonnes restantes sont nulles. Le théorème d'Eckart-Young [31] nous permet alors de déterminer l'erreur de la réduction de dimension :

$$Err_{PCA} = \|V - \Omega_{d'}^T V \Omega_{d'}\|_F^2 = \sum_{i=d'+1}^d \lambda_i^2$$

L'algorithme PCA

Pour appliquer le PCA en pratique sur un jeu de données $\mathcal{D} = \{\mathbf{x}_i\}_{i=1}^n$ on doit suivre l'algorithme dont le pseudo-code est le suivant :

1. Calculer la covariance V de \mathcal{D}
2. Évaluer la SVD de V , on note $\{\omega_i\}_{i=1}^d$ les vecteurs propres et $\{\lambda_i\}_{i=1}^d$ les valeurs propres
3. Ordonner les valeurs propres et les vecteurs propres associés par ordre décroissant
4. Prendre les d' premiers vecteurs tels que $Prop$ atteigne un critère prédéfini
5. Projeter les données sur cette nouvelle base

5.1.3 Utilisation de noyaux (kernels)

Le "kernel trick"

Les méthodes utilisant des noyaux sont utilisées pour définir des fonctions de décision non-linéaire tout en utilisant des méthodes linéaires, comme c'est le cas pour le PCA. Ces méthodes permettent de projeter les données dans un espace non-linéaire pouvant être de dimension infinie, dans lesquels des méthodes linéaires pourraient être utilisées.

Commençons par définir un noyau (on utilisera souvent le terme anglais **kernel**). Il s'agit d'une fonction $\mathcal{K} : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$ qui est symétrique et hermitienne.

Cependant on travaille presque toujours sur des kernels définis positifs, c'est-à-dire un kernel tel que $\forall \{x_1, \dots, x_n\} \in \mathcal{X}^n$ et $\forall \{\alpha_1, \dots, \alpha_n\} \in \mathbb{C}^n$ on a :

$$\sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j^* \mathcal{K}(x_i, x_j) \geq 0$$

Avec un tel kernel on peut définir une matrice de similarité $K = (\mathcal{K}(x_i, x_j))_{i,j \in [|1,n|]}$ qui est définie semi-positive. On peut aussi appeler une telle matrice matrice de Gram ou matrice kernel (c'est une matrice de ce type qui est représentée en Figure 5.2 (a)).

Pour aller plus loin il est nécessaire de rappeler la définition d'un espace de Hilbert. Il s'agit d'un espace vectoriel \mathcal{H} muni d'un produit scalaire réel ou complexe complet, ce qui signifie que toute suite de Cauchy d'élément de cette espace converge dans cet espace.

On dispose du théorème de Moore-Aronszajn qui stipule qu'un kernel \mathcal{K} est défini positif si et seulement si il existe un espace de Hilbert \mathcal{H} et une fonction $\phi : \mathcal{X} \rightarrow \mathcal{H}$ telle que

$$\mathcal{K}(x_i, x_j) = \langle \phi(x_i), \phi(x_j) \rangle_{\mathcal{H}}$$

Où $\langle \cdot, \cdot \rangle_{\mathcal{H}}$ est le produit scalaire de l'espace \mathcal{H} .

Définissons maintenant un espace de Hilbert à noyau reproduisant (abrégé en RKHS en anglais). Soit $(\mathcal{H}, \langle \cdot, \cdot \rangle_{\mathcal{H}})$ un espace de Hilbert de fonctions de \mathcal{X} dans \mathbb{C} . La fonction $\mathcal{K} : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{C}$ est le noyau reproduisant de \mathcal{H} , en supposant qu'il en admette un, si et seulement si on a les propriétés suivantes :

- Pour tout $x \in \mathcal{X}$, la fonction $k(x, \cdot) : t \rightarrow \mathcal{K}(x, t)$ appartient à \mathcal{H}
 - Pour tout $x \in \mathcal{X}, g \in \mathcal{H}$, la propriété reproduisante est vérifiée c'est à dire: $g(x) = \langle g, k(x, \cdot) \rangle_{\mathcal{H}}$
- Si un noyau reproduisant existe, alors \mathcal{H} est dit espace de Hilbert à noyau reproduisant (RKHS).

Une fois cette notion définie on peut énoncer le théorème du représentant (Representer theorem). Soient un espace \mathcal{X} doté d'un kernel défini positif \mathcal{K} , $\mathcal{H}_{\mathcal{K}}$ le RKHS correspondant et $x_1, \dots, x_n \in \mathcal{X}$ un jeu fini de points. Soit $\psi : \mathbb{R}^{n+1} \rightarrow \mathbb{R}$ une fonction de $n + 1$ variables strictement croissante par rapport à la dernière variable. Alors toute solution du problème d'optimisation :

$$\min_{g \in \mathcal{H}_{\mathcal{K}}} \psi(g(x_1), \dots, g(x_n), \|g\|_{\mathcal{H}_{\mathcal{K}}})$$

Admet une représentation de la forme :

$$\forall x \in \mathcal{X}, \quad g(x) = \sum_{i=1}^n \alpha_i \mathcal{K}(x_i, x)$$

Où $\|g\|_{\mathcal{H}_{\mathcal{K}}} = \sqrt{\langle g, g \rangle_{\mathcal{H}_{\mathcal{K}}}}$.

Ce théorème est très important, car il permet d'approximer n'importe quelle fonction dans le RKHS.

Le kernel PCA

On considère toujours un jeu de données $\mathcal{D} = \{\mathbf{x}_i\}_{i=1}^n$. On peut utiliser une fonction de *mapping* vers un RKHS \mathcal{H} , qui peut avoir des propriétés intéressantes :

$$\phi = \begin{cases} \mathbb{R}^d & \rightarrow \mathcal{H} \\ \mathbf{x}_i & \rightarrow \phi(\mathbf{x}_i) \end{cases}$$

La fonction ϕ peut être non-linéaire et dépend du choix du noyau utilisé. En supposant que l'on ait choisi un kernel, le but du kernel PCA (KPCA) est de trouver l'ensemble $\{\omega_j\}_{j=1}^d$ qui minimise la quantité :

$$\forall j \in [|1, P|], \quad \frac{1}{n} \sum_{i=1}^n \|\phi(\mathbf{x}_i) - \langle \phi(\mathbf{x}_i), \omega_j \rangle_{\mathcal{H}_{\mathcal{K}}} \frac{\omega_j}{\|\omega_j\|_{\mathcal{H}_{\mathcal{K}}}^2}\|^2_{\mathcal{H}_{\mathcal{K}}}$$

En faisant un calcul similaire à celui effectué pour le PCA on finit par obtenir :

$$\mathcal{L}(\omega_j, \lambda) = \frac{1}{n} \sum_{i=1}^n \langle \phi(\mathbf{x}_i), \omega_j \rangle_{\mathcal{H}_{\mathcal{K}}}^2 - \lambda (\|\omega_j\|_{\mathcal{H}_{\mathcal{K}}}^2 - 1)$$

Avec $\lambda \in \mathbb{R}$. Le théorème du représentant nous permet d'écrire :

$$\omega_j = \sum_{q=1}^n \alpha_{q,j} \phi(\mathbf{x}_q)$$

D'où

$$\mathcal{L}(\alpha_j, \lambda) = \frac{1}{n} \sum_{i=1}^n \left(\sum_{q=1}^n \alpha_{q,j} \langle \phi(\mathbf{x}_i), \phi(\mathbf{x}_q) \rangle_{\mathcal{H}_{\mathcal{K}}} \right)^2 - \lambda \left(\sum_{(k,q) \in [|1,n|]^2} \alpha_{q,j} \alpha_{k,j} \mathcal{K}(\mathbf{x}_q, \mathbf{x}_k) - 1 \right)$$

On peut le réécrire sous forme matricielle avec la notation $K = (\mathcal{K}(x_i, x_j))_{i,j \in [|1,n|]}$:

$$\mathcal{L}(\alpha_j, \lambda) = \frac{1}{n} \alpha_j^T K^2 \alpha_j - \lambda (\alpha_j^T K \alpha_j - 1)$$

Le principal avantage du kernel trick vient du fait que l'on peut utiliser différents kernels sans avoir à déterminer explicitement la fonction de mapping ϕ . Grâce à cela on peut utiliser une très grande variété de kernels. Parmi les plus populaires on peut citer :

- les noyaux polynomiaux : $\mathcal{K}(\mathbf{x}_i, \mathbf{x}_j) = (\langle \mathbf{x}_i, \mathbf{x}_j \rangle_{\mathbb{R}^d} + c)^P$ avec P le degré du noyau et c une constante.
- les noyaux rbf, ou gaussiens : $\mathcal{K}(\mathbf{x}_i, \mathbf{x}_j) = \exp\left(-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|_{\mathbb{R}^d}^2}{2\sigma^2}\right)$ avec un paramètre σ . À noter que ce noyau amène les données dans un espace de dimension infinie.

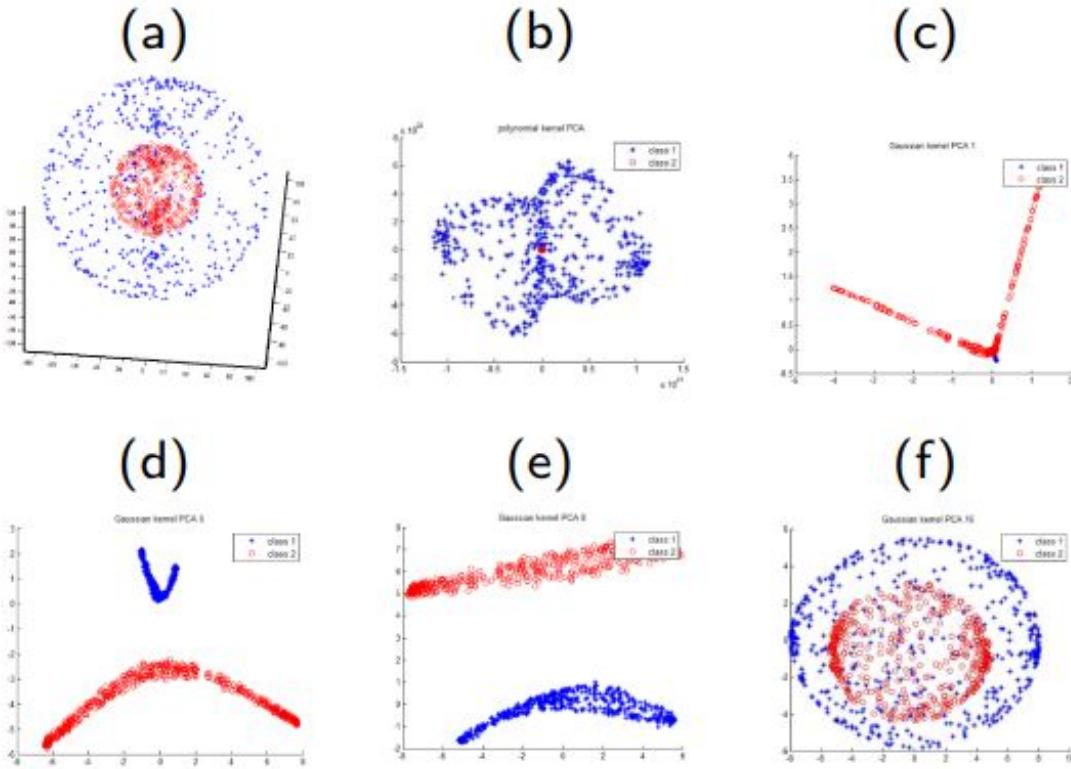


Figure 5.4: (a) Données classées en 2 sphères concentriques, (b) KPCA polynomial avec $p = 5$, (c) KPCA gaussien avec σ , (d) KPCA gaussien avec $5.\sigma$, (e) KPCA gaussien avec $8.\sigma$, (f) KPCA gaussien avec $15.\sigma$

La Figure 5.4 montre bien l'importance du choix du KPCA utilisé. En effet, les données initiales sont difficiles à séparer à cause de la forme sphérique des deux classes et il semble alors judicieux de les transformer. Toufois tous les KPCA n'ont pas la même efficacité sur ces données et les noyaux utilisés en (b) et (f) n'offrent aucune aide à la séparation des données. Sur cet exemple il serait judicieux de choisir un des KPCA utilisés en (d) ou (e) car les données y sont clairement séparées.

5.1.4 Positionnement multidimensionnel (MDS)

Le positionnement multidimensionnel, ou *Multidimensional scaling (MDS)*, est une technique utilisée en data mining permettant de diminuer la dimension des données en faisant en sorte de conserver les distances entre les données :

$$\mathcal{D} = \{\mathbf{x}_i\}_{i=1}^n \rightarrow \mathcal{D}' = \{\mathbf{x}'_i\}_{i=1}^n$$

Avec $\forall i, j \in [|1, n|]^2$, $\|\mathbf{x}_i - \mathbf{x}_j\| \simeq \|\mathbf{x}'_i - \mathbf{x}'_j\|$

Il existe principalement deux fonctions objectifs pouvant être utilisées pour cette méthode, la plus simple est :

$$\phi(\mathcal{D}') = \sum_{i,j \in [|1, n|]^2} |\|\mathbf{x}_i - \mathbf{x}_j\|_2^2 - \|\mathbf{x}'_i - \mathbf{x}'_j\|_2^2|$$

La seconde est appelée fonction de coût de Sammon :

$$\phi(\mathcal{D}') = - \frac{1}{\sum_{i,j \in [|1, n|]^2} \|\mathbf{x}_i - \mathbf{x}_j\|_2} \sum_{i,j \in [|1, n|]^2} |\|\mathbf{x}_i - \mathbf{x}_j\|_2^2 - \|\mathbf{x}'_i - \mathbf{x}'_j\|_2^2|$$

La principale différence entre ces deux fonctions est le facteur $\frac{1}{\sum_{i,j \in [|1, n|]^2} \|\mathbf{x}_i - \mathbf{x}_j\|_2}$ qui permet de mettre en valeur les éléments de \mathcal{D} qui sont les plus similaires. Dans la suite de cette section on notera D la matrice de distance entre les données telle que $D_{i,j} = \|\mathbf{x}_i - \mathbf{x}_j\|_2$ et S la matrice telle que $S_{i,j} = \|\mathbf{x}_i - \mathbf{x}_j\|_2^2$.

L'idée de base du MDS est de transformer la matrice de distances en une matrice de produits scalaires. Cette matrice G , aussi appelée matrice de Gram, peut être définie telle que $G_{i,j} = \langle \mathbf{x}_i, \mathbf{x}_j \rangle$ ou encore comme étant $G = X^T X$ avec X la matrice contenant les données.

Montrons maintenant le lien entre D et G . Par définition on a :

$$\begin{aligned} \|\mathbf{x}_i - \mathbf{x}_j\|_2 &= \sqrt{\langle \mathbf{x}_i - \mathbf{x}_j, \mathbf{x}_i - \mathbf{x}_j \rangle} \\ \|\mathbf{x}_i - \mathbf{x}_j\|_2 &= \sqrt{\langle \mathbf{x}_i, \mathbf{x}_i \rangle + \langle \mathbf{x}_j, \mathbf{x}_j \rangle - 2 \langle \mathbf{x}_i, \mathbf{x}_j \rangle} \\ D_{i,j} &= \sqrt{G_{i,i} + G_{j,j} - 2G_{i,j}} \end{aligned}$$

Pour la suite on aura besoin d'une matrice de centralisation $H = I - \frac{1}{n} \mathbf{1}\mathbf{1}^T$ avec $\mathbf{1}$ le vecteur colonne de taille n ne contenant que des 1. On peut montrer rapidement en utilisant que $\mathbf{1}\mathbf{1}^T = n \cdot \mathbf{1}\mathbf{1}^T$:

$$H \times H = H$$

De même avec $\mathbf{1}^T \times \mathbf{1}\mathbf{1}^T = n \cdot \mathbf{1}^T$ on a :

$$H \times \mathbf{1} = 0$$

Enfin si on a X une matrice de données centrée, c'est-à-dire que $moyenne(X) = 0$, alors $H \times X = X$. On peut le montrer en utilisant qu'avec X centrée, $X \times \mathbf{1}\mathbf{1}^T = 0$.

Dans la suite on indiquera qu'une matrice M est centrée ainsi : M^c . Les trois propriétés précédentes sur H nous permettent de dire que, pour une matrice de données X^c de matrice de Gram associée G^c , on a :

$$H \times G^c \times H = G^c \quad (5.3)$$

La preuve tient en une ligne : $G^c = (X^c)^T X^c = H(X^c)^T X^c H = HG^c H$.

Après ça on peut montrer en utilisant que $\|\mathbf{x}_i - \mathbf{x}_j\|_2 = \|(\mathbf{x}_i - \text{moyenne}(X)) - (\mathbf{x}_j - \text{moyenne}(X))\|_2$:

$$D_{i,j} = \sqrt{G_{i,i}^c + G_{j,j}^c - 2G_{i,j}^c} \quad (5.4)$$

Montrons maintenant la relation entre la matrice G^c et la matrice S^c :

$$G^c = \frac{-1}{2} \cdot S^c = \frac{-1}{2} \cdot HS \quad (5.5)$$

Pour le prouver on traduit matriciellement la relation (5.3) :

$$H \times G^c \times H = (I - \frac{1}{n} \cdot \mathbf{1}\mathbf{1}^T) \times G^c \times (I - \frac{1}{n} \cdot \mathbf{1}\mathbf{1}^T)$$

$$\text{Ainsi } H \times G^c \times H = G^c - \frac{1}{n} \cdot \mathbf{1}\mathbf{1}^T \times G^c - \frac{1}{n} \cdot G^c \times \mathbf{1}\mathbf{1}^T + \frac{1}{n^2} \cdot \mathbf{1}\mathbf{1}^T \times G^c \times \mathbf{1}\mathbf{1}^T = G^c$$

$$\text{Donc, } \mathbf{1}\mathbf{1}^T \times G^c + G^c \times \mathbf{1}\mathbf{1}^T - \frac{1}{n} \cdot \mathbf{1}\mathbf{1}^T \times G^c \times \mathbf{1}\mathbf{1}^T = 0$$

Cependant on a :

$$\mathbf{1}\mathbf{1}^T \times G^c = \begin{pmatrix} 1 & \dots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \dots & 1 \end{pmatrix} \times \begin{pmatrix} G_{11}^c & \dots & G_{1n}^c \\ \vdots & \ddots & \vdots \\ G_{n1}^c & \dots & G_{nn}^c \end{pmatrix}$$

$$\text{Puis par produit on a } \mathbf{1}\mathbf{1}^T \times G^c = \begin{pmatrix} \sum_{i=1}^n G_{i1}^c & \dots & \sum_{i=1}^n G_{in}^c \\ \vdots & \ddots & \vdots \\ \sum_{i=1}^n G_{i1}^c & \dots & \sum_{i=1}^n G_{in}^c \end{pmatrix}$$

$$\text{De la même façon : } G^c \times \mathbf{1}\mathbf{1}^T = \begin{pmatrix} \sum_{j=1}^n G_{1j}^c & \dots & \sum_{j=1}^n G_{1j}^c \\ \vdots & \ddots & \vdots \\ \sum_{j=1}^n G_{nj}^c & \dots & \sum_{j=1}^n G_{nj}^c \end{pmatrix}$$

$$\text{Et alors : } \mathbf{1}\mathbf{1}^T \times G^c \times \mathbf{1}\mathbf{1}^T = \begin{pmatrix} \sum_{i,j \in [1,n]^2} G_{ij}^c & \dots & \sum_{i,j \in [1,n]^2} G_{ij}^c \\ \vdots & \ddots & \vdots \\ \sum_{i,j \in [1,n]^2} G_{ij}^c & \dots & \sum_{i,j \in [1,n]^2} G_{ij}^c \end{pmatrix}$$

On obtient finalement:

$$\begin{aligned} & \begin{pmatrix} \sum_{i=1}^n G_{i1}^c & \cdots & \sum_{i=1}^n G_{in}^c \\ \vdots & \ddots & \vdots \\ \sum_{i=1}^n G_{i1}^c & \cdots & \sum_{i=1}^n G_{in}^c \end{pmatrix} + \begin{pmatrix} \sum_{j=1}^n G_{1j}^c & \cdots & \sum_{j=1}^n G_{1j}^c \\ \vdots & \ddots & \vdots \\ \sum_{j=1}^n G_{nj}^c & \cdots & \sum_{j=1}^n G_{nj}^c \end{pmatrix} \\ & - \frac{1}{n} \cdot \begin{pmatrix} \sum_{i,j \in [1,n]^2} G_{ij}^c & \cdots & \sum_{i,j \in [1,n]^2} G_{ij}^c \\ \vdots & \ddots & \vdots \\ \sum_{i,j \in [1,n]^2} G_{ij}^c & \cdots & \sum_{i,j \in [1,n]^2} G_{ij}^c \end{pmatrix} = 0_n(\mathbb{R}) \end{aligned}$$

En multipliant cette équation à droite par $\mathbf{1}$ on a:

$$\begin{pmatrix} \sum_{i,j \in [1,n]^2} G_{ij}^c \\ \vdots \\ \sum_{i,j \in [1,n]^2} G_{ij}^c \end{pmatrix} + n \cdot \begin{pmatrix} \sum_{j=1}^n G_{1j}^c \\ \vdots \\ \sum_{j=1}^n G_{nj}^c \end{pmatrix} - \begin{pmatrix} \sum_{i,j \in [1,n]^2} G_{ij}^c \\ \vdots \\ \sum_{i,j \in [1,n]^2} G_{ij}^c \end{pmatrix} = 0_{n,1}(\mathbb{R})$$

Donc $\forall i \in [1, n]$ on a $\sum_{j=1}^n G_{ij}^c = 0$ et grâce à la symétrie de G^c , $\forall j \in [1, n]$ on a $\sum_{i=1}^n G_{ij}^c = 0$. Cependant on a $D_{i,j}^2 = G_{i,i}^c + G_{j,j}^c - 2G_{i,j}^c$ grâce à (5.4) donc finalement,

$$\sum_{i=1}^n D_{i,j}^2 = \sum_{i=1}^n G_{i,i}^c + n \cdot G_{j,j}^c \quad (5.6)$$

$$\sum_{j=1}^n D_{i,j}^2 = n \cdot G_{i,i}^c + \sum_{j=1}^n G_{j,j}^c \quad (5.7)$$

Mais on a aussi

$$\begin{aligned} S^c &= H \times S \times H = (I - \frac{1}{n} \cdot \mathbf{1}\mathbf{1}^T) \times S \times (I - \frac{1}{n} \cdot \mathbf{1}\mathbf{1}^T) \\ S^c &= S - \frac{1}{n} \cdot \mathbf{1}\mathbf{1}^T \times S - \frac{1}{n} \cdot S \times \mathbf{1}\mathbf{1}^T - \frac{1}{n^2} \cdot \mathbf{1}\mathbf{1}^T \times S \times \mathbf{1}\mathbf{1}^T \end{aligned}$$

En réécrivant matriciellement, et en utilisant $S = \{D_{ij}^2\}_{i,j \in [1,n]^2}$ ainsi que toutes les opérations matricielles que l'on a effectuées précédemment on a :

$$S_{ij}^c = D_{ij}^2 - \frac{1}{n} \cdot \sum_{i=1}^n D_{i,j}^2 - \frac{1}{n} \cdot \sum_{j=1}^n D_{i,j}^2 + \frac{1}{n^2} \cdot \sum_{i,j \in [1,n]^2} D_{ij}^2 \quad (5.8)$$

En utilisant 5.6 et en mettant ces équations dans 5.8 on a la nouvelle équation :

$$\begin{aligned} S_{ij}^c &= D_{ij}^2 - \frac{1}{n} \cdot \left(\sum_{i=1}^n G_{i,i}^c + n \cdot G_{j,j}^c - n \cdot G_{i,i}^c + \sum_{j=1}^n G_{j,j}^c - \frac{1}{n} \cdot \sum_{i,j \in [1,n]^2} D_{ij}^2 \right) \\ S_{ij}^c &= D_{ij}^2 - G_{ii}^c - G_{jj}^c \end{aligned}$$

Grâce à (5.4) on a :

$$S_{ij}^c = -2 \cdot G_{ij}^c \quad (5.9)$$

Maintenant que l'on dispose de cette relation, introduisons la norme Frobenius pour les matrices $A = \{a_{ij}\}_{i,j \in [1,n]^2} \in M_n(\mathbb{R})$:

$$\|A\|_F = \sqrt{\sum_{i,j \in [1,n]^2} |a_{ij}|^2} \quad (5.10)$$

Cette norme a la propriété suivante : si on a $A, U \in M_n(\mathbb{R})^2$ tel que $\|U\|_F = 1$, alors on a :

$$\|A \times U\|_F = \|U \times A\|_F = \|A\|_F \quad (5.11)$$

Revenons à notre problème qui consiste en la minimisation de

$$\phi(\mathcal{D}') = \sum_{i,j \in [1,n]^2} (\|\mathbf{x}'_i - \mathbf{x}'_j\|^2 - \|\mathbf{x}_i - \mathbf{x}_j\|^2)$$

On note X la matrice associée au jeu de données $\{\mathbf{x}_i\}_{i \in [1,n]} \in \mathbb{R}^d$, X' celle associée à $\{\mathbf{x}'_i\}_{i \in [1,n]} \in \mathbb{R}^{d'}$, S et S' les matrices des distances euclidiennes au carré de X et X' , G et G' les matrices de Gram associées.

Un autre point de vue pour chercher S' est de minimiser

$$\phi(S') = \sum_{i,j \in [1,n]^2} (S'_{ij} - S_{ij}) \text{ avec } \text{rang}(S') = d'$$

$$\sum_{i,j \in [1,n]^2} (S'_{ij} - S_{ij}) \leq \sum_{i,j \in [1,n]^2} |S'_{ij} - S_{ij}|$$

Grâce à l'équivalence des normes dans $M_n(\mathbb{R})$, on a :

$$\exists \alpha \in \mathbb{R}^+ \|S' - S\|_1 = \sum_{i,j \in [1,n]^2} |S'_{ij} - S_{ij}| \leq \alpha \cdot \|S' - S\|_F$$

Ainsi notre but est maintenant de minimiser

$$\Phi(S') = \|S' - S\|_F \text{ avec } \text{rang}(S') = d' \quad (5.12)$$

Ainsi en utilisant (5.4) avec $\tilde{H} = \frac{H}{\|H\|_F}$ on a :

$$\begin{aligned}\|S' - S\|_F &= \|\tilde{H} \times (S' - S) \times \tilde{H}\|_F = \frac{1}{\|H\|_F^2} \cdot \|H \times (S' - S) \times H\|_F \\ \|S' - S\|_F &= \frac{2}{\|H\|_F^2} \cdot \|G' - G\|_F\end{aligned}$$

On doit donc trouver G' qui minimise :

$$\phi(G') = \|G' - G\|_F \text{ avec } \text{rang}(G') = d'$$

Pour cela on va utiliser le théorème d'Eckart-Young.

Soit une matrice $A \in M_{nm}(\mathbb{R})$, avec $(n, m) \in \mathbb{N}^2$. On peut écrire la décomposition en valeurs singulières $A = U_A \times \Sigma_A \times V_A^T$, où $\Sigma_A = \text{diag}(\sigma_1, \dots, \sigma_r)$ est diagonale et contient les valeurs singulières de A (les racines des valeurs propres de $A \times A$) dans l'ordre décroissant, $U_A \in M_{nr}(\mathbb{R})$ et $V_A \in M_{rm}(\mathbb{R})$ ont des vecteurs orthogonaux qui contiennent les vecteurs singuliers gauche et droite de A , et où $r = \text{rang}(A)$. Si l'on appelle $A' \in M_{nm}(\mathbb{R})$ la meilleure matrice de rang d' approximant A pour la norme Frobenius alors :

$$A' = U_{A,d'} \times \Sigma_{A,d'} \times V_{A,d'}^T$$

Avec $\Sigma_{A,d'}$ la matrice diagonale contenant les d' valeurs singulières les plus grandes de A dans l'ordre décroissant, et $U_{A,d'} \in M_{nr}(\mathbb{R})$ et $V_{A,d'} \in M_{rm}(\mathbb{R})$ qui contiennent les vecteurs singuliers de A associés.

De plus l'erreur de minimisation est

$$\|A' - A\|_F = \sqrt{\sum_{i=d'+1}^r \sigma_i^2} \tag{5.13}$$

On applique alors ce théorème à notre problème et on a $G' = U_{G,d'} \times \Sigma_{G,d} \times U_{G,d'}^T$. Comme on a $G' = X'^T \times X'$ alors $X' = \Sigma_{G,d'}^{1/2} \times U_{G,d'}^T$.

5.1.5 Algorithme t-SNE

L'algorithme t-SNE (pour *t-distributed Stochastic Neighbor Embedding*) est un algorithme d'apprentissage automatique non supervisé permettant de visualiser des données de grande dimension en projectant chaque point dans une carte de dimension deux ou trois. Cette méthode peut trouver des connexions non-linéaires contrairement au PCA. Elle repose sur trois grandes étapes :

- On calcule les similarités des points dans l'espace initial de grande dimension.
- On crée un espace de dimension inférieure dans lequel on représentera nos données.
- On optimise la répartition des points dans le nouvel espace.

Première étape

On dispose d'une base de données $\mathcal{D} = \{\mathbf{x}_i\}_{i=1}^n \in \mathbb{R}^d$. La première étape consiste à calculer les probabilités p_{ij} qui expriment la similarité des données \mathbf{x}_i et \mathbf{x}_j . Pour $i \neq j$ on a :

$$p_{j|i} = \frac{\exp(-\|\mathbf{x}_i - \mathbf{x}_j\|_2^2/2\sigma^2)}{\sum_{k \neq i} \exp(-\|\mathbf{x}_i - \mathbf{x}_k\|_2^2/2\sigma^2)}$$

On en déduit chaque p_{ij} :

$$p_{ij} = \frac{p_{i|j} + p_{j|i}}{2n}$$

Le paramètre σ du noyau gaussien est appelé la perplexité et est à définir.

Deuxième étape

Le but du t-SNE est d'apprendre un mapping $\mathcal{D}' = \{\mathbf{x}'_i\}_{i=1}^n \in \mathbb{R}^{d'}$ qui reflète au mieux les similarités p_{ij} . En règle générale, d' vaut 2 ou 3 puisque l'on veut utiliser la réduction de dimension pour pouvoir visualiser les données. De manière analogue à la première étape, on calcule les similarités q_{ij} des points dans l'espace nouvellement créé en utilisant une loi de Student au lieu d'une gaussienne. On obtient donc :

$$q_{ij} = \frac{(1 + \|\mathbf{x}'_i - \mathbf{x}'_j\|_2^2)^{-1}}{\sum_{k=1}^n \sum_{l \neq k} (1 + \|\mathbf{x}'_k - \mathbf{x}'_l\|_2^2)^{-1}}$$

Troisième étape

On utilise la divergence de Kullback-Leibler pour faire en sorte que la distribution de probabilité jointe des nouvelles données \mathbf{x}'_i soit aussi similaire que possible à celle de l'ancienne base de données. Ainsi on minimise la divergence de Kullback-Leibler entre les distributions P et Q :

$$\text{KL}(P\|Q) = \sum_{i \neq j} p_{ij} \log \frac{p_{ij}}{q_{ij}}$$

Ce problème de minimisation se résout en utilisant l'algorithme de la descente de gradient. **Attention** contrairement à la PCA cette technique est juste une technique de visualisation.

5.1.6 Auto-encodeur

Un auto-encodeur est un réseau de neurones créé dans le but d'apprendre une fonction d'identité de manière non supervisée pour reconstruire les données entrées initialement tout en les compressant pendant le processus.

Dans cette méthode on cherche encore à passer d'une base $\mathcal{D} = \{\mathbf{x}_i\}_{i=1}^n \in \mathbb{R}^d$ à une base $\mathcal{D}' = \{\mathbf{x}'_i\}_{i=1}^n \in \mathbb{R}^{d'}$. On notera g_ϕ l'encodeur DNN (par exemple Deep Neural Network). ϕ représente les poids du réseau de neurones. De même on note f_θ le décodeur DNN où θ représente les poids du réseau de neurones. On a alors que $\forall i \in [|1, n|]$, $\mathbf{x}'_i = f_\theta(g_\phi(\mathbf{x}_i))$.

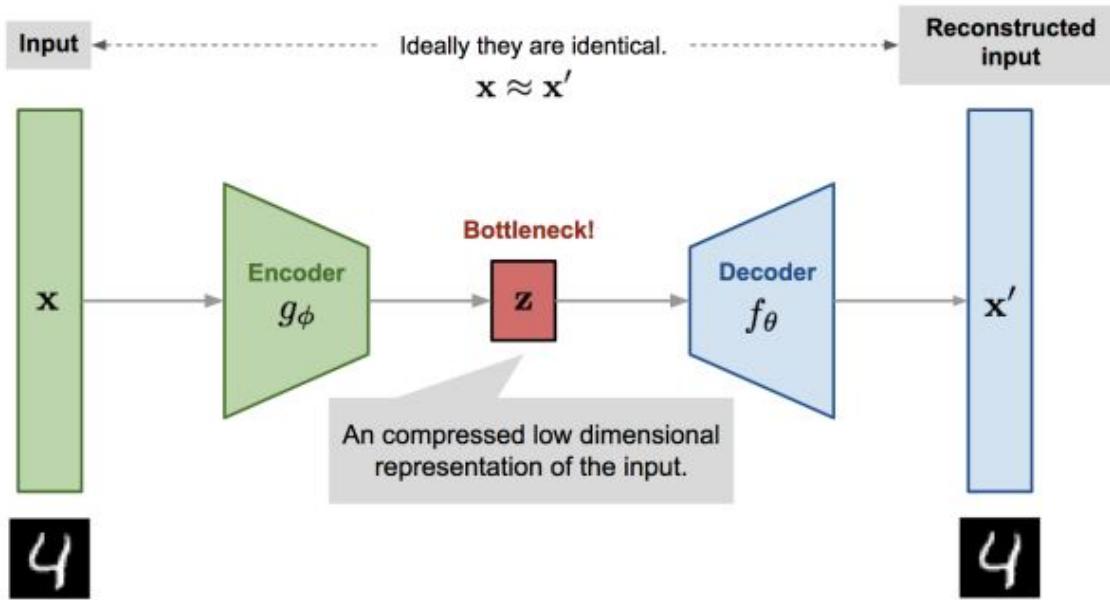


Figure 5.5: Schéma de fonctionnement d'un auto-encodeur [32]

Il existe beaucoup de métriques différentes pour quantifier la différence entre deux vecteurs, comme l'entropie croisée quand la fonction d'activation est une sigmoïde (une fonction de la forme $f(x) = \frac{1}{1-\exp(-\lambda x)}$). On peut aussi utiliser la fonction de coût des moindres carrés :

$$\mathcal{L}(\phi, \theta) = \frac{1}{n} \sum_{i=1}^n \|\mathbf{x}_i - f_\theta(g_\phi(\mathbf{x}_i))\|_2^2$$

Il existe une variante des auto-encodeurs qui sont dits "robustes". En effet il est possible qu'un auto-encodeur rencontre un problème de sur-apprentissage lorsqu'il y a plus de paramètres pour le réseau de neurones que de données. Une solution à ce problème peut-être de corrompre partiellement l'entrée en ajoutant du bruit ou en masquant aléatoirement certaines valeurs par exemple.

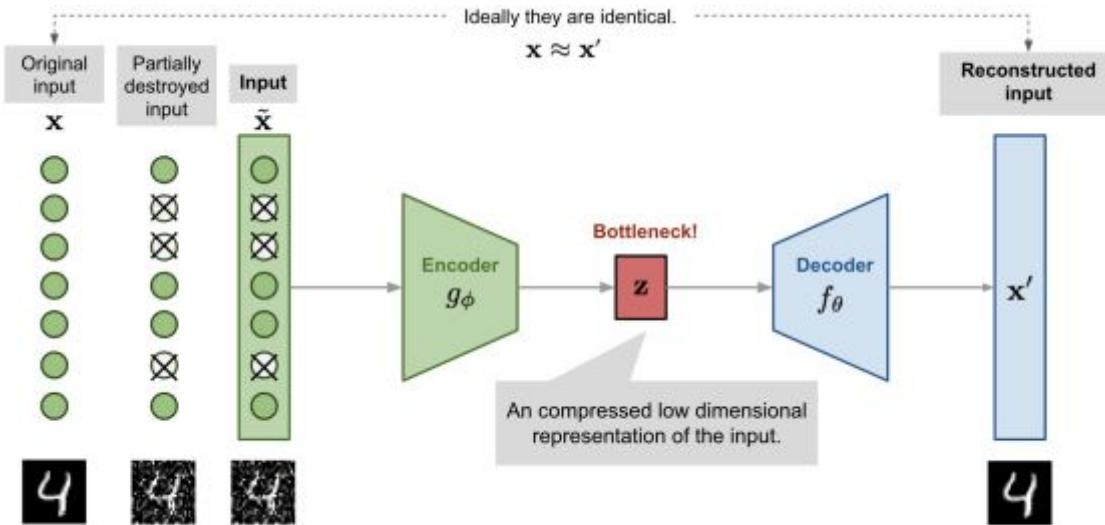


Figure 5.6: Schéma de fonctionnement d'un auto-encodeur robuste [32]

5.2 Clustering

Une autre application de l'apprentissage non supervisé est le clustering. On considère cette fois une base de données $\mathcal{D} = \{\mathbf{x}_i\}_{i=1}^n \in \mathbb{R}^d$ et on considère qu'il existe un jeu de K distributions C_k avec $k \in [|1, K|]$. On considère également que tous les \mathbf{x}_i sont une réalisation de l'un des C_k . Toutefois, dans le cas général, on ne dispose pas d'information sur K et les C_k .

Le but du clustering va alors être de déterminer le nombre de clusters (ou au moins avoir un nombre qui fait sens) et de répartir les données de \mathcal{D} sans avoir aucune information.

5.2.1 Algorithme K-means

Supposons que nous ayons choisi une valeur pour K . Construisons maintenant une nouvelle variable z_i avec $i \in [|1, n|]$, qui assigne à chaque \mathbf{x}_i un cluster.

$$\forall i \in [|1, n|], z_i = k \text{ si l'on assigne } \mathbf{x}_i \text{ à la classe } k$$

L'objectif de l'algorithme K-means peut être décrit ainsi :

$$\mathcal{L}(z, \mu) = \arg \min_{z, \mu} \|\mathbf{x}_i - \mu_{z_i}\|_2^2$$

$$\text{Avec } \mu_k = \frac{1}{|C_k|} \sum_{i \in C_k} \mathbf{v}_i$$

$$\text{Avec } C_k = \{\mathbf{x}_i, \forall i \in [|1, n|] | z_i = k\}$$

Ainsi, l'algorithme du K-means apprend les clusters de telle sorte que leur distance avec la moyenne du cluster (= le centroïde) ne soit pas trop loin. Pour faire simple le but est de faire en sorte de regrouper les données qui ressemblent à la valeur moyenne.

L'algorithme K-means

Entrée : \mathcal{D} , K le nombre de clusters, MaxIter le nombre d'itération, ϵ le seuil d'arrêt
 Initialiser chaque centroïde de cluster avec une valeur aléatoire

Répéter tant que l'on a pas atteint le seuil d'arrêt ou MaxIter :

- Assigner chaque observation au groupe avec le centroïde le plus proche.
- Recalculer les centroïdes à partir des individus rattachés aux groupes.
- Évaluer si la fonction de coût a atteint le seuil ϵ

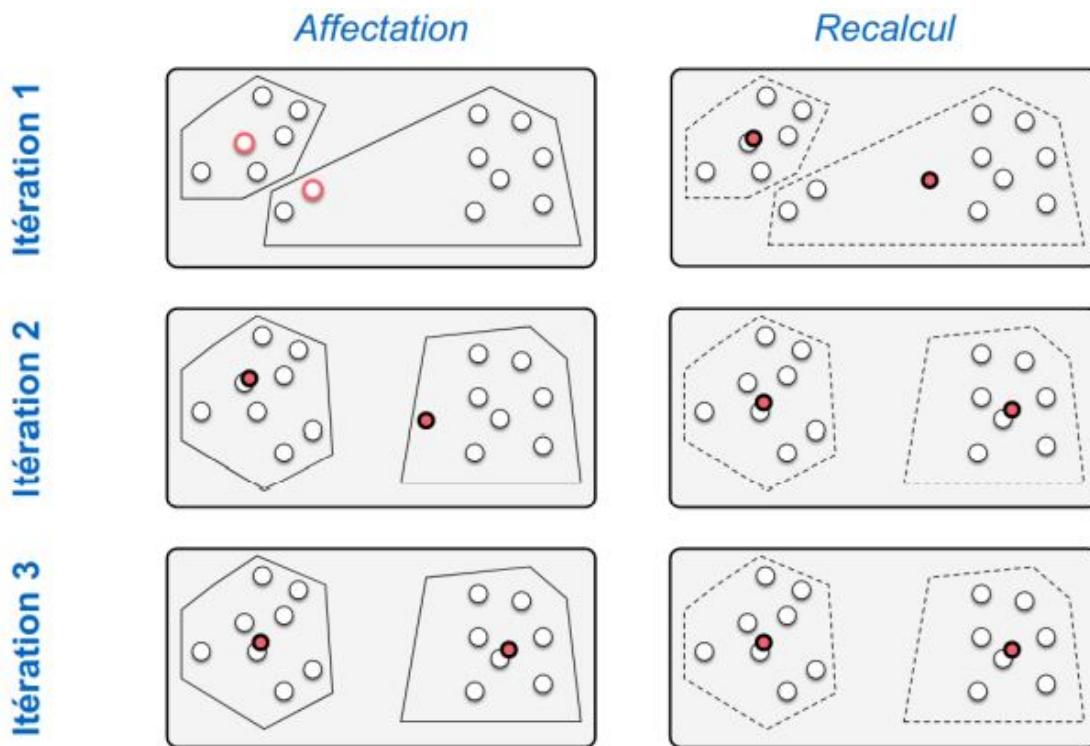


Figure 5.7: Exemple de cheminement de l'algorithme K-means [33]

L'algorithme K-means a l'avantage d'être facile à comprendre et à interpréter en plus d'être apte à s'adapter à des bases de données très grandes. En effet seules les coordonnées des centroïdes doivent être stockées en mémoire.

Cependant sa simplicité n'est pas synonyme de perfection et le temps de calcul peut être problématique puisque l'on traite plusieurs fois chaque donnée. De plus il n'y a pas de garantie que l'algorithme atteigne l'optimum global de la fonction de coût. Enfin, et peut-être le point le plus gênant, la solution sur laquelle débouche l'algorithme dépend des valeurs initialement choisies pour les centroïdes.

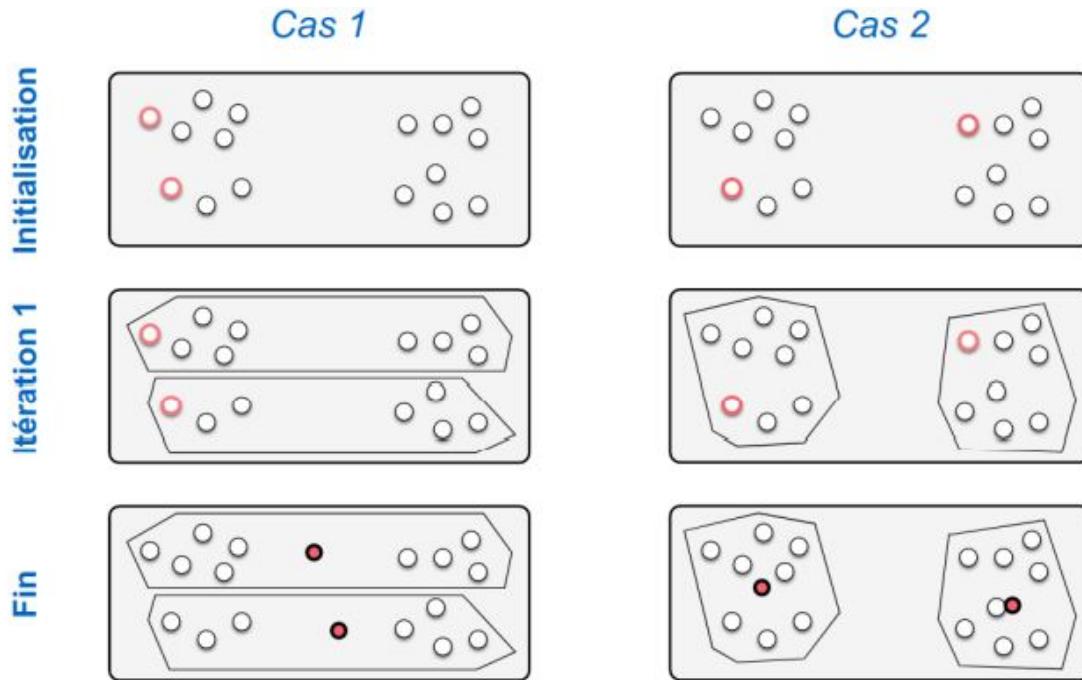


Figure 5.8: Exemple de mauvaise initialisation [33]

Nous avons ici illustré l'algorithme K-means avec la distance euclidienne, on peut néanmoins utiliser d'autre mesures de dissimilarités :

- La distance cosinus détermine l'angle entre les vecteurs formés par deux points avec l'origine :

$$d(\mathbf{x}, \mathbf{y}) = 1 - \frac{\mathbf{x}^T \cdot \mathbf{y}}{\|\mathbf{x}\| \cdot \|\mathbf{y}\|}$$

- La distance de Manhattan qui somme les différences entre chaque coordonnée des deux points :

$$d(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^d |x_i - y_i|$$

- La distance de Minkowski, aussi connue sous le nom de métrique de distance généralisée :

$$d(\mathbf{x}, \mathbf{y}) = \left(\sum_{i=1}^d |x_i - y_i|^{\frac{1}{p}} \right)^p$$

5.2.2 DBSCAN

Cette méthode part du constat que, dans un problème de clustering, les clusters correspondent aux régions où la densité de données est élevée. Ces clusters sont séparés par des zones où cette densité est faible. Un cluster est défini comme étant un jeu maximal de points connectés par densité. Cette méthode permet de découvrir un nombre arbitraire de clusters de n'importe quelle forme.

Pour pouvoir mesurer la densité d'un point il faut définir ce qu'est un voisinage. Le ϵ -voisinage d'un point x pour une distance d est l'ensemble des points situés à une distance inférieure à ϵ de x .

$$V_\epsilon(x) = \{y, d(x, y) \leq \epsilon\}$$

La méthode DBSCAN va, pour un ϵ et un entier $MinPts$, classer les points dans trois catégories distinctes. La première est appelée points cœur, un point est catégorisé comme tel s'il y a plus de $MinPts$ situés dans son ϵ -voisinage :

x est un point cœur si $V_\epsilon > MinPts$

Un point cœur est un point situé à l'intérieur d'un cluster. À partir de cette notion on peut définir l'accessibilité par densité. On dit qu'un objet p est directement accessible par densité à partir d'un objet q si ce dernier est un point cœur et que p est dans le ϵ -voisinage de q .

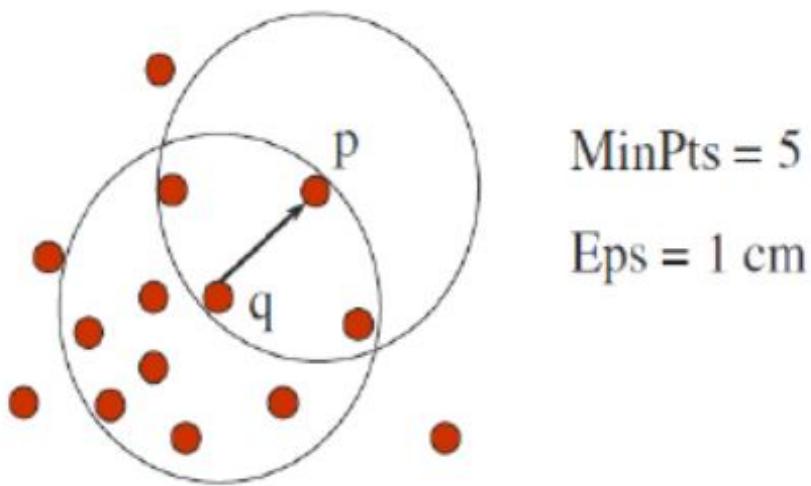


Figure 5.9: p est directement accessible par densité par q mais pas l'inverse car p n'est pas un point cœur [34]

On remarque que cette relation n'est pas symétrique. On dit qu'un point p est accessible par densité par un point q s'il existe une chaîne de points directement accessibles par densité qui relie q à p .

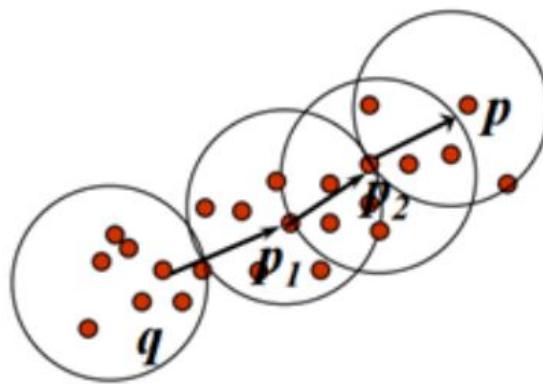


Figure 5.10: p est accessible par densité par q mais pas l'inverse car p n'est pas un point cœur [34]

De même cette relation n'est pas symétrique.

Comme évoqué plus tôt il existe deux autres types de points dans la méthode DBSCAN : les points de bordure et les points aberrants (anomalies). Les premiers sont définis comme étant ceux qui ont moins de $MinPts$ dans leur ϵ -voisinage, mais qui sont situés dans le voisinage d'un point cœur.

Les points aberrants sont simplement définis comme étant les points qui ne sont ni des points cœur ni des points de bordure.

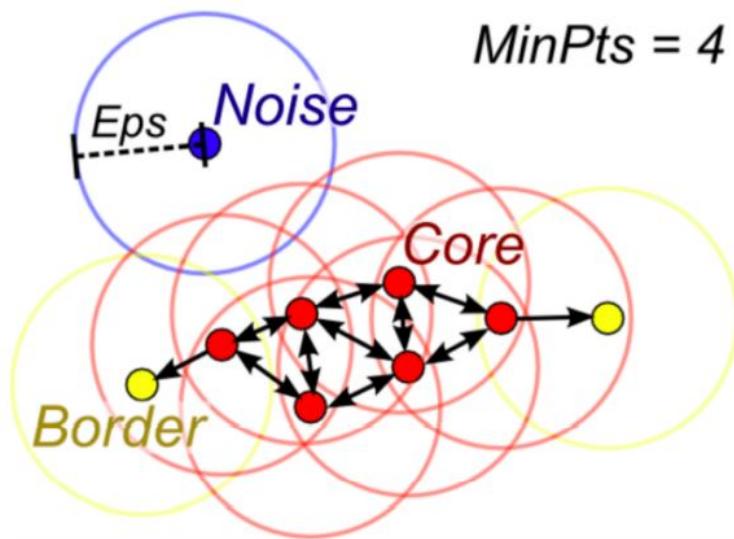


Figure 5.11: Représentation des 3 types de points dans la méthode DBSCAN [35]

Un cluster C est un sous-ensemble maximal de points x tel que tous les points de C sont connectés par densités deux à deux. On dit qu'un sous-ensemble est maximal si tout élément accessible par densité par un point de ce sous-ensemble appartient au même cluster.

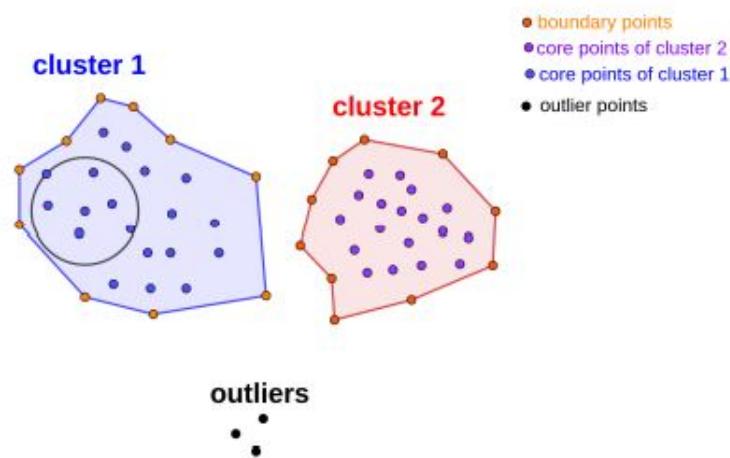


Figure 5.12: Exemple de résultat du DBSCAN

La méthode DBSCAN à l'avantage d'être moins sensible au bruit que l'algorithme K-means, qui peut très vite donner de mauvais résultat à cause de points aberrants et ne pas dépendre du nombre du cluster. Cependant les hyperparamètres de cette technique sont plus dure à maîtriser.

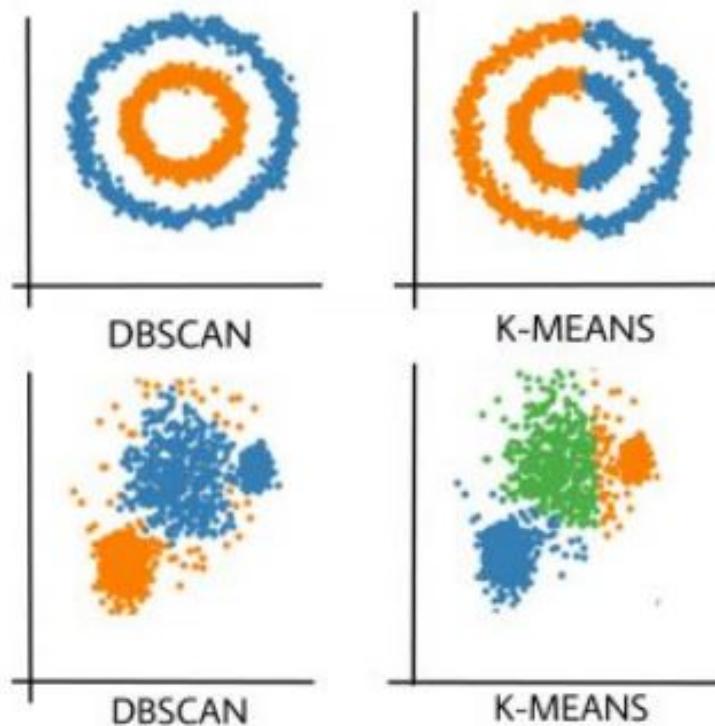


Figure 5.13: Comparaison des méthodes DBSCAN et K-means [36]

Bibliography

- [1] *Analyse des parties entre Kasparov et Deep Blue de 1997*. 1 (cit. on p. 7).
- [10] *k-nearest neighbors algorithm*. 10 (cit. on pp. 29, 30).
- [11] http://nsi.renoir.free.fr/term/arbre/cours_arbres.pdf. 11 (cit. on p. 34).
- [12] *machine-learning-an-overview*. 12 (cit. on p. 35).
- [13] *Introduction au Machine Learning par Chloé-Agathe Azencott*. 13 (cit. on pp. 36, 52, 53, 55, 63, 89).
- [14] https://fhernanb.github.io/libro_mod_pred/arb-de-regre.html. 14 (cit. on p. 40).
- [15] <https://python.plainenglish.io/adaboost-classifier-in-python-8d34a9f20459>. 15 (cit. on p. 42).
- [16] <https://medium.com/big-apps-tech/nlp-natural-language-processing-text-classification-workflow-take-n-grams-as-input-e549037b658d>. 16 (cit. on p. 43).
- [17] <https://medium.com/@ar.ingenious/applying-random-forest-classification-machine-learning-algorithm-from-scratch-with-real-24ff198a1c57>. 17 (cit. on p. 44).
- [18] <https://dustinstansbury.github.io/theclevermachine/bias-variance-tradeoff>. 18 (cit. on p. 45).
- [19] <https://www.csee.umbc.edu/courses/graduate/678/spring21/boosting.pdf>. 19 (cit. on p. 48).
- [2] *Article traitant de la victoire d'Alpha GO sur le champion du monde Ke Jie*. 2 (cit. on p. 7).
- [20] <https://www.mdpi.com/2079-6374/10/12/193>. 20 (cit. on p. 57).
- [21] <http://cedric.cnam.fr/vertigo/cours/ml2/coursSVMLineaires.html>. 21 (cit. on pp. 58, 60).
- [22] <https://www.slideserve.com/aimee/gradient-descent>. 22 (cit. on p. 62).
- [23] <https://www.slideshare.net/Paxcel/binary-and-multi-class-strategies-for-machine-learning>. 23 (cit. on p. 68).
- [24] <https://medium.com/@b.terryjack/tips-and-tricks-for-multi-class-classification-c184ae1c8ffc>. 24 (cit. on p. 69).
- [25] <https://durofy.com/machine-learning-introduction-to-the-artificial-neural-network>. 25 (cit. on p. 71).
- [26] <https://eszkola.pl/biologia/tkanka-nerwowa-i-tkanka-glejowa-3474.html>. 26 (cit. on p. 71).
- [27] http://ciml.info/dl/v0_99/ciml-v0_99-ch04.pdf. 27 (cit. on p. 78).
- [28] <https://www.youtube.com/watch?v=Uj3WGuJ1zao>. 28 (cit. on p. 80).
- [29] <https://arxiv.org/abs/1812.11118>. 29 (cit. on p. 81).
- [3] *Article traitant de la victoire d'AlphaStar*. 3 (cit. on p. 8).
- [30] <https://lucidar.me/fr/neural-networks/single-layer-gradient-descent/>. 30 (cit. on p. 83).
- [31] *The approximation of one matrix by another of lower rank par Carl Eckart et Gale Young*. 31 (cit. on p. 93).
- [32] <https://lilianweng.github.io/>. 32 (cit. on pp. 103, 104).
- [33] <https://www.irit.fr/ Yoann.Pitarch/>. 33 (cit. on pp. 105, 106).

- [34] <https://engineering.buffalo.edu/computer-science-engineering.html>. 34 (cit. on pp. 107, 108).
- [35] <https://elutins.medium.com/dbscan-what-is-it-when-to-use-it-how-to-use-it-8bd506293818>. 35 (cit. on p. 108).
- [36] <https://www.geeksforgeeks.org/>. 36 (cit. on p. 109).
- [37] <https://www.datasciencecentral.com/lenet-5-a-classic-cnn-architecture/>. 37 (cit. on p. 88).
- [38] <https://k21academy.com/microsoft-azure/convolutional-neural-network/>. 38 (cit. on p. 88).
- [39] <https://nasirml.wordpress.com/2019/01/08/convnet-in-tensorflow/>. 39 (cit. on p. 86).
- [4] *Article détaillant le fonctionnement d'Alpha GO Zero*. 4 (cit. on p. 10).
- [5] *The Elements of Statistical Learning* de T.Hastie, R.Tibshirani et J.Friedman. 5 (cit. on p. 11).
- [6] *Explications concernant les courbes ROC*. 6 (cit. on p. 14).
- [7] *Reconhecimento facial*. 7 (cit. on p. 16).
- [8] *Pattern Recognition and Machine Learning* de C.M.Bishop. 8 (cit. on pp. 18–20).
- [9] *PythonDataScienceHandbook*. 9 (cit. on pp. 22, 26, 27).