Principes de l'apprentissage statistique supervisé
(Supervised Machine Learning)

Romain COUILLET

Table des matières

Chapitre 1

Introduction à l'apprentissage machine : passé, présent et futur

1.1. Qu'est-ce que l'apprentissage machine ?

L'*apprentissage machine*, ou *apprentissage automatique*, est intrinsèquement lié au grand domaine de l'*intelligence artificielle*, et est d'ailleurs vu aujourd'hui comme la composante du *cœur mathématique et méthodologique* de l'intelligence artificielle. En termes simples, l'apprentissage machine consiste à la mise en place d'algorithmes capables d'apprendre "seuls" une règle de décision ou d'inférence à partir d'un jeu de données souvent compliqué à modéliser (et donc a priori non accessible à des modèles mathématiques rigoureux).

Le domaine de l'intelligence artificiel (IA) renvoie au grand public l'illusion de machines (d'ordinateurs) capables, sans aucun support humain, d'engranger des observations (stimulus) du monde extérieur, de les traiter d'une manière inaccessible à l'humain et de prendre des décision en conséquences. Cette vision "magique" de l'IA est évidemment excessive mais s'apparente néanmoins à certaines branches de l'apprentissage automatique (comme l'apprentissage par renforcement que nous évoquerons brièvement) qui consistent en l'implémentation d'algorithmes permettant à la machine de tester prendre d'elle-même des choix initialement aléatoires, d'en observer et d'en évaluer les conséquences, afin de faire des choix toujours plus pertinents (à savoir des choix maximisant un certain bénéfice) par la suite. Cependant, même dans ce cas où le contrôle humain est minimal, le fonctionnement interne de la machine repose sur des notions théoriques solides, telles que la théorie des jeux ou l'automatique, sans lesquelles aucune machine ne pourrait apprendre.

Nous allons voir dans ce cours, et c'est ici un choix délibéré, que l'apprentissage machine se rapproche, sinon parfois se confond, avec des théories mathématiques très formelles. C'est notamment le cas notamment des statistiques, des probabilités, du traitement du signal, de la théorie de l'information, etc. Nous n'aborderons évidemment pas tous ces liens, mais ferons en sorte de les mentionner au fil du cours. En lien avec cette connexion mathématique profonde, un autre choix délibéré du cours, certainement moins conventionnel que la plupart des cours introductifs à l'apprentissage automatique, est celui de tisser une *trame cohérente* qui relie de multiples sous-domaines de l'apprentissage. C'est ainsi notamment que nous verrons que, formellement, de nombreux algorithmes et méthodes d'apprentissage peuvent se ramener peu ou prou (ou être vus comme des extensions évidentes) à une simple régression linéaire ; et ce, qu'il s'agisse de classification supervisée par hyperplans séparateurs (nous le verrons à travers les machines à vecteurs de support) ou même de réseaux de neurones (nous le verrons dans le contexte des dits "extreme learning machines" ainsi que dans celui des réseaux de neurones "echo-state").

L'objectif de cette vision délibérée de ce cours introductif à l'apprentissage machine est multiple :

— en proposant une trame formelle consistante, tournant systématiquement autour de la notion de régression linéaire, on évite l'écueil des introductions

plus conventionnelles qui apportent plus une boîte à outils large et éclectique qu'une théorie sous-jacente solide ; par ce biais, il est bien plus simple de comparer et de comprendre le lien entre les différents algorithmes et outils que nous présenterons au cours de cette introduction au domaine ;
— la recherche moderne en apprentissage automatique, que nous évoquerons succinctement, suggère fortement que les méthodes les plus simples, si elles sont mieux comprise que ce n'est le cas actuellement, s'avèrent souvent être les plus performantes, les plus fiables, et les plus économiques en ressources (on rejoint d'ailleurs ici la vision physicienne du rasoir d'Occam : le monde physique est régi par des lois simples et se doit d'être compris par des mécanismes tout aussi simples) ; par ailleurs, la recherche moderne suggère également, suivant toujours ce principe de simplicité, que les algorithmes, même très compliqués, développés aujourd'hui ne traitent en somme les données (notamment celles de grandes dimensions, de plus en plus accessibles par les capacités d'échanges et de stockage actuels) que comme s'il s'agissait de simples grands vecteurs gaussiens.

1.2. Problème de base et vision d'ensemble du cours

Bien qu'ils ne soient pas clairement circonscrits, les algorithmes d'apprentissage automatique consistent en des mécanismes automatisés (pour nous des programmes ou algorithmes) permettant, à l'aide d'une source potentiellement très riche de données connues, d'effectuer une inférence (prendre une décision, classer, étiqueter, effectuer une régression) généralisable à toute nouvelle donnée ou tout nouvel ensemble de données jusqu'alors non connus. Nombre de ces problèmes se retrouvent déjà évidemment dans d'autres domaines scientifiques, on a déjà cité le traitement du signal ou les statistiques notamment ; en cela, l'apprentissage automatique n'a rien d'un domaine si isolé. Mais il est un de ces problèmes "phare" du domaine, et qui est souvent brandi en étendard de la discipline : la classification automatique et supervisée de données.

C'est en effet un exemple assez parlant qui permet de rassembler plusieurs notions au cœur de la discipline et qui se distingue (en tout cas partiellement, et nous tenterons de démontrer le contraire pendant ce cours) des autres disciplines des sciences des données. Plus emblématique encore est la classification d'images de "chiens" et d'images de "chats" en deux classes "classe chat" et "classe chien", illustrée en Figure 1.2.1. Dans le cadre de ce problème, l'algorithme que l'expert en apprentissage doit développer est contraint comme suit :

— un ensemble de données connues dites "d'entrainement" sont accessibles, à savoir un ensemble d'images de chiens et de chats ;
— ces images peuvent être complètement (dans le cas dit "supervisé"), partiellement (dans le cas dit "semi-supervisé") ou pas du tout (dans le cas dit "non-supervisé") préalablement étiquetées comme appartenant effectivement des images de chiens ou de chats (avec potentiellement des erreurs d'étiquetage mais nous n'élaborerons pas ce point) ;
— la sortie de l'algorithme consiste en une fonction capable, à l'aide des données d'entrainement, d'étiqueter le mieux possible toute nouvelle image (non connue) de chien ou chat.

La grande difficulté de ce problème réside dans l'extraction d'informations pertinentes issues des données brutes reçues par la machine (à savoir ici un ensemble de pixels associé à chaque donnée). Au contraire de nombreuses autres disciplines des sciences physiques, il est difficile de modéliser mathématiquement les concepts "image de chien" et "image de chat" ; il ne s'agit pas d'un signal, d'une onde, d'une

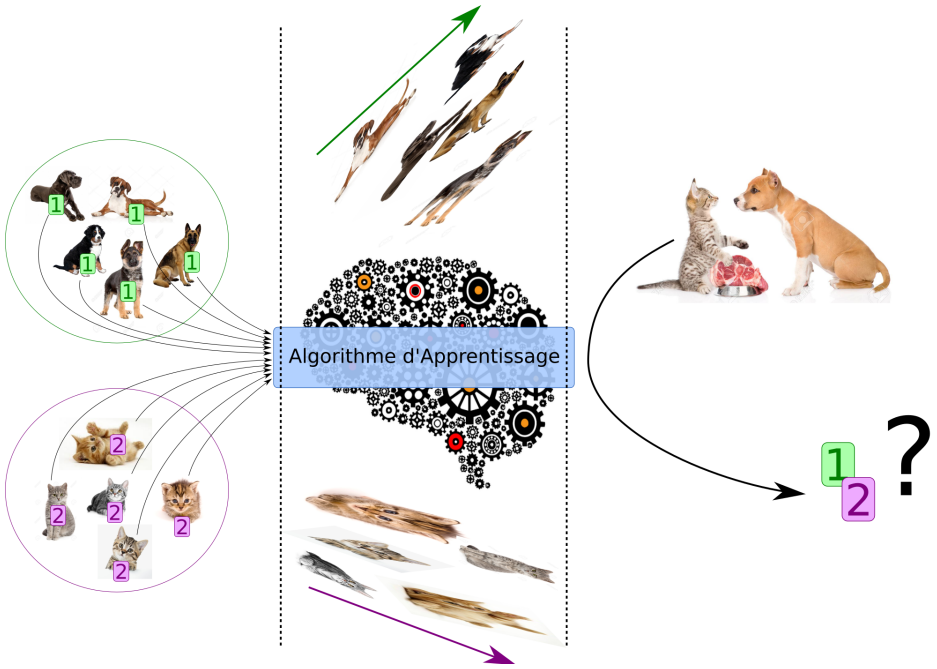

FIGURE 1.2.1. Représentation schématique du problème standard d'apprentissage de reconnaissance d'images de chiens et de chats.

mesure physique élémentaire qui pourraient entrer dans une théorie physique existante. Il s'agira donc d'être capable d'identifier manuellement ou automatiquement des *représentations* pertinentes (en anglais, on parle de "features") des concepts. Concrètement, dans le cadre d'images, on pourra chercher à identifier des couleurs, des contours, des angles, via les gradients de teintes (c'est notamment ce qui est fait dans la représentation HOG : histogram of oriented gradients), ou alors laisser la machine les découvrir par elle-même via un mécanisme d'apprentissage séquentiel (la machine teste des représentations au hasard, se trompe, corrige dans un sens qui diminue son erreur, et ce jusqu'à convergence vers une solution qu'on espère plutôt correcte : c'est ce qui se passe dans les puissants réseaux de neurones profonds modernes). Si $\mathbf{z}$ est une donnée brute (par exemple le vecteur des pixels d'une image), la représentation de $\mathbf{z}$ sera alors généralement vue comme un vecteur $\mathbf{x} = \phi(\mathbf{z})$ avec lequel l'algorithme travaillera en lieu et place de $\mathbf{z}$; la fonction ϕ associe ici de manière déterministe (ou parfois aléatoire) à chaque donnée sont vecteur de représentants.

Une question qui se pose cependant est de savoir différencier une bonne d'une mauvaise représentation ϕ, et de l'exploiter dans un cadre de décision de classification. Il existe différentes façons de voir les choses mais qui, comme nous le verrons, se ramènent essentiellement à la même idée de base : deux classes de représentants sont bien distinctes s'il est possible de créer un "plan" qui divise l'espace des (représentations des) données de sorte à ce que les données d'une même classe se retrouvent systématiquement du même côté du plan. C'est ce qui est illustré dans la Figure 1.2.2, et c'est surtout l'intuition qui est au cœur de la méthode fondamentale dite des "machines à vecteurs de support".

Nous allons très vite débattre, dans la section historique qui suit, de la distinction et des passes d'armes entre les réseaux de neurones, très populaires aujourd'hui, et les algorithmes fondateurs tels que les machines à vecteurs support, populaires

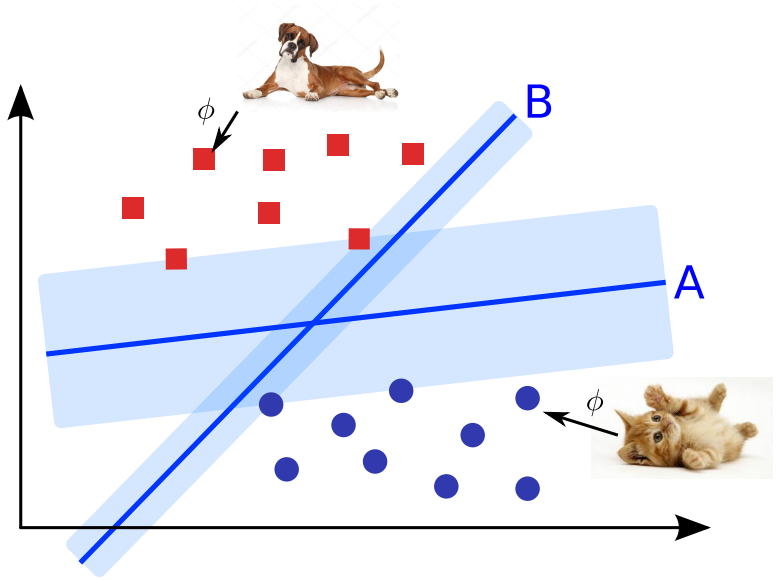

FIGURE 1.2.2. Représentation schématique de la classification par machines à vecteurs de support.

hier. Cependant, nous verrons au sein de ce cours que, si les deux approches sont d'apparence très distinctes, elles se rejoignent (et des ponts explicites peuvent même être établis) sur l'objectif de générer une fonction de représentation ϕ des données telles que les données $\mathbf{x} = \phi(\mathbf{z})$ se séparent très bien par un plan (en fait, plus correctement un hyperplan) dans l'espace des données. En cela, ces algorithmes ne sont pas si différents.

D'autre part, la notion d'hyperplan séparateur suppose qu'une partie de l'espace des données est associé à une classe (a une étiquette disons $+1$) tandis que l'autre est étiquetée -1. On sent donc, sous-jacent à l'algorithme de classification l'existence d'une fonction de décision "forte" qui associe brutalement $+1$ ou -1 à chaque donnée observée $\mathbf{x}$, et ce indépendamment de la plausibilité forte ou faible de l'attribution. Il est assez naturel d'imaginer qu'un bon mécanisme de classification va au contraire utiliser des règles de décisions plus "souples" en mettant des poids plus forts sur les données "de confiance" et plus faible sur les autres. On va donc très vite passer d'une classification binaire à une classification continue, et on entre alors très naturellement dans des questions, non plus de classification, mais de régression. Et, si la représentation des données est efficace (à savoir, il permet d'identifier un hyperplan séparateur clair), la classification se ramènera alors à un problème de régression linéaire.

Cette discussion établit un lien très naturel entre le problème généralement difficile de données brutes et celui très élémentaire de la régression linéaire, connue de nombreux domaines scientifiques. Même si le trait est volontairement poussé à l'extrême ici (l'apprentissage automatique ne se résume heureusement pas à une régression linéaire), il est néanmoins bon de garder cette vision en tête afin de mieux comprendre l'apport des mécanismes nombreux et parfois à première vue troublants du domaine.

Cependant, avant d'aller plus loin dans ces concepts et de débuter formellement le cours, il est instructif de positionner le domaine dans le contexte historique qui l'a vu naître, et surtout de comprendre les différentes phases de son évolution qui est loin d'avoir été un long fleuve tranquille.

1.3. Brève histoire de l'apprentissage automatique et intuitions de base

1.3.1. L'intelligence artificielle ou comment l'ordinateur modélise le cerveau. Il est difficile, et certainement peu pertinent, d'établir un point de départ à la théorie de l'apprentissage automatique. Néanmoins, l'avènement des premiers ordinateurs, et avant eux des premiers calculateurs mécaniques, qui ne sont rien d'autres que des "machines" répondant à un stimulus (une entrée) par une réponse (une sortie) associée, a donné lieu d'imaginer la création de machines aussi sophistiquées que le cerveau d'un être humain. Le lien est d'autant plus fort que la correspondance entre les unités de calcul (les transistors de l'ordinateur et les neurones du cerveau) et les ponts d'échanges (le câblage métallique et les axones et dendrites) est assez évidente. Seuls les ordres de grandeur (entre des dizaines de milliards de neurones contre seulement des millions de transistors) et la reconfigurabilité du réseau (flexible dans le cerveau, statique en électronique) diffèrent de manière fondamentale.

Dans les années 1950, le développement des ordinateurs et parallèlement les premières maigres avancées biologiques sur la compréhension du cerveau ont fortement motivé la recherche (et bien au delà, la société entière, à commencer par les romanciers de l'époque !) à imaginer dans un futur proche la reproduction d'une machine (d'un robot) capable de se suppléer – et même de dépasser les capacités – d'un être humain. La question posée est simple : à l'aide de silicium, transistors et connexions métalliques, est-on capable de mimer le système décisionnel humain tel qu'illustré en Figure 1.3.1 ? Derrière cette question se cache plusieurs problèmes successifs : (i) celui de la modélisation pertinente d'un réseau de neurones réaliste : nous allons l'évoquer tout de suite, et surtout (ii) celui du fonctionnement de l'apprentissage d'un réseau de neurones : ce problème est bien plus délicat et nous l'évoquerons dans une moindre mesure dans ce cours.

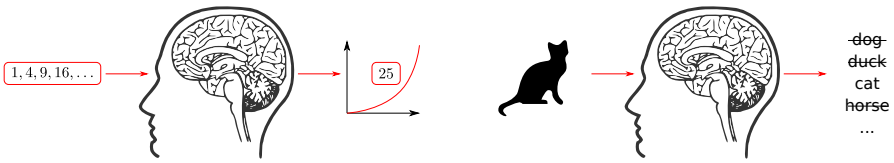

FIGURE 1.3.1. Représentation schématique d'une inférence par le cerveau.

1.3.1.1. *Le neurone.* Biologiquement, chaque neurone reçoit d'un ensemble de dendrites issues d'autres neurones un potentiel électrique qui, sommé au sein du neurone, s'il dépasse un seuil, dépolarise et propage alors le signal le long de l'axone qui part en direction d'autres neurones. Ceci est illustré dans la représentation de gauche de la Figure 1.3.2. L'élément-clé qui fait la puissance du neurone (du moins ce que nous en comprenons mathématiquement) réside dans la relation *non-linéaire* entre l'entrée et la sortie du neurone : en effet, un modèle simple du neurone consiste en une fonction de sommation (linéaire) $\sum_{i=1}^{p} w_i x_i$ des signaux $x_1, \dots, x_p$ reçus au niveau des p dendrites, possiblement pondérés par des coefficients d'intensité $w_1, \dots, w_p$, à laquelle on applique alors une fonction de seuillage $\sigma(t) : t \mapsto 1_{t>\text{seuil}}$; on obtient alors une sortie y associée, fonction non-linéaire des entrées, de la forme

$$y = \sigma\left(\sum_{i=1}^{p} w_i x_i\right).$$

C'est ce qui est représenté schématiquement sur la représentation de droite de la Figure 1.3.2.

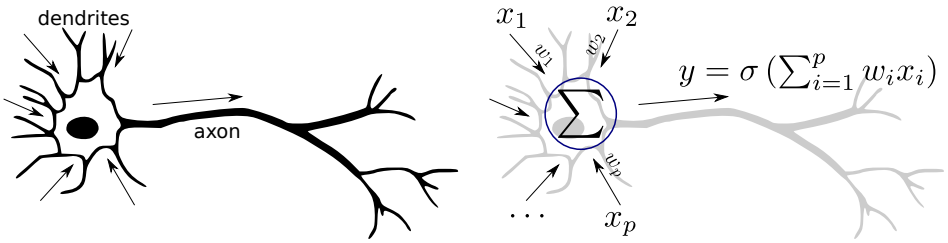

FIGURE 1.3.2. Représentation schématique d'un neurone et sa modélisation informatique.

1.3.1.2. *Le perceptron.* Évidemment, on ne modélise pas le fonction du cerveau au moyen d'un simple neurone mais d'un *réseau de neurones* interconnectés. Dans sa version la plus élémentaire, proposée initialement par Rosenblatt dans les années 1950, et nommé *perceptron*, un premier réseau de neurones consiste en deux *couches* de neurones : (i) la première couche contient les signaux d'entrée $x_1, \dots, x_p$ que l'on rassemblera en un vecteur $x = [x_1, \dots, x_p]^\intercal \in \mathbb{R}^p$, et (ii) la seconde consiste en N neurones activés parallèlement par les p entrées, et dont les sorties $y_1, \dots, y_N$ vérifient $y_j = \sigma(\sum_{i=1}^p w_{ji} x_i)$ où w_{ji} est le poids liant le neurone j à l'entrée x_i. En définissant également le vecteur $y = [y_1, \dots, y_N]^\intercal \in \mathbb{R}^N$, on obtient alors la forme vectorielle élégante

$$y = \sigma(Wx)$$

où $W = \{w_{ji}\}_{1 \le i \le p, 1 \le j \le N}$ et où nous utilisons un abus de langage assez classique, en supposant que $\sigma : \mathbb{R} \to \mathbb{R}$ est appliquée entrée-par-entrée au vecteur $Wx \in \mathbb{R}^N$. Cette configuration du perceptron de Rosenblatt est illustrée dans la partie haute de la Figure 1.3.3. En pratique, le fait d'avoir une seule fonction de seuillage réduit la flexibilité du réseau, et on ajoute communément une variable b_j supplémentaire au niveau de chaque neurone j, qui permet de déplacer le seuil, et donc d'obtenir plutôt $y_j = \sigma(\sum_{i=1}^p w_{ji} x_i + b_j)$, ou sous forme vectorielle $y = \sigma(Wx + b)$ avec $b = [b_1, \dots, b_N]^\intercal \in \mathbb{R}^N$. Ces variables, dites de *biais*, b_j peuvent être vues comme des poids supplémentaires attribués à une entrée x_0 fixe et égale à $x_0 = 1$ comme illustré dans la représentation centrale de la Figure 1.3.3.

Ce réseau élémentaire à deux couches est déjà de manière surprenante extrêmement riche. Il peut en effet s'apparenter à système de *représentations non-linéaires des données* qui entre au cœur de nombreuses méthodes d'apprentissage et que nous aurons longuement l'occasion d'évoquer pendant ce cours. En termes simples, imaginons que le vecteur $x \in \mathbb{R}^p$ soit l'ensemble des pixels d'une image de chien ou de chat et que notre objectif soit de construire une machine permettant de discriminer parfaitement les images de chiens et de chats : on cherche donc formellement à découvrir une fonction inconnue (et vraisemblablement complexe !) $f : \mathbb{R}^p \to \{-1, 1\}$ qui est telle que pour tout x image de chien, $g(x) = 1$ et pour tout x image de chat $f(x) = -1$. En définissant $\phi(x) = \sigma(Wx + b) \in \mathbb{R}^N$, nous créons un système de n représentants non-linéaires $y = \phi(x)$ pour chaque x, paramétré par les poids W et les biais b. Nous prétendons que ce système de représentation est extrêmement en ce sens qu'il a été prouvé dans les années 1980 qu'il est *universel* (ici le résultat est celui dû à Hornik en 1989) :

THÉORÈME 1. *Propriété d'approximation universelle Soit $f : \mathbb{R}^p \to \mathbb{R}$ une fonction (Borel) mesurable, μ une mesure de probabilité sur $\mathbb{R}^p$ et $\sigma : \mathbb{R} \to \mathbb{R}$ une fonction "sigmoïde" (à savoir croissante et telle que $\sigma(t) \underset{t \to -\infty}{\longrightarrow} 0$, $\sigma(t) \underset{t \to \infty}{\longrightarrow} 1$).*

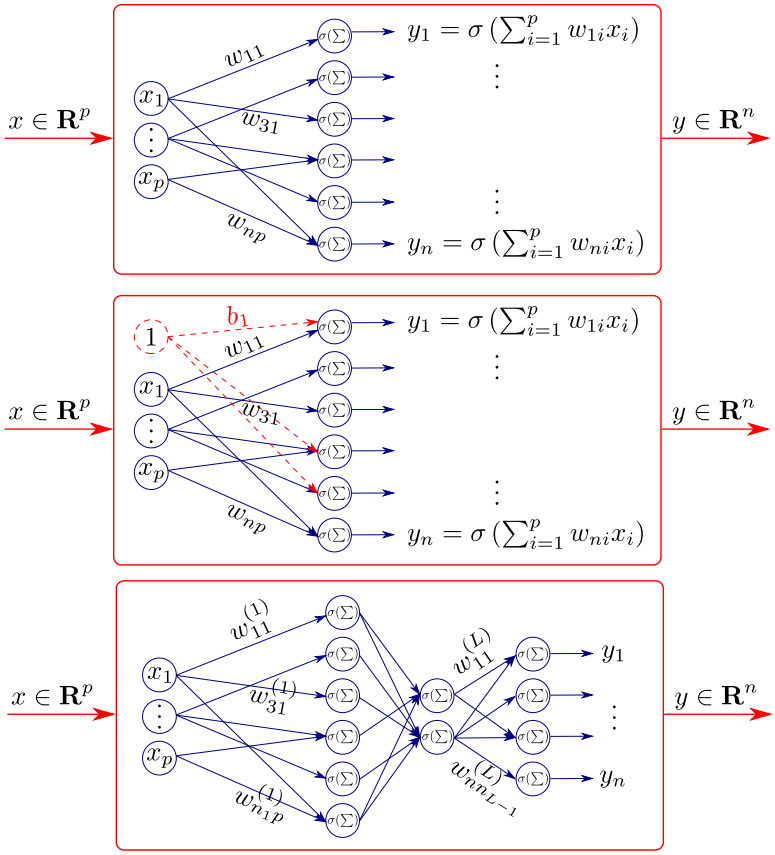

FIGURE 1.3.3. Le modèle du perceptron à une (en haut) et plusieurs (en bas) couches.

Définissons par ailleurs l'ensemble

$$\mathcal{G} \equiv \left\{g : x \mapsto a^\mathsf{T}\sigma(Wx+b),\ a,b \in \mathbb{R}^N,\ W \in \mathbb{R}^{p\times N},\ N \in \mathbb{N}\right\}$$

de toutes les combinaisons linéaires *possibles à la sortie de perceptrons à nombre arbitraire de neurones. Alors, pour tout $\varepsilon > 0$, il existe une fonction $g \in \mathcal{G}$ et un compact $K \subset \mathbb{R}^p$ tels que $\mu(K) > 1-\varepsilon$ et*

$$\mu\left\{x,\ |f(x)-g(x)| > \varepsilon\right\} < \varepsilon.$$

Le résultat du Théorème 1 est illustré en Figure 1.3.4.

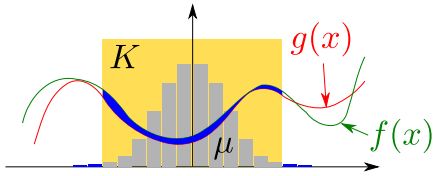

FIGURE 1.3.4. Représentation schématique du Théorème 1. Pour tout compact K qui contient une grande proportion du support de la mesure μ des données, il existe une fonction g issue de la classe $\mathcal{G}$ qui approxime arbitrairement bien la fonction objectif f.

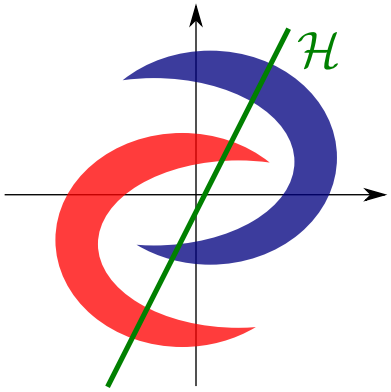

FIGURE 1.3.5. Deux classes de données de $\mathbb{R}^2$ (deux lunes imbriquées) qu'il est impossible de séparer par aucun hyperplan affine $\mathcal{H}$.

À la manière de l'approximation universelle de toute fonction mesurable sur un compact par des fonctions polynomiales (le fameux théorème de Stone-Weierstrass), le Théorème 1 dit concrètement que pour toute fonction objectif $f : \mathbb{R}^p \to \mathbb{R}$, il est possible de créer un perceptron (d'une certaine taille et avec des poids et des biais bien choisis) et donc une représentation $\phi(x)$ de chaque donnée x, de sorte qu'il existe une bonne combinaison linéaire $\sum_{i=1}^{N} a_i \phi_i(x)$ pour laquelle $\sum_{i=1}^{n} a_i \phi_i(x) \simeq g(x)$. Ce résultat a été fondateur pour la recherche sur les réseaux de neurones et démontre, contrairement à l'intuition actuelle, qu'il suffit d'une seule couche de neurones pour apprendre des tâches arbitrairement complexes.

Il est important d'insister ici sur l'importance du caractère *non-linéaire* de σ et donc des fonctions $\phi(x) = \sigma(Wx + b)$ disponibles. En effet, si $\sigma(t) = t$, l'ensemble $\mathcal{G} = \mathcal{G}(\sigma)$ du théorème se réduit alors à l'ensemble des fonctions affines $g : \mathbb{R}^p \to \mathbb{R}$, $x \mapsto w^\mathsf{T} x + b$ pour tous choix de (w, b). En remarquant que l'équation $(\mathcal{H})$: $w^\mathsf{T} z + b = 0$ définit un hyperplan affine de normale w, [1] et que le signe de $w^\mathsf{T} x + b$ décrit la position de x relative à cet hyperplan (le demi-espace "supérieur" ou "inférieur"), se restreindre à des fonctions σ linéaires revient à diviser l'espace des données par un hyperplan : la règle de décision quant à la classe d'appartenance de x ("chien" ou "chat") est donc réduite à l'indice du demi-espace d'appartenance de x. Ainsi, la classification n'est possible *que* lorsque les données "brutes" x sont séparables par un hyperplan affine, et ce n'est pas toujours possible, comme le montre l'exemple populaire des deux lunes imbriquées de la Figure 1.3.5. C'est en cela que le caractère non-linéaire de σ est essentiel, et c'est paradoxalement ce même caractère non-linéaire qui pose tant de difficultés à la théorie des réseaux de neurones...

Car en effet, pour entamer l'ensemble des difficultés que connaissent les réseaux de neurones depuis leur création, le Théorème 1 n'est pas constructif : il ne dit pas, pour un problème donné, comment choisir la structure du réseau (quelle taille n prendre ?) et encore moins quels sont les poids W et biais b appropriés. Cela pourrait sembler être un détail technique que trente ans de recherche a su depuis combler : ce n'est pas le cas, et nous sommes toujours aujourd'hui incapable de construire le perceptron universel prévu par Hornik. Néanmoins, nous avons vite compris que le nombre typique de neurones n nécessaires à l'approximation d'une fonction très complexe g (dans le cas binaire où $g(x) \in \{\pm 1\}$, par complexe, on entend

1. Rappelons en effet qu'un hyperplan affine de $\mathbb{R}^p$ est défini par un vecteur normal $u \in \mathbb{R}^p$ et par un point z_0 de l'hyperplan : l'ensemble des $z \in \mathbb{R}^p$ de l'hyperplan sont tels que $w^\mathsf{T}(z - z_0) = 0$ ou de manière équivalente $w^\mathsf{T} z + b = 0$ avec $b = w^\mathsf{T} z_0$.

une fonction qui divise l'espace d'entrée $\mathbb{R}^p$ de manière extrêmement "tordue", non connexe, etc.) peut être gigantesque et inapproprié en pratique.

1.3.1.3. *Le populaire réseau "feedforward".* Il a vite été intuité, et on s'éloigne malheureusement déjà ici de socles mathématiques formels (un grand problème en réseaux de neurones, nous aurons l'occasion de le réévoquer), qu'il est pertinent d'étendre le perceptron de Rosenblatt à ce qu'on appelle le *perceptron à plusieurs couches* ; à savoir, un modèle de réseau de neurones contenant plusieurs couches successives de perceptrons élémentaires. Ceci est illustré par la représentation du bas de la Figure 1.3.3. Intuitivement, on se dit ici que la capacité d'une couche de neurones à "linéariser" l'espace ambiant des données est un outil puissant qui, opéré plusieurs fois à travers des couches successives, est plus puissant (a un pouvoir plus "déformant" de l'espace ambiant) que celui d'un simple perceptron à deux couches. Du point de vue de l'ordre de grandeur, on raisonne qu'une augmentation linéaire du nombre de couches du réseau équivaut vraisemblablement à une augmentation exponentielle du nombre de neurones d'un seul perceptron.

C'est ainsi qu'on est passé rapidement du perceptron de Rosenblatt à ce qu'on appelle aujourd'hui un *réseau feedforward* à plusieurs couches, comme illustré dans la représentation inférieure de la Figure 1.3.3. Dans cette représentation feedforward, on réplique simplement l'opération de mélange linéaire par une matrice W et l'application de la fonction d'activation σ plusieurs fois en série, de sorte qu'après ℓ couches (ℓ étant utilisé ici pour *layers*), on obtient la représentation $x^{(\ell)} \in \mathbb{R}^{N_\ell}$ définie par récurrence par les relations :

$$x^{(\ell)} = \sigma(W^{(\ell)} x^{(\ell-1)} + b^{(\ell)})$$
$$x^{(0)} = x$$

où $W^{(\ell)} \in \mathbb{R}^{N_\ell \times N_{\ell-1}}$, $b^{(\ell)} \in \mathbb{R}^{N_\ell}$ sont la matrice de connexion et le vecteur de biais de la couche $\ell - 1$, de taille $n_{\ell-1}$, à la couche ℓ, de taille n_ℓ, avec pour convention $N_0 = p$.

L'aspect "feedforward" est en fait à mettre ici en opposition à une autre généralisation du perceptron de Rosenblatt, toujours inspirée de la biologie, qui est celui des *réseaux récurrents*. En effet, on sait naturellement que les réseaux de neurones biologiques sont extrêmement interconnectés (bien plus qu'un réseau de transistors dans une microprocesseur). La généralisation "par défaut" du perceptron de Rosenblatt se devrait d'être celle d'un réseau similaire au perceptron à deux couches mais dans lequel les neurones, au lieu de recevoir des stimuli de l'entrée x seule, recevraient d'autres stimuli de leurs neurones voisins. Formellement, on s'attendrait donc plutôt à une modélisation *dynamique* du type :

$$y_t = \sigma(W x_t + W_{\text{res}} y_{t-1} + b)$$

où x_t est l'entrée reçue par le réseau à un certain instant t, y_{t-1} la sortie au temps précédent $t - 1$ et $W_{\text{res}} \in \mathbb{R}^{N \times N}$ est la matrice d'interconnexion du réseau. Cette dernière est notée ici W_{res} en référence au domaine du *reservoir computing* qui traite de certaines formes de réseaux de neurones récurrents (on doit comprendre ici par la notion de "réservoir" celle d'un réservoir de mémoire). Nous aurons l'occasion au cours de ce module de revenir sur ces réseaux récurrents de type "réservoir".

Les réseaux récurrents sont cependant très compliqués à implémenter et sont notamment très instables numériquement. L'aspect de "réservoir" y est pour beaucoup : en accumulant des données, "l'énergie" du réservoir peut augmenter au point de rendre le réseau instable et, afin de se prémunir contre cette augmentation d'énergie, le plus simple est malheureusement de faire en sorte d'évacuer rapidement l'énergie accumulée, de sorte que la mémoire du réseau s'évapore exponentiellement vite (avec le temps t). Théoriquement, ces réseaux sont d'ailleurs toujours

mal compris aujourd'hui, et assez peu étudiés, sauf lorsque l'on traite de données issues de séries temporelles : à savoir, lorsque les $x_t \in \mathbb{R}^p$ entrant successivement dans le réseau sont issus de séries temporelles complexes à analyser, **telles que les séries temporelles en finance**. Nous reviendrons sur ce cas spécifique à la fin du cours.

Les réseaux de neurones artificiels utilisés le plus couramment dans la littérature présente et passée sont donc à défaut de mieux des réseaux feedforward. Leur succès récent, que nous réévoquerons dans une section à venir, intervient notamment dans le domaine de la vision par ordinateur (*computer vision*). Fait étrange, mais qui a motivé la recherche à poursuivre dans cette voie, il a en fait été établi que les réseaux de neurones du cortex visuel de mammifères sont en fait stratifiés précisément à la manière d'un réseau de neurones feedforward, comme cela est illustré en Figure 1.3.6, ici chez le singe. Plus fort encore, il a été établi que les premières couches des réseaux de neurones du cortex visuels fonctionnent précisément comme celui d'un réseau de neurones feedforward moderne : à savoir, les premières couches se contentent d'identifier des angles, contours et formes. Il est également établi que ces réseaux "purement visuels" (on parle bien ici de vision et non d'interprétation des images observées) se reconfigurent structurellement assez peu. A contrario, les réseaux de neurones des aires de la mémoire s'organisent comme des réseaux naturellement très interconnectés (et donc récurrent) avec une spécificité notable qui isole la mémoire à court terme (qui se reconfigure peu) à la mémoire à long terme (qui se renforce par de nouveaux liens, notamment pendant la nuit qui permet de consolider les informations acquises le jour).

1.3.1.4. *Du modèle à l'apprentissage.* L'apprentissage des réseaux de neurones, comme tout algorithme d'apprentissage dit *supervisé* s'effectue à l'aide d'un ensemble d'exemples de données, appelés collectivement *base d'entrainement*, à savoir des vecteurs $x_1, \dots, x_n \in \mathbb{R}^p$, dont les sorties résultantes sont connues. Dans le cadre de la classification binaire est attribuée à chaque donnée x_i une étiquette $y_i \in \{\pm 1\}$. Comme les degrés de libertés d'un réseau de neurone sont les valeurs prises par ses couches linéaires de connexion $W^{(1)}, \dots, W^{(L)}$ ainsi que ses biais $b^{(1)}, \dots, b^{(L)}$ (dans le cadre d'un réseau à L couches), il s'agit donc de déterminer ces matrices et vecteurs de sorte que la sortie

$$x^{(L)} = \sigma(W^{(L)}\sigma(W^{(L-1)} \cdots x + b^{(1)}) + b^{(L)})$$

soit le plus proche possible de la valeur de l'étiquette souhaitée. Pour cela, comme les couples (x_i, y_i) sont connus, il s'agira mathématiquement de minimiser la distance entre la sortie obtenue $x_i^{(L)}$ et la sortie souhaitée y_i en fonction de tous les paramètres du problème.

C'est ici que les choses se compliquent fortement. Pour commencer, indépendamment du choix de la fonction de coût à minimiser, du fait des fonctions non-linéaires σ et de l'empilement des couches du réseau, le problème est hautement non-linéaire. Il peut même être établi que le nombre de minima locaux et points selles de ce qu'on appelle le *paysage énergétique* (*energy landscape*) du réseau augmente exponentiellement avec la taille du réseau. Il devient dès lors impossible d'espérer trouver le minimum global et, pire, en partant d'un ensemble d'initialisation arbitraire des poids $W^{(1)}, \dots, W^{(L)}$ et biais $b^{(1)}, \dots, b^{(L)}$, on court naturellement le risque de trouver un mauvais minimum local, associé à de très mauvaises performances.

Mais le problème est en fait encore plus compliqué qu'il n'y parait car, assez étonnamment, et c'est ici un point que nous traiterons dans les notions clés du début du cours, il *n'est pas forcément souhaitable* de découvrir le minimum global ou un minimum trop profond, qui pourrait donner lieu au phénomène dit de

"surapprentissage". Dans ce contexte, le réseau a si bien "forcé les poids" à lier les données d'entrainement x_i à leurs sorties espérées y_i que la fonction d'approximation résultante est très erratique (pensez à un polynôme d'ordre très élevé utilisé pour interpolé un jeu arbitraire de quelques points), ce qui a pour conséquence que toute nouvelle donnée non étiquetée évaluée par le réseau risque d'engendrer des résultats très erronés. Ce n'est évidemment pas souhaitable, car on cherche avant tout à rendre le réseau capable de *généralisation* à des données inconnues.

Cette faille majeure dans la stabilité et la maitrise des réseaux de neurones ajouté au manque cruel (et à l'importante frustration tant intellectuelle que pratique) de compréhension théorique de leur comportement a suscité l'effondrement de l'intérêt scientifique pour ces méthodes dans les années 1990. C'est à ce moment là qu'un outil nouveau, celui des *machine à vecteurs de support* a révolutionné le domaine de l'apprentissage automatique, et a rapidement passé les réseaux de neurones dans l'oubli.

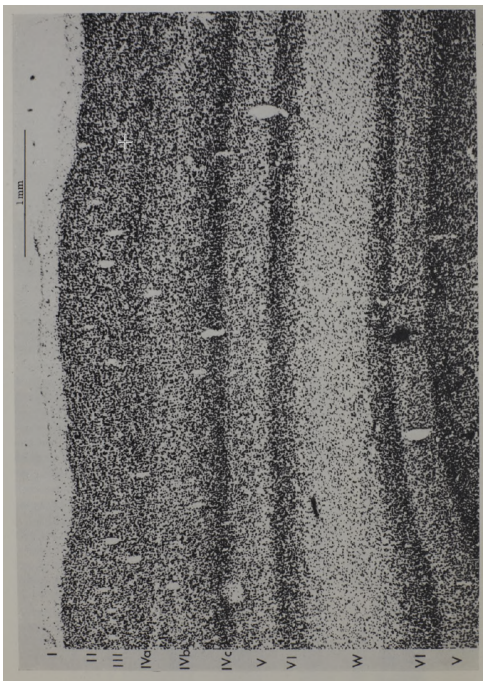

FIGURE 1.3.6. Coupe cérébrale du cortex visuel d'un primate, clairement réparti en couches successives de neurones (zones denses).

1.3.2. Machines à vecteurs de support : le retour en force des mathématiques.

1.3.2.1. *Les SVM : simplicité et liens avec la régression.* Les machines à vecteurs de support (*support vector machines*, ou SVM, en anglais) partent de l'idée simple, déjà évoquée précédemment, selon laquelle un algorithme de classification binaire efficace parviendra à séparer l'espaces des données en deux sous-espaces associés à chacune des classes. Le concept se généralise évidemment à des classifications à plus de deux classes mais nous allons déjà nous poser sur celui de deux classes.

Comme nous l'avons déjà évoqué, si un bon système de représentation $\phi(x)$ est découvert, on a espoir que les données $\phi(x_1), \ldots, \phi(x_n)$ de la base d'entrainement soient séparables dans leur espace ambiant par un hyperplan affine (qui ne passe

pas forcément par l'origine). Disons pour faire simple que les x_i eux-mêmes sont en effet *linéairement séparables* ; en pratique, on l'a déjà vu dans ce cours, les données x_i subissent souvent des transformations et on est généralement amené à travailler avec des représentants qu'on peut, pour se simplifier la vie et les notations, tout simplement renommer x_i. Dans ce cas, il existe au moins un hyperplan affine qui isole les deux classes de données, mais en général on en trouve plutôt une infinité. Ceci est illustré dans le dessin supérieur de la Figure 1.3.7 dans une configuration où l'espace ambiant est de dimension 3 et l'hyperplan bleu est l'un des multiples hyperplans séparateur de 11 vecteurs x_i de la base d'entrainement de $\mathbb{R}^3$.

La spécificité des machines à vecteurs de support consiste à sélectionner parmi ces hyperplans l'unique hyperplan qui est "optimal" au sens de la maximisation de la plus petite distance aux points x_i. En d'autres termes, il s'agit de choisir, non pas un, mais trois hyperplans affines $\mathcal{H}_0 : \ (w^\star)^\mathsf{T} z + b = 0$, $\mathcal{H}_1 : \ (w^\star)^\mathsf{T} z + b = \delta$ et $\mathcal{H}_{-1} : \ (w^\star)^\mathsf{T} z + b = -\delta$ de même normale $w^\star$ et tels que $\mathcal{H}_0$ est équidistant de $\mathcal{H}_1$ et $\mathcal{H}_{-1}$, sous la contrainte de maximiser la *marge* $2\delta > 0$ qui sépare les hyperplans pour une norme de $w^\star$ donnée. Ainsi, on maximise la distance entre les vecteurs x_i de la base d'apprentissage à l'hyperplan affine séparateur $\mathcal{H}_0$ qui est, en ce sens, vu comme optimal.

Bien évidemment, pour que les hyperplans soient optimaux, l'hyperplan $\mathcal{H}_{-1}$ doit au moins contenir un vecteur x_i étiqueté $y_i = -1$ et l'hyperplan $\mathcal{H}_1$ doit au moins contenir un vecteur x_i étiqueté $y_i = 1$, sinon on pourrait augmenter encore la marge 2δ. Ainsi, un certain nombre de points x_i se trouvent contenus dans les hyperplans externes et permettent en fait à eux seuls de décrire l'hyperplan (un dessin en deux ou trois dimensions permet de s'en rendre compte) : ces points sont nommés *vecteurs de support* (des hyperplans) et doivent leur nom aux machines à vecteurs de support. Tout ceci est illustré dans la partie inférieure de la Figure 1.3.7.

Dans cette description, il est important de contraindre la norme de $w^\star$ car évidemment l'équation $(w^\star)^\mathsf{T} z + b = 0$ est équivalente à $(\alpha w^\star)^\mathsf{T} z + \alpha b = 0$ pour tout $\alpha \in \mathbb{R}$, ce qui laisse une certaine latitude sur la définition du couple $(w^\star, b)$: ne pas contraindre la norme de $w^\star$ ou la valeur de b ne donne alors plus beaucoup de sens à la maximisation de δ dans le problème précédent. Ce point est très fondamental car le problème d'optimisation classique des machines à vecteurs support ne maximise pas δ mais plutôt, et c'est en effet équivalent, impose $\delta = 1$ et cherche alors à minimiser la norme de $w^\star$. Précisément, le problème d'optimisation, de la dite *machine à vecteurs de support linéaire "dure"*, revient alors à la formulation *primale* suivante :

$$(w^\star, b) = \operatorname{argmin}_{(w,b)} \frac{1}{2}\|w\|^2$$

sous la contrainte

$$y_i(w^\mathsf{T} x_i + b) \geq 1, \ \forall i \in \{1, \dots, n\}.$$

Deux choses sont à noter ici avant d'aller plus loin : (i) en imposant la demi-marge $\delta = 1$, on cherche bien à avoir $w^\mathsf{T} x_i + b \geq 1$ pour les données x_i de la classe 1 (donc d'étiquette $y_i = 1$) et $w^\mathsf{T} x_i + b \leq -1 \Leftrightarrow -[w^\mathsf{T} x_i + b] \geq 1$ pour les données x_i de la classe -1 (donc d'étiquette $y_i = -1$), ce qui explique la formulation commune de la contrainte pour tout x_i de la base d'entrainement ; (ii) la formulation apparemment étrange $\operatorname{argmin}_{(w,b)} \frac{1}{2}\|w\|^2$ qui est évidemment équivalente à $\operatorname{argmin}_{(w,b)} \|w\|$ permet en fait d'anticiper une résolution par multiplicateur de Lagrange (nous le verrons plus tard dans le détail) qui va consister à dériver le lagrangien associé au

problème

$$\mathcal{L}(\alpha) = \frac{1}{2}\|w\|^2 - \sum_{i=1}^{n} \alpha_i (y_i(w^\mathsf{T} x_i + b) - 1)$$

pour les multiplicateurs $\alpha = [\alpha_1, \ldots, \alpha_n]^\mathsf{T} \geq 0$ et dont les dérivées partielles par rapport aux entrées de w et b donnent la solution

$$w^\star = \sum_{i=1}^{n} \alpha_i y_i x_i = X D_y \alpha$$
$$0 = \sum_{i=1}^{n} \alpha_i y_i = \alpha^\mathsf{T} y$$

où $X = [x_1, \ldots, x_n] \in \mathbb{R}^{p \times n}$ et $D_y = \mathrm{diag}(y_1, \ldots, y_n)$. Comme le problème initial est convexe en w (c'est un problème quadratique sous contraintes linéaires), il peut se résoudre par dualité forte en maximisant le problème dual (en remplaçant $w^\star$ dans le lagrangien) :

$$\begin{aligned}\alpha^\star &= \mathrm{argmax}_{\alpha \geq 0} -\frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j y_i y_j x_i^\mathsf{T} x_j + \sum_{i=1}^{n} \alpha_i \\ &= \mathrm{argmax}_{\alpha \geq 0} -\frac{1}{2} \alpha^\mathsf{T} D_y X^\mathsf{T} X D_y \alpha + \alpha^\mathsf{T} 1_n\end{aligned}$$

avec contrainte d'égalité (et non plus d'inégalité)

$$0 = \sum_{i=1}^{n} \alpha_i y_i = \alpha^\mathsf{T} D_y 1_n.$$

Que ce soit sous la forme primale ou duale, on a affaire à un problème d'optimisation quadratique sous contrainte d'égalités et inégalités linéaires, qui se résout numériquement efficacement par *programmation quadratique*. Outre le fait que les inégalités soient plus simples ($\alpha_i \geq 0$) dans la forme duale, le réel gain de cette forme est dans son interprétation. Une fois obtenu $\alpha^\star$, remarquons que l'hyperplan optimal est donné par $w^\star = \sum_{i=1}^{n} \alpha_i y_i x_i$, à savoir une combinaison linéaire des vecteurs $y_i x_i$ de la base de donnée d'apprentissage. Par ailleurs, du fait des contraintes du primal $y_i(w^\mathsf{T} x_i + b) \geq 1$, il est évident que la solution $w^\star$ va induire des cas d'égalité $y_i((w^\star)^\mathsf{T} x_i + b) = 1$ pour certains indices i (car sinon on pourrait trouver un autre $w^\star$ qui d'énergie moindre qui assurerait encore toutes les contraintes, ce qui est impossible par définition de l'optimum). Par la théorie de la dualité, ces cas d'égalité vont être associés aux *seuls* α_i non nuls du vecteur α (vu qu'on paye un prix nul pour ces α_i dans $\mathcal{L}(\alpha)$ alors que les autres "coûtent"). On a donc finalement pour solution

$$w^\star = \sum_{i | y_i((w^\star)^\mathsf{T} x_i + b) = 1} \alpha_i y_i x_i$$

qui constitue une combinaison linéaire des seuls vecteurs x_i *appartenant aux hyperplans* $\mathcal{H}_1$ *et* $\mathcal{H}_{-1}$: on les appelle les *vecteurs de support* et donnent leur nom à l'algorithme. Pour trouver b, on utilisera le fait que $y_i((w^\star)^\mathsf{T} x_i + b) \geq 0$ avec égalité pour ces vecteurs de support. Notons qu'en général ces vecteurs de support sont peu nombreux (ils sont de l'ordre de la dimension p de l'espace et pas du nombre n de données), ce qui permet d'"encapsuler" l'algorithme SVM final en ne tenant compte que de ces quelques vecteurs, et d'éliminer les autres vecteurs de la base d'apprentissage de l'algorithme : numériquement, cela est avantageux et bien moins coûteux que d'autres méthodes alternatives qui utiliseront plus systématiquement toutes les données d'apprentissage.

Ainsi, pour toute nouvelle donnée x non étiquetée, la décision d'appartenance à une classe sera donnée par le test :

$$g(x) = (w^\star)^\mathsf{T} x + b \underset{\mathcal{H}_{-1}}{\overset{\mathcal{H}_1}{\gtrless}} 0.$$

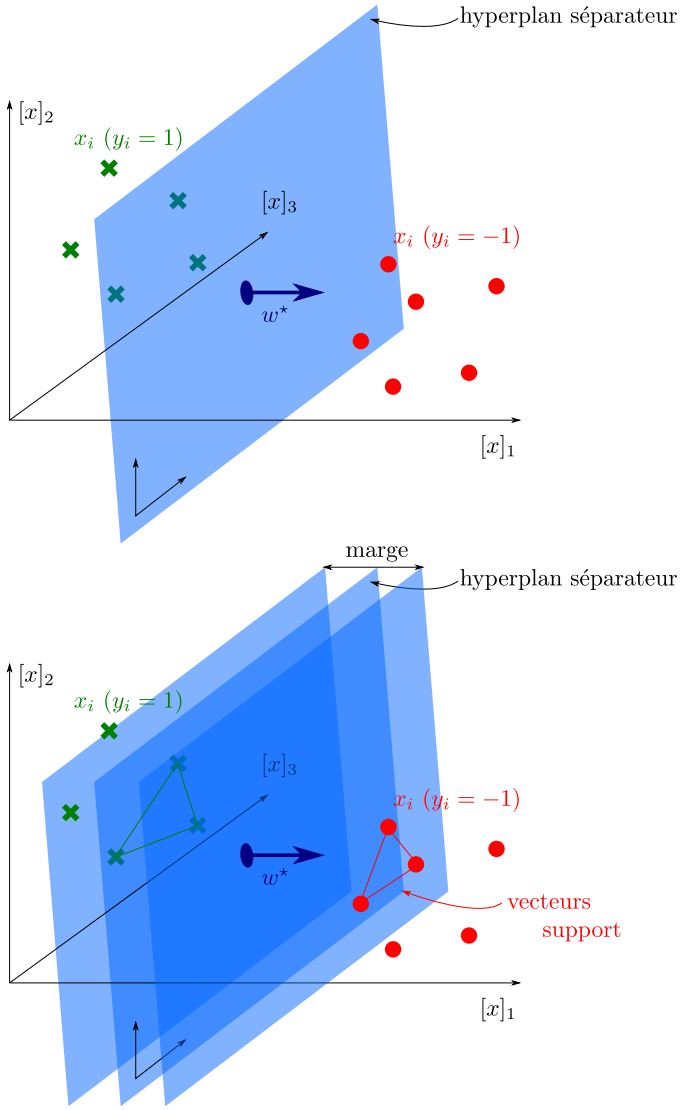

FIGURE 1.3.7. Visualisation du concept de machines à vecteurs support.

Il est intéressant de remarquer ici que nous avons affaire, de manière un peu détournée, à un problème de *régression*. En effet, il s'agit ici de déterminer un vecteur $w^\star$ qui permet d'assurer que chaque projection $w^\mathsf{T} x_i + b$ "est du signe souhaité". De là, il n'y a qu'un pas pour en fait demander que $w^\mathsf{T} x_i + b$ soit en fait contraint à être égal (ou du moins aussi proche que possible) de ± 1. En remplaçant en effet la *contrainte d'inégalité* $y_i(w^\mathsf{T} x_i + b) \geq 0$ par un *coût quadratique* $(w^\mathsf{T} x_i + b - y_i)^2$, on change évidemment structurellement le problème de départ, mais par contre on simplifie largement les équations en écrivant un problème primal maintenant sous

la forme

$$(w^\star, b^\star) = \text{argmin}_{(w,b)} \frac{1}{2}\|w\|^2 + \gamma \frac{1}{2n}\sum_{i=1}^n e_i^2 \tag{1.3.1}$$
$$e_i = w^\mathsf{T} x_i + b - y_i$$

où les e_i dénotent des "erreurs" et $\gamma > 0$ est un paramètre de régularisation. Ce problème est maintenant une simple optimisation quadratique sous contrainte d'égalités linéaires, et a donc l'avantage d'avoir une solution explicite, donnée précisément (nous en reparlons plus tard dans le cours) par :

$$\begin{aligned} w^\star &= X D_y \alpha^\star \\ \alpha^\star &= \frac{1}{n} Q(1_n - b^\star y) \\ b^\star &= \frac{\frac{1}{n} y^\mathsf{T} Q 1_n}{\frac{1}{n} y^\mathsf{T} Q y} \\ Q &= \left(\frac{1}{n} D_y X^\mathsf{T} X D_y + \frac{1}{\gamma} I_n\right)^{-1}. \end{aligned}$$

On a donc maintenant parfaitement affaire à un problème de régression linéaire one-peut-plus standard vis-à-vis des vecteurs $y_i x_i$ qu'on régresse ici sur le vecteur 1_n (ou de manière similaire vis-à-vis des vecteurs x_i qu'on régresse ici sur le vecteur y) : le vecteur de régression est alors donné par

$$\begin{aligned} w^\star &= \frac{1}{n} X D_y \left(\frac{1}{n} D_y X^\mathsf{T} X D_y + \frac{1}{\gamma} I_n\right)^{-1} (1_n - b^\star y) \\ &= \frac{1}{n} X \left(\frac{1}{n} X^\mathsf{T} X + \frac{1}{\gamma} I_n\right)^{-1} (y - b^\star 1_n) \end{aligned}$$

où les deux égalités font apparaitre les deux formes de régression évoquées précédemment. Sous cette forme, l'algorithme de régression logistique résultant est connu sous le nom de *machine à vecteurs de support des moindres carrés*, ou *least squares support vector machine* (LS-SVM) en anglais.

Notons que, contrairement à l'habitude, apparait le terme $-b1_n$ dans cette régression. Elle est liée au fait que les vecteurs x_i n'ont pas été centrés : il ne changerait en effet rien d'imposer que $\frac{1}{n}\sum_i x_i = 0$ puisqu'on décalerait alors simplement l'ensemble des points par un vecteur constant. Dans ce cas $X1_n = \sum_i x_i = 0$ et on voit alors, en utilisant l'astuce $A(I+BA)^{-1} = (I+AB)^{-1}A$, que $\frac{1}{n}X(\frac{1}{n}X^\mathsf{T}X + \frac{1}{\gamma}I_n)^{-1}(y - b1_n) = \frac{1}{n}(\frac{1}{n}XX^\mathsf{T} + \frac{1}{\gamma}I_p)^{-1}X(y-b1_n) = \frac{1}{n}(\frac{1}{n}XX^\mathsf{T} + \frac{1}{\gamma}I_p)^{-1}Xy$, forme qu'on reconnait mieux en régression linéaire.

En anticipant quelque peu sur la suite, il est bon de remarquer que, pour des données de petites dimensions (en $2D$ ou $3D$ disons), cela semble extrêmement contre-productif de remplacer la contrainte de barrière $y_i(w^\mathsf{T}x_i + b) \geq 0$ par un coût quadratique $(w^\mathsf{T}x_i + b - y_i)^2$: un seul élément isolé loin de l'hyperplan séparateur pourrait déstabiliser totalement l'algorithme, qui est donc très peu robuste. Cependant, nous verrons par la suite que pour des données (ou plutôt des bonnes représentations de données) réelles, généralement aujourd'hui de grandes dimensions, la différence entre les deux méthodes s'atténue : en grandes dimensions (lorsque la taille p des x_i est grande), les vecteurs x_i ont tendance à se répartir uniformément au voisinage proche d'une grande hypersphère de $\mathbb{R}^p$ et on ne trouve alors plus beaucoup de vecteurs fondamentalement éloignés des hyperplans séparateurs. Cela homogénéise et même stabilise les algorithmes, et en plus rend de plus en plus efficaces les méthodes très simples telles que le LS-SVM.

Évoquons également rapidement un second point fondamental dont nous aurons à reparler : nous avons effectué l'hypothèse qu'il est possible de séparer l'ensemble des couples données-étiquettes (x_i, y_i) par un hyperplan dans l'espace ambiant. Il n'est pas bien compliqué de voir que c'est évidemment toujours possible lorsque $p > n$, à savoir quand la taille de l'espace est plus petite que le nombre de points présents, à moins qu'une partie de ces points soient alignés (ce qui en pratique, pour des données réelles, n'arrive pour ainsi dire jamais). Mais évidemment le cas intéressant est celui où $n > p$, voire même $n \gg p$. Dans ce cas, il est facile de générer des situations, même aléatoires, telles qu'avec haute probabilité, aucun hyperplan ne peut diviser l'espace des points en deux classes isolées. Prenons tout simplement dans $\mathbb{R}^2$ des vecteurs x_i tirés uniformément sur le cercle unité associés à des y_i tirés uniformément dans $\{\pm 1\}$. Il suffit de $n = 4$ vecteurs x_i pour avoir avec une probabilité non triviale l'absence d'hyperplan séparateur, et lorsque n augmente, très vite la probabilité d'absence d'un hyperplan séparateur tend vers 1. Dans ce cas le problème d'optimisation SVM n'a pas de solution : on aura alors recourt à une version relaxée du problème qui, au lieu d'imposer la *contrainte forte* $y_i(w^\mathsf{T} x_i + b) \geq 0$, ajoutera plutôt à la fonction de coût une *contrainte douce* du type

$$(w^\star, b^\star) = \operatorname{argmin}_{(w,b)} \frac{1}{2}\|w\|^2 + \gamma \sum_{i=1}^{n} \max\left\{0, 1 - y_i(w^\mathsf{T} x_i + b)\right\}$$

pour un certain $\gamma > 0$. Cette formulation consiste en fait à remplacer le coût "infini" de l'insatisfaction de la contrainte $y_i(w^\mathsf{T} x_i + b) \geq 1$ par un coût linéaire (d'autant plus grand que $y_i(w^\mathsf{T} x_i + b) - 1$ est fortement négatif) multiplié par la constante γ. On permet donc ici à certains x_i de ne pas valider la contrainte $y_i(w^\mathsf{T} x_i + b) \geq 1$ (et donc de se trouver du mauvais côté de l'hyperplan), à condition que la distance à l'hyperplan e_i associée soit minimale. On parle ici de *machines à vecteurs de support linéaire doux*.

Quant au LSSVM, qui est déjà une version "relaxée" du problème initial, lui reste valable même lorsqu'aucun hyperplan séparateur existe.

1.3.2.2. *Les noyaux et le SVM.* Les SVMs *linéaires* sont une idée simple, pertinente, numériquement aisée à résoudre (des outils puissants existent pour résoudre des problèmes de programmation quadratique), mais ils souffrent d'un postulat fort : celui que les données x_i, ou plus précisément leur représentation $\phi(x_i)$ est bien séparable (ou pas loin de l'être) dans leur espace ambiant. Ceci suppose en particulier qu'une telle "bonne fonction de représentation" ϕ existe.

C'est ici qu'intervient la notion de *noyaux* qui, très répandue en apprentissage et dont nous aurons l'occasion de parler, est extrêmement pertinente en SVM, car elle permet de faire jouer à plein ce qu'on appelle *l'astuce du noyau*, plus connue en anglais comme *kernel trick*.

Rappelons que nous avons montré un principe d'universalité des réseaux de neurones (Théorème 1) selon lequel il est possible de créer un système de représentants $\phi(x) = \sigma(Wx + b) \in \mathbb{R}^N$ qui, s'il est assez grand, crée une "diversité" suffisante de fonctions non-linéaires permettant de reproduire de manière arbitrairement bonne toute fonction mesurable de l'espace ambiant (sur des compacts). Nous avions mentionné que le théorème n'est cependant pas constructif et qu'on ne sait donc pas bien choisir les fonctions ϕ. Les réseaux de neurones le feront par apprentissage des poids et biais (W, b), mais les SVMs (et les méthodes à noyaux en général) optent pour une autre approche : l'idée est plutôt de prendre une fonction $\phi(x) \in \mathbb{R}^N$ de dimension très grande, voire même souvent infinie (mais dénombrable), mais qui a la propriété que les produits scalaires $\phi(x_i)^\mathsf{T}\phi(x_j)$ s'évaluent très facilement. En effet, opter pour un $\phi(x)$ de grande dimension permet de se rapprocher de l'idéal

du Théorème 1 et, même si les entrées de $\phi(x)$ ne sont pas optimisées, si elles sont nombreuses, on peut espérer qu'elles soient suffisamment riches et diversifiées pour "bien occuper" l'espace des fonctions $f : \mathbb{R}^p \to \mathbb{R}$; cependant, calculer ces nombreuses fonctions a un coût, d'autant plus inacceptable que N est grand (et encore plus si $N = \infty$). Le point-clé de l'astuce du noyau est de remarquer que, dans la formulation du test du *SVM non-linéaire* (ici en effet on travaille avec les représentants $\phi(x)$ au lieu de x)

$$g(x) = w^\star \phi(x) + b \underset{\mathcal{H}_{-1}}{\overset{\mathcal{H}_1}{\gtrless}} 0$$

on est amené à évaluer

$$g(x) = \phi(x)^\mathsf{T} \Phi D_y \alpha$$

où $\Phi = [\phi(x_1), \dots, \phi(x_n)]$ et où α se résout maintenant en maximisant la quantité $-\frac{1}{2}\alpha^\mathsf{T} D_y \Phi^\mathsf{T} \Phi D_y \alpha + \alpha^\mathsf{T} 1_n$ sous la contrainte $\alpha^\mathsf{T} D_y 1_n = 0$. Ainsi, tout le problème est fonction, non pas des $\phi(x_i)$ et $\phi(x)$ eux-mêmes, mais plutôt de la matrice $\Phi^\mathsf{T}\Phi$ et du vecteur $\phi(x)^\mathsf{T}\Phi$ dont les composantes sont les produits scalaires $\phi(x_i)^\mathsf{T}\phi(x_j)$ et $\phi(x)^\mathsf{T}\phi(x_i)$. En définissant l'opérateur

$$k(z, z') = \phi(z)^\mathsf{T} \phi(z')$$

qu'on appellera *noyau* (la notation $k(\cdot,\cdot)$ vient de *kernel* en anglais), évaluer $g(x)$ pour des représentations ϕ des données revient donc à *évaluer seulement* les $n(n-1)/2$ scalaires $k(x_i, x_j)$ et les n scalaires $k(x, x_i)$.

Évidemment, évaluer $k(z, z')$ lorsque $\phi(z)$ est de grande dimension coûte toujours très cher et le problème subsiste. L'astuce du noyau consiste à remarquer que, pour certains bons choix de ϕ, $k(z, z')$ peut s'évaluer très rapidement à coût réduit. L'exemple emblématique et dont nous parlerons beaucoup dans ce cours (du fait de son lien avec les réseaux de neurones aléatoires) est celui du noyau *gaussien* (aussi appelé noyau de la chaleur). Celui-ci est obtenu ainsi :

$$\phi(z) = [\phi_1(z), \dots, \phi_N(z)]^\mathsf{T} \in \mathbb{R}^N$$
$$\phi_i(z) = \frac{1}{\sqrt{N}} \exp(-\imath \omega_i^\mathsf{T} z)$$

où ω_i sont tirés aléatoirement et indépendamment selon la loi $\mathcal{N}(0, I_p)$ et où N est pris très grand. Dans ce cas, par la loi des grands nombres

(1.3.2)

$$\phi(z)^\mathsf{T}\phi(z') = \frac{1}{N}\sum_{i=1}^N \phi_i(z)^2 \underset{N\to\infty}{\longrightarrow} \mathbb{E}_{\omega_1 \sim \mathcal{N}(0,I_p)}[\phi_1(z)^2] = \exp\left(-\frac{1}{2}\|z - z'\|^2\right) = k(z, z')$$

(exercice : démontrer ce résultat !). Ainsi, au lieu de générer un grand nombre N de fonctions aléatoires $\frac{1}{\sqrt{N}}\exp(-\imath\omega_i^\mathsf{T} z)$,[2] et d'effectuer les produits scalaires correspondants, il suffit donc d'évaluer simplement $k(z, z') = \exp(-\frac{1}{2}\|z - z'\|^2)$ qui est bien moins coûteux si $p \ll N$.

Mais en fait, l'astuce du noyau est bien plus puissante que cela : il n'est en effet en général pas nécessaire d'exhiber des bons choix de fonctions $\phi(z)$ qui donnent lieu à un produit scalaire $\phi(z)^\mathsf{T}\phi(z')$ facile à évaluer. En pratique, on utilise un résultat fondamental, dû à Mercer, et qui dit la chose suivante :

2. Remarquez que ces fonctions sont complexes, cas que nous n'avons pas vraiment permis jusque là ; cependant il est facile de décomposer $\phi_i(z)$ sous la forme de deux fonctions $\phi_{i;R}(z) = \cos(\omega_i^\mathsf{T} z)$ et $\phi_{i;I}(z) = -\sin(\omega_i^\mathsf{T} z)$ que l'on peut recombiner de même en le noyau $k(\cdot,\cdot)$.

THÉORÈME 2 (Noyaux de Mercer). *Toute fonction $k(z, z') : \mathbb{R}^p \times \mathbb{R}^p \to R$ continue, symétrique (à savoir telle que $k(z, z') = k(z', z)$) et semi-définie positive, à savoir telle que pour tout n et pour toute famille $z_1, \dots, z_n$, la matrice*

$$K = \{k(z_i, z_j)\}_{1 \leq i,j \leq n}$$

est définie positive, alors il existe une fonction $\phi : \mathbb{R}^p \to H$ pour un certain espace pré-hilbertien H (un espace vectoriel muni d'un produit scalaire, possiblement de dimension infinie), telle que[3]

$$k(z, z') = \phi(z)^\mathsf{T} \phi(z').$$

D'après le théorème et suite à ce que nous avons détaillé auparavant, pour faire fonctionner un SVM, il n'est en fait pas du tout nécessaire d'exhiber la représentation $\phi(\cdot)$ qui sous-tend le noyau $k(\cdot, \cdot)$ qu'on utilisera alors exclusivement pour opérer le SVM. Ceci a un avantage calculatoire certain mais d'un autre côté nous fait perdre peu à peu la vision concrète sur ce que nous sommes en train de faire : en effet, le fait de travailler avec un noyau $k(\cdot, \cdot)$ arbitraire nous dit dès lors assez peu ce que le système de représentants sous-jacent parvient à extraire comme information pertinente sur les données. Pour se donner encore plus de flexibilité, on pourrait même choisir (c'est ce qui est fait parfois dans la littérature scientifique) des noyaux $k(\cdot, \cdot)$ qui ne sont pas semi-définis positifs et ne satisfont donc pas les conditions de Mercer du Théorème 2. Ceci reste en effet formellement possible étant donné que le SVM donnera de toute façon une valeur $g(x)$ à toute donnée x indépendamment du choix du noyau $k(\cdot, \cdot)$. Les intuitions de base sur le fonctionnement des SVMs s'éloignent dans ce cas de plus en plus.

Sans aller trop loin ici car ce n'est pas l'objet du cours, évoquons tout de même que l'astuce du noyau s'étend bien au delà de son application aux SVMs. En fait, on définit comme *méthodes à noyau* un ensemble assez vaste d'outils en apprentissage qui, d'une manière ou d'une autre, exploite l'astuce du noyau. Alors qu'ici nous sommes dans un cadre d'apprentissage *supervisé*, en ce sens que toutes les données d'apprentissage sont déjà étiquetées (à chaque x_i est associé un y_i connu), dans d'autres contextes d'apprentissage, semi-supervisé (seule une petite portion des étiquettes sont connues), non-supervisé (aucune étiquette n'est connue), par transfert (on connait des couples (x_i, y_i) mais associés à un autre problème d'apprentissage, proche de celui qui nous intéresse), etc., nombreuses sont les méthodes qui exploitent les représentations par noyaux.

Pendant le cours, nous serons amené à travailler en détail sur l'algorithme du SVM, notamment dans le cadre d'une séance de **travaux pratiques**.

1.3.2.3. *Les limitations fortes du SVM.* Les SVMs ont connu un fort succès dans les années 1990 à 2000 et ont même largement supplanté les réseaux de neurones dont il n'était presque plus question en apprentissage à cette époque. Ceci est dû notamment à l'instabilité des réseaux de neurones, elle-même liée à une difficulté théorique structurelle à les comprendre : rappelons que l'apprentissage d'un réseau de neurone consiste à résoudre (comme on le peut !) un problème non-convexe contenant un nombre important de minima locaux aux performances très variées, tandis qu'entrainer un SVM revient à résoudre un problème convexe dont la solution est unique et numériquement satisfaisante.

Cependant les SVMs peuvent être vite coûteux à mettre en place, notamment lorsque le nombre de données d'entrainement n est grand, car alors le problème d'optimisation, de dimension n, peut être trop important pour être numériquement intéressant. Par ailleurs, même si l'algorithme est simple et son implémentation

3. On abuse ici de la notation $(\cdot)^\mathsf{T}(\cdot)$ pour définir le produit scalaire, même en dimension infinie.

numérique est plutôt fiable, il repose néanmoins que le choix souvent arbitraire d'un noyau $k(\cdot,\cdot)$ associé à une représentation sous-jacente ϕ souvent totalement ignorée. En fait, on peut même dire que l'expérimentateur a en général si peu de contrôle sur l'effet du noyau qu'une grande majorité des ingénieurs et chercheurs exploitant des méthodes à noyaux se contentent bien souvent de prendre comme noyau la fonction

$$k(z,z') = \exp\left(-\frac{1}{2\sigma^2}\|z-z'\|^2\right)$$

pour un choix σ^2 du noyau (qu'on appelle souvent *la bande passante*) réglé à la main ou par *validation croisée* (à savoir, et nous y reviendrons, en testant différentes valeurs de σ^2 sur une sous-partie des données d'entrainement et en sélectionnant celle qui permet à $g(\cdot)$ de mieux classer l'autre sous-partie des données d'entrainement). Ce manque de contrôle sur le système de représentation est un problème fondamental en ce sens que la méthode SVM devient trop "statique" et essentiellement utilisée sur toutes les données que l'on souhaite classer de manière aveugle et non-différenciée ; par cela nous entendons notamment que l'on classera en général des images (chien ou chat) ou des textes (relatifs à la religion ou au sport) en utilisant le même noyau, sans exploiter du tout la structure des données.

Cette dernière difficulté d'une représentation adaptée, et non arbitraire, des données, est vraiment centrale et a fait le jeu récemment des réseaux de neurones qui, eux, s'attachent justement à apprendre finement la structure des données.

1.3.3. La révolution de l'apprentissage profond : le retour en grâce des réseaux de neurones. Alors qu'ils n'étaient plus en vogue, quelques chercheurs ont continué à pousser l'idée des réseaux de neurones en apprentissage, jusqu'à un jour crucial où, au moyen d'un seul article scientifique, le domaine de l'apprentissage (en tout cas au début le sous-domaine de la vision assistée par ordinateur) a littéralement basculé. Il faut pour cela se replacer dans le contexte des années 2000-2010, notamment en vision assistée par ordinateur (*computer vision* en anglais) : dans ce domaine, depuis des années, les capacités de reconnaissance d'images, de classification notamment, stagnaient à des niveaux plutôt modestes de qualités. Par modeste, on entend ici que les capacités d'identification des objets par un être humain restaient bien supérieures aux capacités des machines, de sorte que, commercialement, ces algorithmes étaient peu utilisés. Les grands mouvements scientifiques à cette époque consistaient en l'élaboration de filtres (ou noyaux) intelligents déduits aux images, sons, etc., *guidés par la connaissance humaine et statistique* : il s'agissait notamment d'effectuer non plus des filtres simples par transformées de Fourier comme c'est la pratique universelle en traitement des signaux et images, mais de concevoir des filtres plus généraux, dits en *ondelettes*, qui passent mieux à l'échelle et sont plus proches de la structure hiérarchique des données.

Les réseaux de neurones dits profonds (*deep neural networks* en anglais) ont totalement changé la donne, en basculant de nouveau vers un apprentissage des filtres à l'aide d'exemples (on dit en anglais *data-driven*) au lieu de filtres finement réglés par les connaissances d'ingénieurs de chaque domaine. En outre, dès le premier article à ce sujet, les performances de reconnaissance d'images du réseau de neurones profond a soudainement largement dépassé en performance les méthodes alternatives, à commencer par les SVMs aussi finement réglés que possible. Les quelques mois et années qui ont suivi ont vu ces chiffres de performance encore augmenter au fil de l'affinement de la méthode initiale, pour aujourd'hui atteindre des qualités de détection et de classification parfois surhumaines (à savoir que le réseau se trompe en moyenne moins qu'un humain).

Dans l'esprit, il s'agit toujours de travailler avec le modèle du perceptron multi-couches, mais avec essentiellement trois spécificités qui changent complètement le niveau de performances et de stabilité du réseau de neurones :

— **la structure des couches du réseau** : afin de prendre en compte les corrélations multi-échelles (dans le cas des images : entre pixels proches tout d'abord, puis entre zones plus éloignées de l'image), chaque couche du réseau de neurones opère comme un filtre linéaire "local" au lieu d'être un filtre opérant globalement sur la donnée (on évite ainsi de corréler des zones petites et éloignées de la donnée) ; ceci réduit sensiblement le nombre de paramètres de construction des couches $W^{(1)}, \ldots, W^{(L)}$, qui sont dites *couches convolutionnelles* (car, sous cette contrainte, elles effectuent un calcul de convolution locale par un motif dont les paramètres sont alors appris par le réseau). Une opération non-linéaire supplémentaire de sous-échantillonnage (dite en anglais de *pooling*) permet encore de réduire le nombre de paramètres en ne retenant que les motifs dont la convolution a donné les meilleurs résultats (il suffit d'employer une fonction "max" par exemple). Une visualisation sommaire du fonctionnement d'un réseau de neurones profonds est présentée en Figure 1.3.8.

— **la taille du jeu de données annotées** : au contraire des réseaux de neurones présentés précédemment, l'apprentissage des réseaux de neurones profonds s'effectue sur un ensemble de données annotées colossal, de l'ordre en général de millions de données ; c'est notamment le cas de la populaire base de données ImageNet représentée à gauche de la Figure 1.3.9 qui contient environ 16 millions d'images réparties en plusieurs centaines de classes d'objets. Il est important de souligner qu'initialement *toutes ces données ont été annotées* par des êtres humains, ce qui enlève beaucoup de la "magie" du réseau profond tant médiatisée.

— **la taille du réseau lui-même et la capacité d'apprentissage** : toujours au contraire des réseaux de neurones des années 1990, les réseaux profonds tirent leur nom du fait que le nombre de couches, ainsi que le nombre de neurones de chaque couche, est également colossal. On peut aller aisément aujourd'hui jusqu'à 100 couches (en général un peu moins cependant), contenant chacune plusieurs milliers de neurones. Pour apprendre les poids et biais d'un tel réseau, on procède par une simple méthode de descente de gradient sur l'ensemble du réseau. Cette méthode peut prendre beaucoup de temps, mais a l'avantage de se paralléliser très bien. Au moyen de serveurs multi-cœurs adaptés aux calculs matriciels (c'est pour cela qu'on privilégie des GPUs, spécialisés dans ces opérations, aux CPUs plus génériques), il n'est cependant pas rare d'atteindre des temps de convergence se comptant en jours ou semaines de calcul. A nouveau, mis en comparaison à la capacité d'un cerveau de mammifère, la quantité d'énergie requise pour l'apprentissage d'un tel réseau de neurones artificiels est hors de proportions. [4]

Comme nous sortons ici du cadre strict du cours, et que l'objectif de cette section est une introduction historiques des idées et notions de l'apprentissage automatique, nous ne développerons pas plus avant le détail de l'apprentissage par réseaux de neurones profonds. Cependant, il est bon de comprendre en quoi ces réseaux sont parvenus à révolutionner le domaine du traitement d'images en premier lieu, et maintenant également celui du traitement des sons, vidéos, de la parole et même plus récemment du langage naturel.

4. Ce dernier point a fait récemment naître la demande d'une "sobriété numérique" pour éviter les coûts prohibitifs de ces méthodes d'apprentissage qui, si elles sont pratiques et aisées à prendre en main, ont pour revers d'être extrêmement coûteuses et donc polluantes.

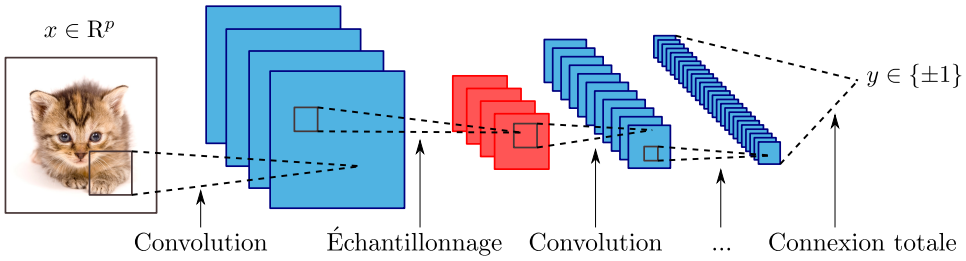

FIGURE 1.3.8. Visualisation du concept d'un réseau de neurones profond.

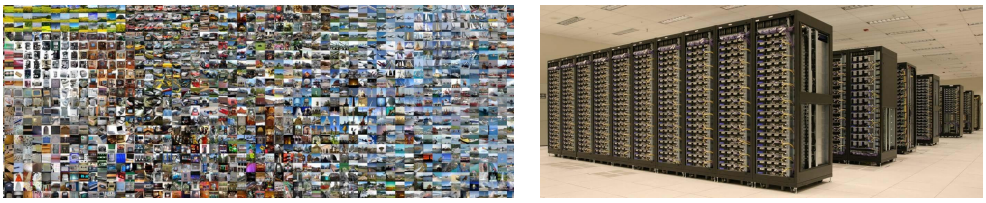

FIGURE 1.3.9. La clé du succès des réseaux profonds : la taille des bases de données *annotées* et la puissance de calcul.

Au contraire d'autres domaines de l'ingénierie (traitement du signal, théorie de l'information, mécanique, physique, etc.), en traitement des images (et encore plus en traitement naturel du langage), il est extrêmement difficile de mettre en place des *modèles* simples des données qui soient suffisamment universels et exploitables. C'est pourquoi les méthodes à noyaux et les filtres par ondelettes, aussi fins soient-ils, ne sont pas suffisamment génériques pour extraire des informations de classification d'une base arbitraire d'images ; du moins, pour une image x, la représentation $\phi(x)$ par ondelettes ou autre noyau élaboré n'est pas assez riche (ou peut-être au contraire trop détaillée ?) pour capturer précisément les éléments "sémantiques" qui font qu'un chien n'est pas un chat. Le réseau de neurones ne prête presque aucune attention à la structure de la source (sinon que c'est une image et qu'il est donc pertinent d'utiliser un filtrage convolutionnel hiérarchique – ce qui est déjà une information importante), et va simplement chercher à maximiser un objectif (ou minimiser un coût) sur la base d'un grand nombre d'exemples. Ainsi, si l'apprentissage s'avère fonctionner correctement, et atteindre notamment un "bon" minimum local du paysage énergétique, le réseau sera parvenu de lui-même et seulement sur la base d'exemples, à créer un filtre adapté à la différentiation d'images ressemblant à celles de la base d'apprentissage.

Cette remarque nous permet de mettre en avant trois points essentiels :

— **les DNNs génèrent "gratuitement" des représentants efficaces** : les DNNs ont un avantage indirect : si on observe plus précisément leur structure en Figure 1.3.8, on observe qu'ils sont constitués d'un ensemble de couches convolutionnelles de tailles plusieurs milliers de neurones, avant de finalement se refermer sur une dernière couche de décision "complètement connectée". Ce qu'il faut comprendre ici est que le réseau génère, à l'aide de ces multiples couches, une représentation finale des données $\phi(x)$ dans la dernière couche, qui sera alors simplement "régressée" sur la sortie ultime (ici ± 1 dans le cas d'une classification binaire). Dès lors, il est possible d'extraire $\phi(x)$ comme un représentant concret de l'image x. Ce représentant pourra même être utilisé pour d'autres problèmes comme point d'initialisation du réseau, lorsqu'on devra par exemple traiter d'autres types ou classes

d'images. Ainsi, les réseaux DNNs donnent accès à des représentants très riches de données, bien supérieurs en qualité à ceux construits manuellement (par des ondelettes par exemple) ou aléatoirement (par un noyau arbitraire).

— **la taille des DNNs les rend très stables** : nous avons évoqué précédemment que les réseaux de neurones sont très instables car possèdent un nombre exponentiellement croissant (avec la taille du réseau) de minima locaux ; de manière a priori assez surprenante, les DNNs, du fait de leur grande taille, convergent systématiquement (du moins avec très haute probabilité) vers des minima locaux de très bonne qualité. Cette observation est aujourd'hui cruciale, et nous allons en reparler très vite : lorsque les *données sont nombreuses et de grandes tailles*, un effet de "loi des grands nombres" intervient et tend à stabiliser les algorithmes d'apprentissage (pas seulement les DNNs). Dans le cadre des DNNs, la conséquence majeure, qui a été observée empiriquement mais qui théoriquement est très difficile à prouver, est que le paysage énergétique du réseau est très *plat* : tous les minima locaux semblent se stabiliser à une valeur plus ou moins constante de sorte qu'indépendamment du point d'initialisation, on trouve toujours une "bonne" solution.

— **mais les DNNs sont spécialisés et impossibles à comprendre** : ces nombreux atouts cachent cependant une réalité insaisissable aujourd'hui, celle de la compréhension théorique de la représentation ϕ générée par le réseau, ainsi que celles de sa stabilité, de son universalité vis-à-vis d'autres données, etc. Comment notamment s'assurer qu'une image légèrement modifiée de chien ne sera pas confondue avec un chat ? (il est en fait possible de volontairement altérer de manière minimaliste une image de chien, qu'un être humain ne pourra clairement pas confondre, de sorte à rendre le DNN absolument certain qu'il s'agit d'un chat) Ces problèmes structurels rendent l'utilisation des réseaux de neurones profonds dans certains domaines de sécurité critique (en médecine, dans l'armée) beaucoup trop hasardeux pour les y employer aveuglément. On connait notamment déjà les problèmes structurels liés à l'utilisation de la vision assistée par ordinateur dans les véhicules autonomes qui peuvent être volontairement trompés et parfois même commettent des erreurs de décision difficiles à comprendre.

Avant de conclure cette note historique, il est important de clarifier une fausse idée véhiculée par le succès des réseaux de neurones profonds, et qui pénètre malheureusement de trop nombreux domaines des sciences (au sens large) : à surmédiatiser la capacité des neurones profonds à atteindre des performances surhumaines (non seulement en vision, mais également en apprentissage par renforcement [5]), la fantaisie de la toute puissance et du *coût scientifique minimal de prise en main* des DNNs s'est emparée d'une large tranche de la population scientifique et de l'ingénierie. Cette fantaisie est elle que même dans des domaines (tels que la théorie de l'information) où on a élaboré dans la difficulté au cours des décennies des théories et algorithmes optimaux puissants, compétitifs, et adaptés aux problèmes *mais* construits à la main par des êtres humains, on en vient à vouloir aujourd'hui créer des réseaux de neurones capables de résoudre, sans vraiment savoir comment ni pourquoi, ces mêmes problèmes "encore mieux" (alors qu'on sait pertinemment être optimal) : l'avantage de se lancer dans ses "pseudo-recherches" est que cela ne coûte pas bien cher intellectuellement (comme il s'agit de "push-button") et qu'on ne peut avoir que de bonnes surprises (qu'on sera cependant bien en mal de comprendre). Si cet exemple est risible, il devient en fait plutôt inquiétant lorsqu'il s'agit de

5. Par exemple dans le cadre des échecs ou plus récemment du jeu de Go, où les machines entrainées avec des DNNs battent les champions humains.

croire, comme malheureusement beaucoup de nouveaux experts auto-proclamés [6] des DNNs le font aujourd'hui, que les réseaux de neurones peuvent résoudre des problèmes clairement inadaptés à l'architecture DNN et très mal dimensionnés : un exemple flagrant est celui d'imaginer inférer des diagnostics médicaux précis en créant un DNN nourri d'une quantité réduite (car peu disponible) de couples "imagerie/compte-rendu – diagnostic" ; il est inévitable que le DNN, s'il a accès à peu de données et que ces données elles-mêmes sont très hétérogènes, ne sera pas en mesure d'élaborer des diagnostics pertinents ; mais si ceux-ci ne sont pas vérifiables (comme dans le cas de maladies difficilement décelables ou chroniques), on court très vite le risque d'utiliser des outils dangereux dans le domaine public.

Cette fantaisie a aujourd'hui encore la dent dure et beaucoup trop d'ingénieurs et élèves-ingénieurs profitent de la simplicité (donc de l'effort intellectuel minimal) et aussi sûrement de l'émerveillement latent des DNNs pour s'y plonger corps et âme, abandonnant parfois complètement les socles mathématiques de leurs propres domaines d'application. C'est un écueil dangereux qui, à moyen terme, risque de faire stagner des pans entiers de domaines de recherche et d'ingénierie, notamment ceux qui ne se prêtent pas à l'utilisation de méthodes d'apprentissage profond.

1.3.4. L'avenir de l'IA : entre algorithmes puissants et nouveaux modèles mathématiques. On admet communément que les réseaux de neurones profonds ont révolutionné le domaine général de l'intelligence artificielle. Il y a clairement une part de vraie dans cette observation mais les premières promesses offertes par les DNNs en vision par ordinateur et les succès qui ont pu suivre en traitement du langage restent des cas plutôt isolés. Beaucoup imaginaient déjà une révolution cybernétique permettant de mettre en place des robots et ordinateurs si évolués qu'ils sauraient traiter des informations et prendre des décisions dans des contextes très variés. Nous n'en sommes cependant pas là et il faut bien comprendre les limites intrinsèques aux méthodes d'IA accessibles aujourd'hui.

La première limite est liée à la structuration convolutionnelle des DNNs qui clairement est un point essentiel de leur succès : entrainer, même aujourd'hui, un perceptron multi-couche générique, aussi grand soit-il, ne permet pas du tout d'atteindre les performances établies par les DNNs en vision notamment. Ceci signifie en particulier qu'il est nécessaire que les données traitées se prêtent à une modélisation par corrélation hiérarchique, à savoir en gros : (i) soit les données sont des images corrélées spatialement, (ii) soit les données sont des séries temporelles (sons, textes) corrélées spatialement. En dehors de ce cadre, les DNNs apportent peu de solutions viables.

La seconde limite est liée à la nécessité pour ces méthodes d'accéder à des bases de données *annotées* de tailles gigantesques. En pratique, ce n'est en général pas le cas car un humain ne peut prendre la peine d'étiqueter des millions d'images, sons, textes, etc. Pire, autant de données, même non annotées, sont rarement à la disposition des expérimentateurs. A défaut, comme nous l'avons évoqué plus haut, les réseaux pré-entrainés par les DNNs permettent toutefois d'obtenir des représentants ϕ qu'on peut espérer convenables si les jeux de données qu'on souhaite entrainer ressemblent suffisamment aux jeux de données plus larges et entièrement annotés qui ont donné naissance à la représentation ϕ ; on peut même reprendre une partie du réseau de neurones pré-entrainé et le modifier légèrement (par quelques étapes de descente de gradient) sur le nouveau jeu de données, afin d'affiner les performances. C'est une façon ici de faire de *l'apprentissage par transfert* (*transfer learning* en anglais).

6. Comme il est réellement facile de créer son propre DNN, tout le monde, sans aucune espèce de connaissance d'un domaine scientifique, peut s'y atteler en quelques semaines.

Cependant, si on regarde les choses positivement, il faut rappeler que le succès des DNNs vient surtout de la stabilisation du réseau lorsque la taille p et le nombre n des données sont très grands. Cette observation simple est clairement le fait d'une version élaborée mais intuitive de la loi des grands nombres : en effet, la dite "performance" d'un algorithme n'est finalement qu'un scalaire obtenu par le mélange, aussi complexe soit-il d'un grand nombre de degrés de libertés intrinsèque aux données, ce qui a tendance à faire converger ce "scalaire" indépendamment des données utilisées. Plus spécifiquement, les réseaux de neurones peuvent être vus comme l'enchainement de multiples fonctions non-linéaires *mais Lipschitz* (donc n'ayant pas une norme divergente) appliquées à de grands vecteurs de données $x \in \mathbb{R}^p$, entrainés sur une base de données de grande taille n, et utilisant un nombre de neurones NL (et donc des poids initialement choisis aléatoires) très grand. On commence à bien comprendre en fait aujourd'hui que l'ensemble des degrés de libertés intrinsèques aux données (on peut les voir comme des versions "bruités" d'une certaine sous-variété de $\mathbb{R}^p$) ainsi qu'au réseau lui-même (les poids initialement aléatoires), manipulés par des fonctions Lipschitz (donc ne faisant pas "exploser les données et le réseau"), a effectivement des comportements mathématiques dits de *concentration* (en lien ici avec la théorie de la concentration de la mesure).

Cette propriété mathématique de concentration de la mesure est d'autant plus fondamentale qu'elle s'accompagne d'une notion *d'universalité* qui, en somme, nous dit que les données $X \in \mathbb{R}^{p\times n}$ nombreuses et de grandes dimensions (pour lesquelles $p, n \gg 1$) se comportent, du point de vue des algorithmes qui opèrent sur elles des fonctions complexes de décision bien souvent Lipschitz $F : \mathbb{R}^{p\times n} \to \mathbb{R}$, $X \mapsto F(X)$, asymptotiquement *comme de grandes matrices gaussiennes*. Cela veut dire que, lorsque les dimensions sont grandes, la "loi" des données (si on pouvait bien les modéliser) importe peu, sinon leurs statistiques d'ordre un et deux (moyennes et covariances) qui, seules, permettent de bien saisir les performances de la plupart des algorithmes d'apprentissage. Ce constant (prouvé mathématiquement dans certains cas) explique d'abord pourquoi les algorithmes qui traitent de grandes données sont plus stables que ceux qui traitent de petites données, mais ouvre aussi la porte à une compréhension plus formelle et claire du comportement de ces algorithmes, comme il ne s'agit dès lors que d'analyser des fonctionnelles de variables aléatoires gaussiennes.

L'avenir de l'IA passe donc vraisemblablement par une redécouverte de la théorie de la concentration de la mesure, par la théorie des grandes matrices aléatoires, et promet d'enfin mieux comprendre, analyser et améliorer *mathématiquement* et non plus empiriquement les divers algorithmes d'apprentissage mooderne.

En quelques mots, cette brève introduction historique peut se résumer ainsi :

— l'IA est née du rêve de pionniers, notamment de l'informatique, de reproduire les capacités de décision, identification, classification, inférence, etc., du cerveau humain ; la découverte du principe d'approximation universelle des réseaux de neurones, seul élément mathématique réellement fondamental du domaine, a très vite suggéré la possibilité de réaliser un cerveau "bionique". Mais l'absence de résultats probants, l'instabilité des réseaux et le manque de compréhension et d'outils théoriques ont été trop d'obstacles pour les réseaux de neurones qui sont vite tombés en désuétudes ;

— l'apparition des notions mathématiques de systèmes de représentants et l'astuce du noyau, la création de filtres élaborés (en traitement du signal et de l'image), et l'idée simple mais convaincante des machines à vecteurs de support et d'un grand nombre d'algorithmes liés aux SVMs et finalement à la notion plus ou moins déguisée de régression, ont changé la vision

de l'IA des années 1990 qui est devenue une science dure intitulée "apprentissage automatisé". Ce sont adjoints à ces algorithmes des méthodes statistiques, probabilistes, graphiques, qui ont étoffé le domaine, sans cependant parvenir à l'unifier : nous entendons ici que pour attaquer chaque nouveau problème est devenue disponible une palette large mais pas facilement exploitable (quel algorithme choisir pour mon programme ?, quelles représentations, quel noyau ?, pourquoi ?, qu'espérer comme gains, qu'attendre comme risques ?) ;

— les réseaux de neurones profonds sont arrivés très récemment avec justement cette promesse, d'une part de performances largement accrues dans certains domaines (vision, parole, sons), mais surtout d'une simplification et une unification des concepts : nul n'est besoin ici de créer manuellement des représentations ou de choisir un noyau, car le réseau s'en charge seul ; mieux encore, le réseau n'a aucunement besoin de connaitre la nature des données qu'il va recevoir et traiter : tout est automatique ! Cette spécificité "*push button*" des réseaux de neurones profonds est à la fois magique mais chargée de problèmes sous-jacents : nous sommes incapables de comprendre ce qu'effectue réellement cette "boite noire", le dimensionnement exact du réseau (combien de couches, combien de neurones sur chaque couche ?) est plus un art qu'une science, les cas d'échec du réseau sont difficiles à anticiper, et surtout l'entrainement correct du réseau demande une quantité phénoménale de données pré-étiquetées et une consommation énergétique considérable ;

— dès lors, le prochain mouvement dans la recherche en IA semble bien être une remise à plat probabiliste et statistique des modèles, issus de la notion d'universalité et de concentration de la mesure des données de grandes dimensions, qui expliquent le comportement si stables des algorithmes traitant de grandes données ; ceux-ci deviennent paradoxalement plus simples à étudier (la loi des grands nombres rendant l'aléatoire déterministe !), ce qui permet d'envisager une nouvelle science des données sur des bases théoriques plus saines et unifiées.

Concluons tout de même en réévoquant le fil rouge de ce cours : les réseaux de neurones (du perceptron aux DNN) peuvent être vus comme des machines à générer des représentations des données, dont le traitement final (classification, inférence) se réduit finalement à une régression par une couche entièrement connectée, donc à une régression. Les SVMs, notamment LS-SVM, sont clairement également des machines à effectuer une régression sur un système de représentations exploitant l'astuce du noyau. Nous verrons dans le cours que d'autres méthodes, encore plus élémentaires, se ramènent aussi peu ou prou à une méthode de régression associée à une hypothèse statistique sur les données (c'est un peu le cas du LDA/QDA, c'est aussi le cas de certaines méthodes graphiques telles qu'utilisées en apprentissage semi-supervisé ou non supervisé) et on peut même voir les méthodes dites "spectrales" (qui extraient des vecteurs propres de matrices d'affinités ou de noyau) comme la limite d'une régression régularisée particulière.

En somme, sans vouloir toutefois être trop réducteur, il est intéressant de visualiser l'apprentissage automatisé comme une palette large et souvent difficile à trier de méthodes ayant comme dénominateur commun une combinaison somme toute très simple de représentations des données suivie d'une régression :

apprentissage automatisé = représentations de données + régression.

1.4. Apprentissage et données financières

Dans le contexte de la finance, les modèles statistiques et stochastiques générés au cours des années sont très fins et rendent assez bien compte des observations, notamment des séries temporelles d'évolution des cours. Les méthodes développées pour traiter ces données, de la prédiction à l'estimation de portefeuilles robustes, donnent des résultats très satisfaisants qui font aujourd'hui de la recherche en statistiques financières un domaine assez marginal (ou du moins localisé dans les entreprises privées et non pas dans les laboratoires publics). Il s'agit là typiquement d'un domaine où les méthodes d'inférence, d'estimation (par exemple, les outils basés sur le modèle de Markowitz) et de prédictions (par des processus auto-régressifs simples) suffisent en général à résoudre les problèmes du quotidien des analystes.

La place de l'apprentissage automatisé n'y est pas claire et les tentatives allant dans ce sens ne sont guère convaincantes. Néanmoins, deux directions spécifiques paraissent être intéressantes à explorer :

— **prédiction de séries temporelles financières réalistes** : une grande difficulté de traitement, notamment de prédiction, des séries temporelles financières réside dans leurs corrélations à temps long, dans les effets de "lag" et "lead", dans la présence d'événements de rupture, etc., qui rendent la modélisation et les méthodes d'inférence potentiellement compliquées ; l'utilisation de réseaux de neurones, mêmes très simples, et en particulier de réseaux *récurrents* de type *echo-state networks* adaptés aux séries temporelles, peut se révéler ici intéressante, et ce pour deux raisons : (i) ces réseaux, nous le verrons en temps voulu, génèrent (aléatoirement ici) un grand ensemble de représentations spatio-temporelles non-linéaires des données qui entrent séquentiellement dans le réseau, donnant ainsi accès à une vaste gamme de fonctions complexes qui sont finalement simplement régressées vers l'objectif (de prédiction) final, (ii) ces réseaux demeurent suffisamment compréhensibles, peu coûteux, peu paramétrés et assez accessibles mathématiquement pour être facilement manipulés et exploités.

— **regroupement statistique d'actifs** : pour améliorer la mise en place de portefeuilles efficaces, il est important d'utiliser des données statistiques les plus stationnaires possibles : ceci est à la fois valable dans le temps (il s'agit donc de considérer un historique assez long pour être statistiquement efficace mais pas trop) mais aussi dans l'espace des actifs conjointement pris en compte ; dans ce dernier cas, afin d'éviter d'avoir trop d'actifs p vis-à-vis du nombre limité n d'observations historiques, on souhaite réduire la taille du portefeuille, soit en le diversifiant sur des actifs mutuellement indépendants (minimisant ainsi le risque), soit au contraire en regroupant des actifs liés si on sait bien contrôler le risque (ce qui peut donner lieu à des revenus plus élevés). Pour cela, les méthodes d'apprentissage non-supervisées, semi-supervisées (lorsqu'on peut clairement aider la machine en pré-créant des "groupes" d'actifs appartenant à des secteurs différents) ou supervisées (si notre souhait est de plutôt de décider de la meilleure classe à attribuer à un nouvel actif) peuvent permettre de regrouper automatiquement les actifs en classes parallèles maximisant un critère bien choisi de distance.

Les travaux pratiques du cours exploreront ces idées.

1.5. Exercices

Les exercices de cette section ont un double objectif : certains ont pour but d'anticiper des résultats qui viendront plus tard dans le cours (permettant ainsi de prendre un peu d'avance), et d'autres permettent d'aller plus loin, sur des sujets

un peu plus “à la marge” mais très intéressants pour comprendre le cours un peu plus en profondeur.

Exercice 1 (Preuve du Théorème 1). Le but de l'exercice est de prouver le Théorème 1. Nous allons procéder par étape :

(1) On commence dans un premier temps par prouver le résultat, non pas sur $\mathcal{G}$, mais sur

$$\mathcal{G}_\Pi(\sigma) \equiv \left\{ g \mid x :\mapsto \sum_{i=1}^{n} a_i \prod_{j=1}^{m} \sigma\left(w_{ij}^\intercal x + b_{ij}\right),\ \forall\ n, m, \{a_i\}, \{w_{ij}\}, \{b_{ij}\} \right\}$$

qui est une classe de fonctions bien plus riches : de fait $\mathcal{G} \subset \mathcal{G}_\Pi$. Pour cela, on va commencer par prouver (ou admettre) cette version du théorème de Stone-Weierstrass :

Théorème 3 (Stone-Weierstrass). *Soit $\mathcal{A}$ une algèbre de fonctions réelles continues sur un compact $K \subset \mathbb{R}^p$ tel que, pour tout $x, y \in K$, il existe $g, \tilde{g} \in \mathcal{A}$ pour lesquels $g(x) \neq g(y)$ et $\tilde{g}(x) \neq 0$. Alors, pour toute fonction mesurable $f : \mathbb{R}^p \to \mathbb{R}$, il existe une séquence $g_1, g_2, \ldots \in \mathcal{A}$ telle que*

$$\lim_{n\to\infty} \sup_{x\in K} |g_n(x) - f(x)| = 0.$$

(*Indice* : Pour prouver le résultat, il suffit par densité de le prouver pour $f(x) = 1_{x\in V(x_0)}$ pour tout voisinage $V(x_0) \subset K$ de $x_0 \in K$. On doit donc créer une fonction proche de $1_{x\in V(x_0)}$ à partir de sommes et produits des fonctions continues de l'algèbre $\mathcal{A}$. Pour cela, on choisit une fonction g_1 de $\mathcal{A}$ distincte en $x_0, x_1 \in K$: on la translate et normalise pour être égale à 0 en x_0 et 1 en x_1. On fait de même pour une série de fonctions g_i en des points x_i qui “pavent” l'espace K. On moyenne les fonctions avec précaution et on élève la fonction résultante à une puissance qui assure de “coller” le voisinage de x_0 à zéro et le reste du support K à 1.)

Montrer alors que, si σ est continue mais non constante et μ est telle que $\mu(K) = 1$, alors $\mathcal{G}_\Pi(\sigma)$ satisfait les conditions du Théorème 3. Conclure en montrant qu'il existe toujours K tel que $\mu(K) > 1 - \varepsilon$ pour toute mesure de probabilité μ.

(2) Par le résultat précédent, montrer que $\mathcal{G}_\Pi(\cos)$ satisfait le théorème.

(3) En utilisant la propriété $\cos(a)\cos(b) = \frac{1}{2}(\cos(a+b) + \cos(a-b))$, montrer que $\mathcal{G}_\Pi(\cos) = \mathcal{G}(\cos)$ (à savoir $\mathcal{G}$ pour $\sigma(t) = \cos(t)$).

(4) Conclure en montrant que $\cos(t)$ peut être approximé sur des compacts par des éléments de $\mathcal{G}(\sigma)$ pour σ une fonction de “squashing” arbitraire. Pour cela, on pourra montrer (i) que le “cosine squasher”

$$\sigma(t) = \frac{1}{2}\left(1 + \cos\left(t + \frac{3\pi}{2}\right)\right) 1_{\{-\frac{\pi}{2} \leq \frac{\pi}{2}\}} + 1_{|t|>\frac{\pi}{2}}$$

est bien une fonction de squashing telle que, pour tout ensemble $[-M, M]$, la fonction $t \mapsto \cos(t)$ appartient à $\mathcal{G}(\sigma)$, et (ii) que pour toute paire (σ_1, σ_2) de fonctions de squashing, σ_1 peut être approximé sur des ensembles compacts par des éléments de $\mathcal{G}(\sigma_2)$.

Exercice 2 (Preuve du LSSVM). En utilisant la méthode des multiplicateurs de Lagrange, obtenir l'expression de $(w^\star, b^\star)$ solution de (1.3.1).

Exercice 3 (Noyau limite). Démontrer la formule (1.3.2) de la convergence asymptotique (presque sûre) des représentations de Fourier aléatoires vers le noyau gaussien.

Exercice 4 (Mon premier perceptron en Python). Cet exercice va nous permettre de réalise un réseau de neurones de type "perceptron" pour la classification supervisée des données populaires de la base MNIST (chiffres manuscrits, voir en utilisant le langage `Python` et le module `TensorFlow`, spécialisé dans la génération et l'apprentissage rapides de réseaux de neurones.

Nous commençons par charger les modules Python et par importer la base de données (si elle n'est pas déjà présente dans le dossier).

Listing 1.1. Chargement des modules Python

```
# Importation des modules, dont TensorFlow
from tensorflow.examples.tutorials.mnist import input_data
import tensorflow as tf
import numpy as np
from matplotlib import pyplot as plt

# Importations des données
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

Il s'agit maintenant de construire le réseau de neurones. Pour cela, nous utiliserons le code suivant, qu'il s'agit de bien comprendre.

Listing 1.2. Génération du réseau de neurones

```
# Création du modèle

def set_neural_network(hidden_layer_size,activation,lr):
    global x,y_,train_step,accuracy

    x = tf.placeholder(tf.float32, [None, 784])
    W = tf.Variable(tf.truncated_normal([784,hidden_layer_size],
        stddev=0.1))
    A = tf.Variable(tf.truncated_normal([hidden_layer_size, 10],
        stddev=0.1))

    b = tf.Variable(tf.constant(0.1,shape=[hidden_layer_size]))


    if activation is 'relu':
        y = tf.matmul(tf.nn.relu(tf.matmul(x, W)+b),A)
    elif activation is 'sign':
        y = tf.matmul(tf.sign(tf.matmul(x, W)+b),A)
    elif activation is 'sigmoid':
        y = tf.matmul(tf.sigmoid(tf.matmul(x, W)+b),A)

    # Définition des fonctions de perte et d'optimisation
    y_ = tf.placeholder(tf.float32, [None, 10])

    # Calcul de l'erreur empirique (ici cross-entropie) entre sorties
        obtenues et souhaitées
    cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(tf.nn.
        softmax(y)), reduction_indices=[1]))
```

```
25
26      # L'entrainement minimise la cross-entropie par descente de
            gradient pour une certain taux d'apprentissage lr
27      train_step = tf.train.GradientDescentOptimizer(learning_rate=lr).
            minimize(cross_entropy)
28
29      # Evaluations
30      correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
31      accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
32
33      return
```

L'exercice consiste à reproduire et comprendre le code précédent. Notamment, il s'agit de faire le lien avec le cours : quel est la réseau de neurone implémenté ici ? Quelle est la fonction de coût utilisée et en quoi consiste-elle ? Il est également important de comprendre le formalisme de "TensorFlow" : à quoi servent notamment les "placeholders" et pourquoi doit-on utiliser les opérateurs de produits matriciels "matmul" de TensorFlow et non pas de Numpy ?

LISTING 1.3. Code principal

```
1  hidden_layer_size = 20
2
3  for activation in ['relu','sigmoid','sign']:
4
5      # Taille de chaque ``batch'' d'images
6      batch_size = 50
7
8      # Nombre d'itérations (nombre de batchs évalués)
9      number_runs = 100
10
11     # Taux d'apprentissage évalués
12     learning_rates = [0.1,0.5,2.5]
13
14     # Matrice des scores
15     score = np.zeros([number_runs,len(learning_rates)])
16
17     for j in range(len(learning_rates)):
18
19         # Initialisation de l'arbre TensorFlow
20         set_neural_network(hidden_layer_size,activation,learning_rates
               [j])
21         sess = tf.InteractiveSession()
22         tf.global_variables_initializer().run()
23
24         # Phase d'apprentissage et scores
25         for i in range(number_runs):
26             batch_xs, batch_ys = mnist.train.next_batch(batch_size)
27             sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys
                   })
28
29             score[i,j]=sess.run(accuracy, feed_dict={x: mnist.test.
                   images,y_: mnist.test.labels})
30
```

```
31          plt.plot(score[:,j], color='{}'.format(j/len(learning_rates)),
                 label='{}'.format(learning_rates[j]))
32
33          # Performances de test
34          print('Learning rate: {:.3f}, Final score: {:.4f}'.format(
                learning_rates[j],sess.run(accuracy, feed_dict={x: mnist.
                test.images,y_: mnist.test.labels})) )
35
36      plt.title('{}'.format(activation))
37      plt.legend()
38      plt.show()
```

À nouveau, comprendre et tester les performances de l'algorithme. Comparer notamment les différents choix de modèles de fonctions d'activation. Généraliser le code précédent à un réseau multi-couches en utilisant les syntaxes Python/-TensorFlow et comparer les performances pour différentes profondeurs (nombre de couches) du réseau.

Chapitre 2

Éléments de base de l'apprentissage statistique

Nous entrons dans ce chapitre concrètement dans le cours, en commençant par formaliser les outils de l'apprentissage automatique à l'aide d'un système de notations et d'un formalisme mathématique clair. Nous discuterons brièvement des différents types et classes de problèmes et d'algorithmes d'apprentissage avant de nous recentrer spécifiquement sur l'apprentissage supervisé, qui est ici le but premier de ce cours (et par ailleurs le plus populaire).

2.1. Formalisation et notations

2.1.1. Données d'apprentissage et algorithmes. L'apprentissage automatique consiste à automatiser des opérations de traitement de données qui appartiennent à un *ensemble d'entrées* $\mathcal{X}$, auxquelles seront associées des décisions ou étiquettes appartenant à un *ensemble de sortie* $\mathcal{Y}$. L'ensemble $\mathcal{X}$ est généralement un espace de vecteurs (continu à support non compact $\mathbb{R}^p$, à support compact $[0,1]^p$, un dictionnaire discret $[a_1,\dots,a_k]^p$, etc., ou un produit de tous ces espaces), mais peut également formellement être un espace d'objets plus complexes (comme des espaces de fonctions $L^2(\mathbb{R})$ ou des séries $\mathbb{R}^{\mathbb{N}}$) même si numériquement on se ramène souvent à des vecteurs pour les implémentations pratiques. L'ensemble $\mathcal{Y}$ peut être quant à lui un ensemble *d'étiquettes*, aussi appelés labels ($\{\pm 1\}$ dans le cas binaire, $\{1,\dots,k\}$ dans le cas n-aire), un vecteur d'*étiquettes douces* $[0,1]^k$ (l'entrée i du vecteur donnant le score d'appartenance de la donnée à la classe i), un score ou valeur de régression ($\mathbb{R}$ dans le cas d'un score, $\mathbb{R}^d$ dans le cas d'un score multiple).

Un élément important qui différencie le domaine de l'apprentissage d'autres secteurs scientifiques est la présence de *données d'apprentissage*. Cette base de donnée est un ensemble, que nous supposerons de taille n, d'éléments de $\mathcal{X}\times\mathcal{Y}$

$$\{(x_1,y_1),\dots,(x_n,y_n)\}\,.$$

En apprentissage supervisé, l'ensemble des couples (x_i,y_i) est connu de l'expérimentateur. Pour d'autres types d'apprentissage, les x_i sont connus mais certains des y_i peuvent ne pas l'être. Il faut bien comprendre ici que généralement ces y_i ont été obtenus manuellement, et pas par le biais d'une autre machine, sinon le problème se mordrait la queue. C'est pourquoi souvent tous les y_i ne sont pas forcément accessibles.

Un algorithme d'apprentissage va alors avoir pour fonction, pour toute donnée-étiquette $(x,y)\in\mathcal{X}\times\mathcal{Y}$ jusqu'alors non observée d'effectuer une inférence sur y à partir de x ainsi qu'à partir de la base d'apprentissage $\{(x_1,y_1),\dots,(x_n,y_n)\}$. Il s'agira donc de générer une fonction de décision

$$g(x)=g(x;\{(x_1,y_1),\dots,(x_n,y_n)\}):\mathcal{X}\to\mathcal{Y}$$

et qui aura pour objectif de minimiser un certain critère d'erreur entre l'inconnue y et l'estimateur $g(x)$. Selon l'algorithme utilisé, toutes les fonctions $g:\mathcal{X}\to\mathcal{Y}$ ne sont pas accessibles, et elles seront dans ce cas choisies au sein d'une sous-famille. Nous avons déjà donné l'exemple des réseaux de neurones de type perceptrons qui,

même s'ils peuvent s'approcher arbitrairement près de n'importe quelle fonction f sur un compact (Théorème 1), n'en sont pas moins réduites à un ensemble concret de la forme $g(x) = a^\mathsf{T}\sigma(Wx+b)$. Nous allons découvrir très vite pourquoi il est fondamental que ces fonctions g puissent s'approcher de fonctions f arbitraires tout en étant cependant fortement contraintes (à être suffisamment "lisses" en l'occurrence).

Dans une vision purement statistique de l'apprentissage, on pourra supposer qu'il existe une fonction cachée f de *vérité terrain* (ground-truth en anglais) qui est telle que

$$f(x) = y$$

pour tout couple "valide" $(x, y) \in \mathcal{X} \times \mathcal{Y}$. À savoir, on peut imaginer que les étiquettes ou scores y sont définis de manière déterministe à partir des données x. Par exemple, dans le cas de la classification binaire où $\mathcal{Y} = \{\pm 1\}$, cela revient à dire que l'espace $\mathcal{X}$ se divise en deux sous-parties $\mathcal{X}_1$ et $\mathcal{X}_{-1}$ telles que $\mathcal{X}_1 \cap \mathcal{X}_{-1} = \emptyset$, $\mathcal{X}_1 \cup \mathcal{X}_{-1} = \mathcal{X}$ et $f(\mathcal{X}_i) = \{i\}$. Dans ce cas, il s'agira pour l'estimateur g d'approximer au mieux la fonction inconnue f, et ce au moyen seul des quelques échantillons $\{(x_1, y_1), \dots, (x_n, y_n)\}$ pour lesquels on sait que $f(x_i) = y_i$. En y réfléchissant bien, on comprend vite qu'en général ce problème peut être largement mal posé ! : si n est petit par rapport à la taille de l'espace des données $\mathcal{X}$, il existe beaucoup trop de fonctions g valident qui coïncident avec f sur l'ensemble d'apprentissage. On cherchera donc généralement parmi ces fonctions g celle qui satisfait d'autres critères : des a priori supplémentaires sur les données, un critère de "douceur" de la fonction, etc. Il est même en général peu pertinent de choisir une fonction g qui s'attache strictement aux données d'apprentissage (à savoir qui est telle que $g(x_i) = y_i$ pour tous les i), au risque d'engendrer une situation dite de *surapprentissage* dont nous parlerons très vite.

L'*approche statistique* a cependant un inconvénient majeur : si on revient sur le problème simple de classification binaire, en imposant l'existence de la fonction de vérité terrain f, on s'oblige à diviser l'espace $\mathcal{X}$ en $\mathcal{X}_1$ et $\mathcal{X}_{-1}$ de sorte que $\mathcal{X}_1 \cap \mathcal{X}_{-1} = \emptyset$ et $\mathcal{X}_1 \cup \mathcal{X}_{-1} = \mathcal{X}$, ce qui crée une frontière nette dans l'espace et rend délicate une modélisation probabiliste des données de $\mathcal{X}$ par des distributions à support non compact. On exclue notamment la possibilité de résoudre le problème élémentaire de classification d'un mélange de deux gaussiennes $x \sim \frac{1}{2}\mathcal{N}(+\mu, 1) + \frac{1}{2}\mathcal{N}(-\mu, 1)$ pour un certain $\mu > 0$ d'étiquettes $+1$ si x est tiré selon $\mathcal{N}(+\mu, 1)$ et -1 si x est tiré selon $\mathcal{N}(+\mu, 1)$: résoudre ce problème est légitime mais le support de $\mathcal{N}(+\mu, 1)$ et de $\mathcal{N}(-\mu, 1)$ étant $\mathbb{R}$ entier, leur intersection est non vide et il n'existe pas de fonction f de vérité terrain. Dans ce cadre, l'*approche est probabiliste*, et on ne cherchera plus à déterminer une fonction g qui s'approche au mieux d'une fonction $f : \mathcal{X} \to \mathcal{Y}$ de vérité terrain (celle-ci ne pouvant être définie) mais on cherchera plutôt g de sorte à minimiser une erreur *en espérance* sur le modèle probabiliste de l'espace des données-étiquettes $(\mathcal{X}, \mathcal{Y})$.

2.1.2. Mesures de performances. Deux, voire trois, types d'erreurs sont en général considérées en apprentissage, qu'il n'est pas toujours facile de bien contrebalancer :

— **l'erreur d'apprentissage** (*training error*), aussi appelé **risque empirique** (*empirical risk*) : comme le praticien n'a accès qu'aux données (x_i, y_i), $i = 1, \dots, n$, de la base d'entrainement, ainsi qu'à d'autres informations a priori (structure des données, loi a priori de tirage dans $\mathcal{X}$, etc.), il parait pertinent en premier lieu de choisir la fonction de décision g de telle sorte que $g(x_i) = y_i$ pour tout $i \in \{1, \dots, n\}$. Si cette égalité n'est pas vérifiée pour tous les x_i, on crée une erreur d'apprentissage définie en général dans

le cas où $\mathcal{Y}$ est *discret et non ordonné* par

$$E_{\text{train}}(g) = \frac{1}{n}\sum_{i=1}^{n} \ell(g(x_i), y_i)$$

pour une certaine distance ou perte (*loss* en anglais) $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}^+$. Par exemple, l'erreur quadratique empirique serait donnée par $E_{\text{train}} = \frac{1}{n}\sum_{i=1}^{n}(g(x_i) - y_i)^2$. Pour plus fortement pénaliser les erreurs importantes, on pourra prendre une norme L^q telle que $E_{\text{train}} = (\frac{1}{n}\sum_{i=1}^{n}|g(x_i) - y_i|^q)^{\frac{1}{q}}$, avec dans la limite $q \to \infty$, $E_{\text{train}} = \max_{1\leq i\leq n}|g(x_i) - y_i|$ qui donne la plus grande déviation aux données connues. Dans le cas où $\mathcal{Y}$ est discret et non-ordonné, on considèrera plutôt $\ell(g(x_i), y_i) = 1_{g(x_i)\neq y_i}$ qui compte les occurrences d'erreur.

— **l'erreur de généralisation** (*generalization error*) : bien évidemment, l'objectif d'un algorithme d'apprentissage est surtout d'approximer au mieux l'ensemble non observé des données qui seront amenées à être traitées. Si les données $(x, y) \in \mathcal{X} \times \mathcal{Y}$ non observées sont tirées suivant une mesure μ_{xy} (qui peut être discrète ou continue, mais en tout cas à support dans $\mathcal{X} \times \mathcal{Y}$), on définira alors pour la fonction g l'erreur

$$E_{\text{gen}}(g) = \mu_{xy}\left(\ell(g(x), y)\right).$$

Dans la vision statistique où $y = f(x)$ pour un f de vérité terrain inconnu, on pourra remplacer μ_{xy} par μ_x la mesure de génération de $x \in \mathcal{X}$, et y par $f(x)$ dans la formule précédente. Cependant, comme on ne connait pas par avance les données non encore observées, cette erreur E_{gen} ne peut être évaluée concrètement. À défaut, on pourra supposer un modèle statistique ou probabiliste pour les données et se contenter alors d'une estimation, en espérant que les vraies données suivent à peu près ce modèle. C'est aussi en cela que le modèle statistique où $y = f(x)$ est très faible car il ne permet de calculer E_{gen} qu'en posant pour a priori que f est connu, et on se mord la queue ; le modèle probabiliste est plus "sain" en ce sens car il est possible de construire simultanément les couples (x, y) avec y une fonction simple de la structure déterministe de x (par exemple le signe de la moyenne μ dans un modèle de mélange binaire gaussien $x \sim \mathcal{N}(\pm\mu, 1)$) et non pas une fonction compliquée $y = f(x)$ du tirage aléatoire de x. Un algorithme g atteignant des valeurs faibles de $E_{\text{gen}}(g)$ sera dit avoir de bonnes capacités de généralisation.

— **erreur de test** (*testing error*) : pour pallier au problème de l'accès à f dans la vision statistique ainsi qu'aux erreurs évidentes de modèles simplificateurs dans le cas probabiliste, une façon d'estimer $E_{\text{gen}}(g)$ est de "mimer" l'observation de données supplémentaires en divisant la base d'entrainement $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ en deux sous-parties disjointes : $\{(x_1, y_1), \ldots, (x_{n_{\text{train}}}, y_{n_{\text{train}}})\}$ de cardinal n_{train} et $\{(x_{n_{\text{train}}+1}, y_{n_{\text{train}}+1}), \ldots, (x_n, y_n)\}$, de cardinal $n_{\text{test}} = n - \text{train}$. La première partie des données servira de base à l'entrainement de l'algorithme g, et permettra notamment d'évaluer $E_{\text{train}}(g)$ mais restreinte à ces seules données, et la seconde partie permettra d'effectuer une *validation croisée* (en anglais, *cross-validation*) des capacités de généralisation de g en calculant

$$E_{\text{test}}(g) = \frac{1}{n_{\text{test}}}\sum_{i=n_{\text{train}}+1}^{n} \ell(g(x_i), y_i).$$

Par la loi des grands nombres, comme les (x_i, y_i) de cette sous-base de test sont tirés selon la loi μ_{xy}, on a bien :

$$E_{\text{test}}(g) \underset{n_{\text{test}} \to \infty}{\longrightarrow} E_{\text{gen}}(g)$$

presque sûrement (ou en probabilité). Le problème majeur avec cette approche, bien évidemment, est que pour obtenir une estimation assez fine de $E_{\text{gen}}(g)$, il est nécessaire de prendre n_{test} aussi grand que possible. Mais la qualité prédictive de g dépend exclusivement des données d'entrainement (au nombre n_{train}) qui, trop peu nombreuses, dégradent $E_{\text{gen}}(g)$. Le compromis est difficile et, surtout, est un crève-cœur car il s'agit réellement de se débarrasser d'une partie des données disponibles simplement dans un souci de mesure de performances.

Une approche qui suit ce principe à moindre coût est celle dite du *leave-one-out* : celle-ci consiste en une division de la base d'apprentissage $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ en $n_{\text{train}} = n - 1$ et $n_{\text{test}} = 1$, avec pour donnée test la donnée (x_i, y_i) prise séquentiellement pour $i = 1, \ldots, n$. L'idée est de construire n estimateurs $g_{-i}(x)$ identiques à ceci près qu'ils sont basés sur les $n - 1$ données $\{(x_1, y_1), \ldots, (x_n, y_n)\} \setminus \{(x_i, y_i)\}$, et d'évaluer leurs performances $E_{\text{gen}}(g_{-i})$ respectives sur l'unique donnée de test $\{(x_i, y_i)\}$. La moyenne de ces performances donne un estimateur de $E_{\text{gen}}(g)$ pour g une généralisation simple des g_{-i} qui prend en compte toutes les données. Cette approche est élégante en ce qu'elle évite le sacrifice d'aucune donnée d'entrainement mais a un double revers : (i) elle est en général extrêmement coûteuse car elle demande la construction de n fonctions g_{-i} parallèles (dans le cadre d'un réseau de neurones, il faudra construire n réseaux, et pour un SVM résoudre n optimisations convexes parallèles), mais pire que cela (ii) du fait de la dépendance forte entre les g_{-i} (qui ont paire-à-paire en commun $n - 2$ données), la loi des grands nombres ne s'applique plus et il est possible que $\frac{1}{n} \sum_i E_{\text{test}}(g_{-i})$ soit un mauvais estimateur de $E_{\text{gen}}(g)$ (du moins il s'agirait de démontrer au cas par cas que la loi des grands nombres reste valable).

2.1.3. Le surapprentissage. On pourrait se demander pourquoi il est nécessaire de différencier l'erreur d'entrainement et l'erreur de généralisation, et encore plus pourquoi il est nécessaire de sacrifier une partie des données d'entrainement pour valider le modèle. Après tout, même si en effet l'objectif est de mettre en place une fonction g qui fonctionne bien sur des données non révélées, si elle fonctionne déjà bien sur les données connues, pourquoi ne serait-ce pas le cas sur des données "similaires" non révélées ? L'erreur fondamentale dans cette affirmation est une question de dépendance : si g est une fonction de décision universelle sur $\mathcal{X}$, elle doit idéalement être une fonction de la loi μ_x (même si elle n'est pas connue) mais elle ne *peut pas* être fonction de tirages particuliers inconnus issus de $\mathcal{X} \times \mathcal{Y}$ (car en effet, dans ce cas, g serait dépendant de données non observables, ce qui n'a pas de sens). En construisant g à partir des (x_i, y_i) connus et en mesurant sa qualité sur cette base uniquement, on plonge négligemment dans cette erreur de dépendance. Pire, en s'efforçant à minimiser l'erreur d'entrainement E_{train} en choisissant une fonction g particulièrement compliquée (par exemple en prenant un perceptron de taille gigantesque pour coller au mieux aux données (x_i, y_i)), on prend le risque de se confronter au problème de *surapprentissage* extrêmement critique en apprentissage automatique.

Le surapprentissage est une difficulté importante de l'apprentissage qui n'est pas toujours bien comprise. Il faut comprendre que, si le problème d'apprentissage auquel nous avons affaire est *bien posé* ou bien *dimensionné*, on doit alors être

capables de trouver une solution g qui passe de manière "très lisse" par, ou pas très loin, des points (x_i, y_i). La contrainte de passer à proximité des (x_i, y_i) a évidemment pour but de satisfaire au moins les connaissances a priori, mais avec cette connaissance seule, beaucoup trop de solutions existent en général et le problème est de toute façon mal posé ; à moins que l'on suppose que ces points décrivent suffisamment bien les tendances de l'espace des paires (x, y) et donc il s'agit parmi ces solutions possibles de prendre celle qui passe de la manière la plus "souple" ou "lisse" au voisinage des points d'entrainement. Le fait que la solution doive être lisse est également cohérent avec la nature "réaliste" des données qui, si elles sont issues d'un système physique réel (comme c'est bien sûr souvent le cas), ne peuvent fondamentalement pas induire un comportement erratique, fortement non lisse des paires (x, y). En contraignant un passage rigoureux par tous les (x_i, y_i), on risque au contraire d'être amené à générer des fonctions g aux variations trop importantes et qui ne vont pas épouser de manière lisse l'espace des (x, y) non connus. On crée alors une solution très artificielle, on peut dire "contre-nature", qui essentiellement ne fonctionnera que sur la base d'entrainement (et $E_{\text{train}}(g) = 0$) mais pas du tout sur les points non observés (de sorte que $E_{\text{gen}}(g)$ peut être très grand)) : l'algorithme a donc *surappris*.

Ce problème est d'ailleurs exacerbé lorsque certaines paires (x_i, y_i) observées (ou même une seule en fait !) sont des données "aberrantes" ou "à la marge" (on parle en anglais d'*outliers*), en ce sens qu'elles ne sont pas typiques de données habituellement observées (par exemple l'image d'un chien vu de dos ou la tête cachée par un objet). Ces données, si on cherche trop à les utiliser (mais comment alors savoir s'il faut ou non les prendre en compte ?) risquent de renvoyer une mauvaise image de la fonction qu'on cherche à reproduire.

Remarque 4 (Surapprentissage : l'exemple des polynômes interpolateurs). Une façon visuelle et aisée de comprendre le problème du surapprentissage consiste à se poser le problème suivant : supposons l'existence d'une fonction inconnue $f : \mathbb{R} \to \mathbb{R}$ assez régulière, dont nous connaissons les points $(x_1, f(x_1)), \ldots, (x_n, f(x_n))$ et qu'il s'agit d'approximer par un polynôme g d'ordre arbitraire. Par le théorème de Lagrange, on sait qu'il est possible d'interpoler f de manière unique par un polynôme g d'ordre au plus $n - 1$. Cependant, f n'étant vraisemblablement pas un polynôme en général, si n est grand, on risque d'utiliser un polynôme à très grandes variations (d'ordre x^{n-1}) pour potentiellement interpoler une fonction très lisse. Autre cas de figure encore plus intéressant : si f est réellement un polynôme de petit degré mais que les observations y_i de $f(x_i)$ sont bruitées, par exemple $y_i = f(x_i) + z_i$ avec z_i un certain bruit de mesure, en utilisant l'unique polynôme interpolateur de degré $n - 1$, on ne va pas chercher à trouver f mais à trouver un polynôme qui passe parfaitement par les y_i, au risque que ce polynôme n'ait rien à voir avec f. C'est ce qui est illustré de manière frappante en Figure 4.

Pour éviter ces écueils, on cherchera en général à obtenir un bon compromis entre $E_{\text{train}}(g)$ et $E_{\text{test}}(g)$, sachant à nouveau que $E_{\text{gen}}(g)$ n'est pas une valeur accessible. C'est une des difficultés importantes de la théorie : quel poids mettre sur $E_{\text{train}}(g)$ et $E_{\text{test}}(g)$ dans l'optimisation ?, quel partitionnement des n données d'entrainement effectuer ? En général, en l'absence de fondement mathématique solide, seule la pratique et les règles de trois guident l'expérimentateur : on divisera par exemple conventionnellement l'ensemble des données accessibles en 90% de données d'entrainement et 10% de données de test.

Ce compromis est aussi connu dans la littérature statistique comme celui du *compromis biais-variance* (en anglais, *bias-variance tradeoff*). Un algorithme g est dit biaisé si la classe $\mathcal{G}$ d'où il est issu est trop éloignée de la fonction f que l'on

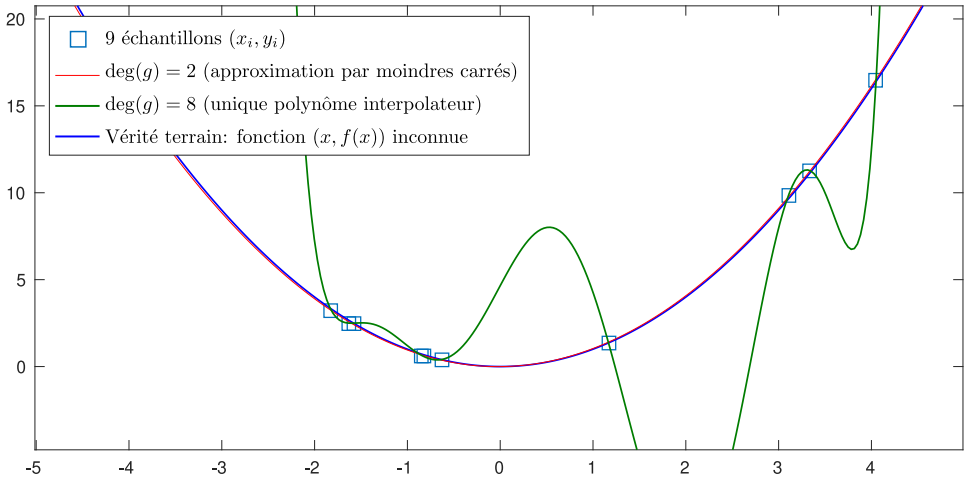

FIGURE 2.1.1. Interpolation polynomiale g optimale (par moindres carrés) de la fonction $f(x) = x^2$ observée à travers $n = 10$ échantillons bruités (x_i, y_i) pour $y_i = f(x_i) + z_i$ et $z_i \sim \mathcal{N}(0, .01)$. Le bruit est très faible (on distingue mal sur la figure la différence entre la vérité terrain $(x_i, f(x_i))$ et les points (x_i, y_i)) mais l'interpolation par g d'ordre 8 est catastrophique : on est en pleine situation de surapprentissage.

cherche : indépendamment du nombre de points observés (x_i, y_i) pendant l'apprentissage, on ne pourra jamais atteindre f de manière satisfaisante si f est trop éloigné de l'espace $\mathcal{G}$, ce qui crée un biais nécessaire (qu'on peut visualiser comme la distance de f à $\mathcal{G}$). Si par contre $\mathcal{G}$ est très large, on a plus de chance de s'approcher de f mais alors on risque aussi d'augmenter la sensibilité aux données (x_i, y_i) de l'algorithme et d'atteindre une fonction g ayant de trop grandes fluctuations : on augmente ici radicalement la variance.

REMARQUE 5 (Quand la multiplicité des algorithmes elle-même est source de surapprentissage !). Il faut bien comprendre que le surapprentissage est un écueil profond qui peut s'immiscer bien au delà des "données". Les compétitions d'IA aujourd'hui génèrent paradoxalement leur propre problème de surapprentissage : dans une compétition en IA, on nous donne accès à un ensemble de données d'entrainement que chaque compétiteur va utiliser pour développer son algorithme. L'organisateur de la compétition va également permettre aux joueurs, de temps en temps (par exemple chaque semaine), de tester son algorithme sur une base de test non observable et dont seul le score final sera reporté : celui-ci permet d'effectuer un classement en temps réel des compétiteurs qui peuvent ainsi voir leur marge de progrès possible ; en agissant ainsi (en ne révélant pas la base de test trop fréquemment), on évite que les joueurs s'entrainent à "surapprendre", volontairement ou non, la base de test, ce qui est une bonne chose. À la fin du jeu, une dernière base de donnée est révélée sur laquelle tous les algorithmes sont testés et le gagnant est celui atteignant le meilleur score sur cette base. Ce qu'on observe assez souvent malheureusement, c'est que les joueurs les mieux classés pendant la compétition obtiennent de très mauvais scores sur la base de données finalement révélée et que le gagnant est un compétiteur dont le classement n'était pas très bon. On se trouve typiquement ici dans un cadre où, du fait de la multiplicité des algorithmes, le classement en temps réel classe premiers les algorithmes qui "par hasard" sont bons sur

la base de test : ceux qui surapprennent même sans le vouloir, mais qui ne sont pas intrinsèquement bons. Cette remarque révèle les difficultés de dimensionnement et les biais inhérents à ces compétitions.

2.1.4. Éviter le surapprentissage : espace de fonctions et "hyperparamètres". Pour éviter le surapprentissage et balancer au mieux les erreurs $E_{\text{train}}(g)$ et $E_{\text{test}}(g)$, assez paradoxalement, il est nécessaire que l'algorithme d'apprentissage soit suffisamment *contraint*. Nous disons "paradoxalement" ici car il s'agit bien d'approximer une fonction f inconnue et potentiellement très complexe au moyen d'une famille $\mathcal{G}$ de fonctions g qu'on souhaite justement la plus réduite possible pour éviter la situation dramatique de la Figure 4. Un compromis entre richesse de $\mathcal{G}$ (pour au mieux approcher f) et limitation des erreurs de surapprentissage (liée à des g trop exotiques) doit être bien compris.

Plusieurs moyens sont utilisés par les algorithmes d'apprentissage pour parvenir à un bon compromis, qu'on peut généralement encapsuler dans la notion d'*hyperparamètres*. Précisément, il s'agit, non plus de chercher g dans un ensemble $\mathcal{G}$ fixe et indépendant du problème (comme par exemple dans le Théorème 1 où $\mathcal{G}$ est l'ensemble des perceptrons arbitrairement grands à une couche), mais plutôt de prendre g dans un ensemble $\mathcal{G}_\theta \subset \mathcal{G}$ où θ est un *hyperparamètre* qui réduit ou augmente la richesse des fonctions choisies. Dans le cadre du Théorème 1, θ pourra par exemple déterminer le nombre maximum de neurones du perceptron : si le nombre de données n est petit, pour éviter le surapprentissage on cherchera par exemple à prendre g dans une famille de perceptrons à nombre fixe $N = O(n)$ de neurones ; on pourra par exemple prendre $\theta = N$ et imposer que $\mathcal{G}_N = \{x \mapsto a^\mathsf{T}\sigma(Wx + b),\ W \in \mathbb{R}^{N\times p},\ a, b \in \mathbb{R}^N\}$. C'est d'ailleurs ce qui est fait aujourd'hui, sans qu'on ne s'en rende plus compte, dans les réseaux de type DNNs : pour éviter des familles trop riches de fonctions générées, on impose (i) une structure convolutionnelle des couches et des méthodes d'échantillonnage (*pooling*) et (ii) le nombre de couches et de neurones est dimensionné à quelques dizaines de couches au maximum (rarement plus) et quelques milliers de neurones par couche (rarement plus), ce qui crée un compromis plutôt correct et stable.

Dans d'autres algorithmes, les hyperparamètres peuvent être plus simples : il s'agit couramment d'un paramètre dit de *régularisation*. Par exemple, la méthode des moindres carrés ou, de manière très proche les machines à vecteurs de support, qui fonctionnent toutes deux comme des régresseurs, vont chercher à estimer un vecteur de régression (le vecteur normal à l'hyperplan séparateur) qui colle au mieux aux données d'apprentissage. Comme on le verra, pour éviter d'avoir des solutions qui colleraient bien trop aux données d'apprentissage et risquent leur surapprentissage, on introduira un hyperparamètre (souvent dénoté γ) pour éviter le surapprentissage. Dans l'Equation 1.3.1 du LS-SVM, ce paramètre donne lieu à une régression des moindres carrés *régularisée* via le filtre $\frac{1}{n}XD_y(\frac{1}{n}D_yX^\mathsf{T}XD_y + \gamma^{-1}I_n)^{-1}(1_n - by)$, en lieu et place du filtre des moindres carrés $\frac{1}{n}XD_y(\frac{1}{n}D_yX^\mathsf{T}XD_y)^{-1}(1_n - by)$ qui donnerait lieu (si $p > n$) à une erreur d'entrainement $E_{\text{train}} = 0$. Ici, la famille $\mathcal{G}$ des fonctions de test peut donc être sous-paramétrée par $\mathcal{G}_\gamma = \{g(x) = \frac{1}{n}x^\mathsf{T}XD_y(\frac{1}{n}D_yX^\mathsf{T}XD_y + \gamma^{-1}I_n)^{-1}(1_n - by)\}$.

L'utilisation de noyaux bien dimensionnés est aussi une manière de contraindre l'espace $\mathcal{G}$. En contrôlant $\phi(x)$, à la fois dans sa dimension et dans le choix des fonctions $\phi_i(x)$, on contrôle la richesse des représentations. En utilisant l'astuce du noyau, on pourra de manière équivalente contrôler l'ensemble des noyaux $k(\cdot,\cdot)$. Ce qui est notamment fait en pratique est d'utiliser un noyau gaussien $k(x, y) = \exp(-\frac{1}{2\sigma^2}\|x - y\|^2)$ où σ est un hyperparamètre de contrôle. On sent bien ici qu'en prenant $\sigma \ll 1$ petit, de nombreux termes de la matrice à noyau $K = \{k(x_i, x_j)\}_{i,j=1}^n$ vont s'annuler, de sorte à ne conserver que les distances entre x_i

"voisins", augmentant ainsi le risque de surapprentissage induit par des x_i isolés (ce qui sera particulièrement le cas si certains x_i sont des *outliers*). Si au contraire on prend $\sigma \gg 1$ grand, alors tous les $k(x_i, x_j)$ risquent d'être proches, $K \simeq 1_n 1_n^\mathsf{T}$ sera une matrice essentiellement composée de 1's, et l'approximation résultante aura très peu de degrés de libertés : on ne collera cette fois pas bien aux données d'entrainement.

D'un point de vue "stabilité" de l'apprentissage et ordres de grandeur, il est également important de noter qu'il est souhaitable que tout jeu de n données d'apprentissage arbitraire donne lieu, à peu de choses près, à la même approximation g, car sinon on obtiendrait autant de solutions variées que de jeux de données d'apprentissage, ce qui est à proscrire absolument. En se rappelant que les performances (de généralisation) de l'algorithme d'apprentissage convergent, par la loi des grands nombres, pour n grand, vers l'espérance $\frac{1}{n_{\text{test}}} \sum_{x_i \text{ test}} \ell(g(x_i), y_i) \to \mu_{xy}(\ell(x, y))$, afin de savoir à quelle vitesse cette convergence a lieu, on peut exploiter le théorème de la limite centrale qui dit que

$$\frac{1}{n_{\text{test}}} \sum_{x_i \text{ test}} \ell(g(x_i), y_i) = \mu_{xy}(g(x, y)) + \mathcal{N}\left(0, \frac{1}{n_{\text{test}}} \text{Var}[\ell(g(x_1), y_1)]\right) + o_p(n_{\text{test}}^{-\frac{1}{2}}).$$

Le terme $\text{Var}[\ell(g(x_1), y_1)]$ doit être aussi petit que possible pour être "écrasé" par n_{test}. Comme cette variance dépend de la variance de $g(x_1)$ et que celle-ci est grande si g est à fortes variations, on cherchera évidemment à éviter de trop grandes variations de la fonction g.

Remarque 6 (La puissance des opérateurs "Lipschitz"). Une façon de comprendre la puissance des réseaux de neurones est de voir qu'ils consistent en des couches successives d'opérateurs linéaires (les matrices de transitions $W^{(\ell)}$) et d'opérateurs non-linéaires *Lipschitz* σ. Par Lipschitz, on entend que $\|\sigma(z) - \sigma(z')\| \leq \alpha\|z - z'\|$ pour un certain α indépendant de z, z'. Comme la combinaison fonctionnelle d'opérateurs Lipschitz reste Lipschitz, un réseau de neurones n'est rien d'autre qu'une machine à construire des fonctions Lipschitz arbitrairement complexes. Cela est extrêmement intéressant parce qu'en contraignant ce caractère Lipschitz, on évite à l'algorithme de créer des fonctions à grandes variations : en effet, si le paramètre de Lipschitz α est bien contrôlé, on ne pourra jamais créer de fonctions dont la dérivée (ou en multivarié la norme du gradient) dépasse α. Cette contrainte forte crée donc des fonctions très lisses mais quand même très riches, ce qui est vraisemblablement l'un des arguments du si bon fonctionnement des DNNs. Si on autorisait σ à ne pas être Lipschitz, par exemple $\sigma(t) = t^2$, on s'expose alors au risque d'une "explosion" de la complexité des fonctions générées par le réseau dans l'enchainement des couches, et en l'occurrence d'inclure des polynômes d'ordre $2L$ sur un réseau à L couches : on retombera probablement dans le même problème qu'évoqué en Figure 4 sur l'interpolation polynomiale.

Plusieurs autres stratégies sont aussi utilisées dans les DNNs, toujours dans le même but : (i) le *dropout* qui consiste à volontairement couper des connexions "faibles" dans les $W^{(\ell)}$ en les mettant à zéro ; cela permet de connecter moins de neurones entre eux, et ainsi d'éviter de créer des fonctions trop complexes, ou (ii) initialiser le réseau avec des poids bornés plutôt que gaussiens : en agissant ainsi, on évite les poids trop "singuliers" qui relieraient trop fortement certains neurones pour donner lieu à des fonctions trop complexes ; le même procédé peut être opéré pendant l'apprentissage en bornant la valeur des poids, et ainsi éviter des dérives du réseau. Tout cela est cependant en pratique effectué "au flair", sans support théorique stable.

Au contraire des réseaux de neurones, les représentations $\phi(\cdot)$ et noyaux $k(\cdot, \cdot)$ trop "statiques", s'ils sont en effet souvent Lipschitz, ne jouissent pas d'assez de

paramètres pour leur permettre d'explorer une assez grande gamme de fonctions. On aura donc tendance à tenter volontairement de les rendre plus "complexes" pour qu'elles épousent mieux la fonction f que l'on cherche, au risque malheureusement de créer des instabilités fortes.

2.2. Les différents types d'apprentissage

Le cours s'attache à discuter de la classe la plus populaires des algorithmes d'apprentissage qui sont ceux dits "supervisés". Il est néanmoins pertinent de connaitre l'ensemble des autres méthodes d'apprentissage qui existent et qui sont aujourd'hui les plus populaires :

- **apprentissage supervisé** : il s'agit du problème le plus standard en apprentissage automatisé où on a accès à une base de donnée $(x_1, y_1), \dots, (x_n, y_n)$ de n données étiquetées à partir desquelles on cherche à reconstruire une fonction $f(x) = y$ au moyen d'une famille de fonctions $\mathcal{G}$ (dans l'interprétation statistique) ou à minimiser pour $g \in \mathcal{G}$ une distance $\mu_{xy}(\ell(g(x), y))$ (dans l'interprétation probabiliste). L'apprentissage supervisé suppose cependant l'existence d'une telle (souvent large) base de données étiquetées, ce qui est en réalité assez rare.
- **apprentissage semi-supervisé** : vraisemblablement la plus "plausible" et rencontrée des situations, l'apprentissage semi-supervisé suppose également l'existence d'une base de données $(x_1, y_1), \dots, (x_n, y_n)$ mais pour laquelle de nombreux y_i sont inconnus. On suppose ici, et c'est bien naturel, qu'un opérateur humain a pu prendre la peine d'étiqueter une partie des données, mais si n est très grand, n'a pas pu tout étiqueter. Il est aussi naturel de se dire qu'un bon algorithme d'apprentissage devrait s'en tirer seul, et l'étiquetage préalable d'une sous-partie des données est une aide extérieure dont peut bénéficier l'algorithme. En général, cette petite aide est très précieuse car : (i) elle permet de régler le nombre de classes de données souhaitées (s'il s'agit d'un problème de classification) et (ii) l'expérimentateur choisira en général spécifiquement les données les plus représentatives à étiqueter, évitant ainsi de trop grandes variances de l'algorithme induites par de mauvaises données. D'autres problèmes se posent par contre : les données étiquetées peuvent l'avoir été pour des raisons extérieures qui peuvent influencer l'algorithme (par exemple, dans un cadre médical, on peut n'avoir étiqueté que des images de patients malades, et aucune de patients sain, ce qui biaise énormément les proportions réelles des classes).
- **apprentissage non supervisé** : il s'agit ici du cas extrême inverse où la base de données $x_1, \dots, x_n$ ne contient aucune étiquette ; c'est alors à l'algorithme de découvrir lui-même le bon *regroupement* (on parle plutôt de regroupement que de classification dans ce cas) des données ou, pour un problème de régression, le bon espace de représentation des données qui les rendent "mutuellement cohérentes" (on procèdera par exemple à une analyse en composantes principales pour découvrir ce que les données ont en commun). De nombreuses difficultés existent évidemment ici : dans le cadre de la classification, on ne sait pas combien de classes sont censées être identifiées, et si un regroupement est découvert, rien ne dit qu'il s'agisse de celui que l'expérimentateur souhaite observer. Pour ce qui est de la régression, découvrir une sous-variété où vivent les données accessibles n'implique pas que l'information que l'on souhaite extraire de ces données se trouvent dans les directions dominantes de cette variété et il est alors difficile d'interpréter

ce que l'on observe (les données peuvent avoir un paramètre fort en commun qui n'est en fait qu'un paramètre de nuisance pour le problème qui nous intéresse).

— **apprentissage par transfert** : un peu moins populaire, mais aujourd'hui gagnant en intérêt, l'apprentissage par transfert consiste en l'apprentissage d'une tâche *objectif* à l'aide d'une autre tâche *source*. L'idée est de supposer que la tâche source est facile à réaliser (on a par exemple accès à énormément de données de chiens et de chats pour créer un classifieur) alors que la tâche objectif est plus dure (on veut classer des tigres et des loups, mais à partir de peu d'images), mais les deux tâches sont fortement liées (les chiens ressemblent à des loups, les tigres à des chats). En bénéficiant des données de la source, on cherche à optimiser les performances de la tâche objectif. Les données peuvent d'ailleurs être les mêmes dans la tâche source et la tâche objectif, et ce qui diffère peut être la tâche elle-même (isoler chiens et chats dans la tâche source, et isoler les animaux par couleurs dans la tâche objectif). L'apprentissage par transfert est d'ailleurs lié à l'apprentissage multi-tâches qui consiste, non pas seulement à optimiser une tâche objectif mais plusieurs tâches simultanément.

— **apprentissage par renforcement** : il s'agit ici d'un contexte assez différent où aucune donnée d'entrée n'est disponible à la machine qui doit néanmoins effectuer au mieux une tâche, et qui n'a comme stimulus que le score de réalisation de la tâche. D'une certaine manière, c'est donc la machine qui créera son propre algorithme g testé sur des données simulées x_i, de telle sorte à obtenir une sortie y_i "satisfaisante". Ceci entre généralement dans un contexte de jeux : jeu d'échec, jeu de Go, etc. La machine émet un choix g dont on va évaluer les sorties associées $g(x_i)$ pour un certain nombre de données x_i (par exemple tous les coups que va pouvoir jouer l'adversaire), et ce de manière itérative, jusqu'à ce que l'on obtienne un algorithme g tel que tous les $g(x_i)$ aient un bon score. Ces méthodes d'apprentissage par renforcement sont très élégantes et peuvent être parallélisées de manière compétitive entre plusieurs algorithmes qui sont en compétition mutuelle. C'est le cas des récents *réseaux antagonistes génératifs* (en anglais *generative adversarial networks* ou simplement *GANs*) qui consistent en deux réseaux de neurones en compétition : l'un ayant pour objectif de générer des images $g(x_i)$ réalistes à partir de bruits gaussiens $x_i \sim \mathcal{N}(0, I_p)$, et l'autre tentant de détecter de vraies images (à partir d'une base externe) des images fausses produites par le premier réseau.

Ces différents types d'apprentissage donnent évidemment lieu à des problèmes concrets différents, et donc à des algorithmes distincts. Néanmoins, on peut isoler assez facilement plusieurs grandes classes d'algorithmes d'apprentissage qui se déclinent parfois les uns des autres, selon qu'on ait affaire à un problème supervisé ou non.

2.3. Exercices

Exercice 5 (Surapprentissage). Il s'agit dans cet exercice de reproduire la situation de la Figure 4 et d'aller un peu plus loin. Commencer par démontrer que, pour

$$(x_1, y_1), \ldots, (x_n, y_n) \in \mathbb{R} \times \mathbb{R}$$

le polynôme

$$P_n(x) = \sum_{i=1}^{n} y_i p_n^i(x)$$
$$p_n^i(x) = \prod_{\substack{1\leq j\leq N\\ j\neq i}} \frac{x - x_j}{x_i - x_j}$$

vérifie $P_N(x_i) = y_i$ pour chaque $1 \leq i \leq N$: c'est un polynôme interpolateur. Prouver que ce polynôme est unique.

Simuler alors la situation très simple suivante : les couples (x, y) sont liés par l'équation $y = f(x)$ avec f un polynôme de petit ordre (on a prit un ordre 2 dans la Figure 4 mais on peut évidemment aller au delà !). Générer alors un ensemble de (x_i, y_i), $1 \leq i \leq n$, en tirant les x_i d'un ensemble compact bien choisi de $\mathbb{R}$ et tels que $y_i = f(x_i)$, et ce pour une valeur de n qu'on pourra adapter. Coder le polynôme interpolateur P_n de f décrit précédemment et tracer à la fois P_n et f. Que constate-on lorsque n est plus grand que l'ordre du polynôme f ? Confirmer par le résultat d'unicité de P_n.

On se place maintenant dans le cas précis de la Figure 4. Prenons maintenant $y_i = f(x_i) + z_i$ avec $z_i \sim \mathcal{N}(0, \sigma^2)$ pour une variance du bruit σ^2 qu'on pourra régler. Simuler maintenant f et P_n pour différentes valeurs de n. Que constate-on désormais ? Établir le lien avec la notion de surapprentissage.

On va tenter maintenant de corriger le problème en permettant à un nouveau polynôme "interpolateur" $Q_n^m(x) = \sum_{i=0}^{m} a_i x^i$ de ne pas passer rigoureusement par chaque couple (x_i, y_i) en lui imposant une contrainte bien choisie sur ses coefficients $a_0, \ldots, a_m$. Pour cela, considérer le problème d'optimisation

$$\min_{a_0,\ldots,a_m} \sum_{i=1}^{n} (Q_n(x_i) - y_i)^2 + \gamma \sum_{j=0}^{m} b_j a_j^2$$

pour des coefficients $\gamma, b_0, \ldots, b_m > 0$ à choisir. Résoudre ce problème et donner l'expression de $Q_n^m(x)$. En jouant sur les paramètres m ainsi que γ et $b_0, \ldots, b_m$, simuler et comparer le polynôme $Q_n^m(x)$ au polynôme $P_n(x)$ ainsi qu'à $f(x)$. Quels sont les choix pertinents des paramètres et pourquoi ? Faire le lien à nouveau avec le surapprentissage et les notions de régularisations vues jusqu'alors dans le cours.

Exercice 6 (L'intérêt des opérateurs Lipshitz). Reprendre l'Exercice 4 en changeant les fonctions d'activation σ par des opérateurs plus libres : (i) des fonctions non-Lipschitz, (ii) des fonctions discontinues, (iii) des fonctions invariantes par loi d'échelle (à savoir telles que $\sigma(\alpha x) = \alpha\sigma(x)$). Qu'observe-t-on ? Analyser ces résultats en terme du compromis "non-linéarité–stabilité". Au vu des résultats, expliquer en quoi la populaire fonction, dite ReLU, à savoir $\sigma(t) = \max(t, 0)$ est tant plébiscitée ? Dans un réseau de neurones biologiques, on a plutôt affaire à des activations de type $\sigma(t) = 1_{|t|>t_0}$ pour un certain seuil t_0 : comment comprendre la stabilité et la performance de ces réseaux ?

Chapitre 3

Algorithmes supervisés élémentaires

Avant de nous plonger pleinement dans le cœur du cours et des méthodes consistant à minimiser le compromis biais-variance entre erreur d'entrainement E_{train} et erreur de généralisation E_{gen} sur lesquelles nous nous appesantirons plus longuement, il est instructif (et sûrement indispensable dans un cours d'apprentissage) de discuter de méthodes historiques naïves (ou du moins élémentaires) mais néanmoins populaires et qui bien souvent sont suffisantes pour le problème traité lorsque celui ci est assez simple. Ces méthodes sont cependant assez empiriques, en ce sens qu'elles ont été créées sans fondement théorique solide ou s'appuient sur des hypothèses qu'on peut considérer en général comme trop fortes. En pratique, elles sont donc difficiles à valoriser et d'autant plus difficiles à comparer théoriquement à des approches qui réalisent réellement une optimisation.

3.1. Vision empirique

3.1.1. Algorithme kNN des plus proches voisins. Si on s'abstrait totalement de la recherche mathématique d'une fonction optimale rendant compte des observations $(x_1, y_1), \dots, (x_n, y_n)$, une première idée intuitive d'allocation d'étiquette ou valeur (y) pour une nouvelle donnée x est de sélectionner pour y l'étiquette y_i du plus proche voisin de x dans l'ensemble $\{x_1, \dots, x_n\}$. Le raisonnement ici consiste simplement à imaginer que, si l'ensemble d'entrainement $\{(x_1, y_1), \dots, (x_n, y_n)\}$ recouvre suffisamment bien le sous-espace de $\mathcal{X} \times \mathcal{Y}$ recherché, x doit nécessairement avoir un voisin x_i proche dont on prendra alors l'étiquette ou la valeur.

Ainsi l'algorithme du *plus proche voisin* consiste à prendre la décision

$$g(x) = y_i, \quad i \in \operatorname{argmax}_{1 \leq j \leq n} \{\|x_j - x\|\}$$

pour une certaine distance $\|\cdot\|$ dans l'espace $\mathcal{X}$. En cas de conflit, lorsque l'argmax n'est pas un singleton, on pourra moyenner les y_i (dans un cadre de régression) ou effectuer un vote majoritaire des y_i les plus représentés dans l'ensemble argmax.

L'algorithme des k *plus proches voisins* (k *nearest neighbors* ou simplement *kNN* en anglais) est une version améliorée de cette approche qui considère non pas seulement le plus proche des x_i mais les k plus proches x_i et opère de manière similaire la décision

$$g(x) = \frac{1}{k} \sum_{i=1}^{k} y_i, \quad i \in \operatorname{argmax}^k_{1 \leq j \leq n} \{\|x_j - x\|\}$$

où $\operatorname{argmax}^k(A)$ représente l'ensemble des arguments des k éléments maximaux de l'ensemble A. Dans le cas d'une classification, il s'agira plutôt de prendre

$$g(x) = \operatorname{mod} \left\{ y_i,\ i \in \operatorname{argmax}^k_{1 \leq j \leq n} \{\|x_j - x\|\} \right\}$$

où l'opérateur mod désigne le "mode" d'un ensemble, à savoir la valeur la plus représentée de l'ensemble. En cas d'égalité, un choix arbitraire est pris (soit aléatoire, soit déterministe comme le plus petit des indices).

Comme indiqué, la méthode kNN est très intuitive lorsque l'espace $\mathcal{X}$ est bien couvert par l'ensemble des données d'entrainement (x_i, y_i). Mais la méthode connait ses limites lorsqu'on sort de ce cadre, et c'est notamment le cas lorsque la dimension intrinsèque p des données est grande. L'écueil le plus flagrant apparait dans le cadre de la classification, lorsque la taille des ensembles de classes et la densité des points x_i dans chaque classe diffèrent : ceci est illustré dans la Figure 3.1.1 dans le cas d'une distribution de deux classes de points en dimension $p = 2$. Mais les limites de la méthode des plus proches voisins apparaissent de manière bien plus surprenante lorsqu'on augmente la dimensions p des données. Dans ce cas on ne visualise plus bien ce qu'il se passe mais un petit calcul simple permet de se rendre compte des catastrophes qui peuvent avoir lieu.

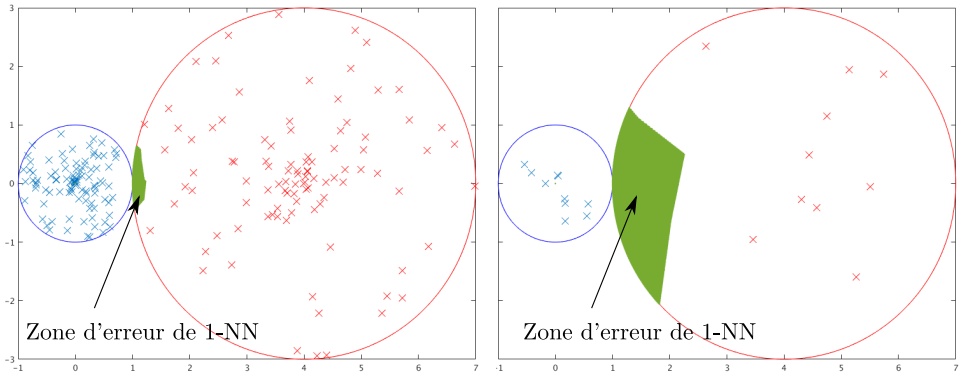

FIGURE 3.1.1. Les limitations de kNN : deux classes échantillonnées avec 200 points (gauche) et 20 points (droite). La plus grande taille de la classe rouge réduit le nombre relatif d'échantillons et induit des proximités favorables à la classe bleue (plus dense et mieux échantillonnée) pour des grandes zones de points de la classe rouge.

LISTING 3.1. Code Matlab de la Figure 3.1.1

```
% On tire les échantillons
n=10;
x1=rand(n,1).*exp(2*pi*1i*rand(n,1));
x2=4+3*rand(n,1).*exp(2*pi*1i*rand(n,1));

figure
hold on;
plot(x1,'x');
plot(x2,'x');

% On trace les deux classes ``vérité terrain''
xs=0:1e-3:1;
C1=exp(2*pi*1i*xs);
C2=4+3*exp(2*pi*1i*xs);
plot(real(C1),imag(C1))
plot(real(C2),imag(C2))

% On explore tout le disque de droite à la recherche de points mal
    classés
```

```
samp=0:1e-3:1;
A=4+3*samp'*exp(2*pi*1i*samp);
B=zeros(length(samp));

for i=1:length(samp)
        for j=1:length(samp)
                B(i,j)=(min(abs(A(i,j)-x1))<min(abs(A(i,j)-x2)));
        end
end

% On représente seulement les points mal classés
AB=A.*B;
plot(real(AB(:)),imag(AB(:)),'.')
```

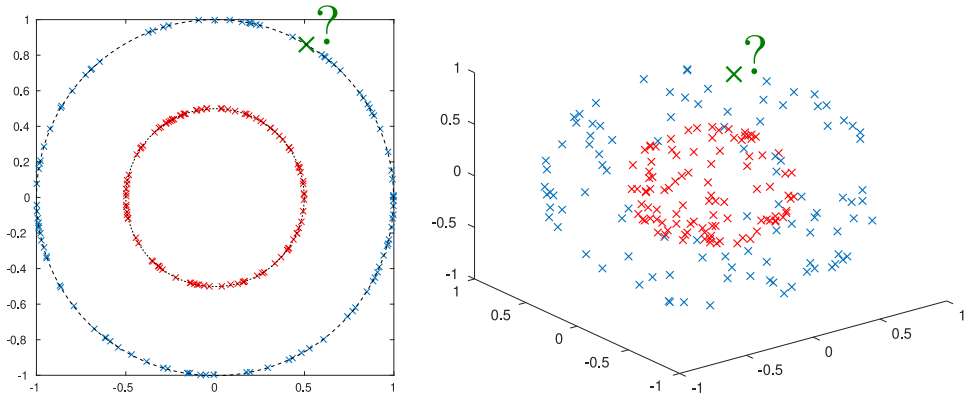

FIGURE 3.1.2. Exemple "jouet" de la classification supervisée de données issues de deux hypersphères concentriques en dimension p (pour $p = 2$ à gauche et $p = 3$ à droite). On échantillonne $n/2 = 100$ données par hypersphère. Alors que clairement la méthode du plus proche voisin ne peut jamais échouer pour $p = 2$, déjà pour $p = 3$ la situation est moins claire. Pour p plus grand, la méthode s'effondre.

Supposons pour faire simple, et surtout pour pouvoir faire des calculs élémentaires, que nous avons affaire à deux classes de données qui sont deux hypersphères $\mathcal{S}(r_1)$ et $\mathcal{S}(r_2)$ de rayons $r_1 > r_2$ de $\mathbb{R}^p$. On tire au hasard $n/2$ points $x_1, \dots, x_n$ dans chaque hypersphère. On peut de faire écrire $x_i = r_1 u_i^{(1)}$ ou $x_i = r_2 u_i^{(2)}$, selon la classe, pour $u_i^{(\cdot)} \in \mathbb{R}^p$ unitaire uniforme sur la sphère unité (on garde en exposant l'indice de la classe juste pour rappel). Supposons maintenant que l'on tire un nouveau point x de la grande hypersphère de rayon r_1 et cherchons à estimer la probabilité d'erreur de l'algorithme 1-NN. Ce problème est visuellement décrit dans la Figure 3.1.2. Une erreur est commise si au moins un point de l'hypersphère de rayon r_2 est plus proche de x que l'ensemble des points de l'hypersphère de rayon r_1. En dénotant $x = r_1 u$ avec $\|u\| = 1$, cet événement peut s'écrire

$$\min_{1 \le i \le n/2} \|r_1 u - r_1 u_i^{(1)}\|^2 > \min_{1 \le j \le n/2} \|r_1 u - r_2 u_j^{(2)}\|^2.$$

En développant, on obtient

$$2r_1^2 \left(1 - \max_{1 \le i \le n/2} u^\mathsf{T} u_i^{(1)}\right) > r_1^2 + r_2^2 - 2r_1 r_2 \max_{1 \le j \le n/2} u^\mathsf{T} u_j^{(2)}.$$

En remarquant qu'un vecteur unitaire $u \in \mathbb{R}^p$ peut s'écrire comme un vecteur gaussien normalisé $u = z/\|z\|$ avec $z \sim \mathcal{N}(0, \frac{1}{p} I_p)$, et que par la loi des grands nombres et le théorème de la limite centrale $\|z\| = 1 + O(1/\sqrt{p})$, on peut très bien approximer $u^\mathsf{T} u_i^{(1)}$ et $u^\mathsf{T} u_j^{(2)}$ par des produits scalaires de vecteurs gaussiens indépendants. Par le théorème de la limite centrale à nouveau $u^\mathsf{T} u_i^{(1)} = \sum_{\ell=1}^p [u]_\ell [u_i^{(1)}]_\ell \sim_{p \gg 1} \mathcal{N}(0, \frac{1}{p})$. Comme les $u_i^{(1)}$ et $u_j^{(2)}$ sont indépendants, conditionnellement à u, les produits scalaires $u^\mathsf{T} u_i^{(1)}$ et $u^\mathsf{T} u_j^{(2)}$ sont donc approximables par des gaussiennes indépendantes dont il s'agit d'extraire les valeurs maximales. La théorie des valeurs extrêmes nous donne la solution : la plus grande valeur extraite d'une suite de n gaussiennes i.i.d. se comporte en $O(\sqrt{\log n})$. On se retrouve donc, sous l'hypothèse où p est grand, avec l'approximation

$$
\begin{aligned}
& 2r_1^2 \left(1 - \max_{1 \le i \le n/2} u^\mathsf{T} u_i^{(1)}\right) > r_1^2 + r_2^2 - 2r_1 r_2 \max_{1 \le j \le n/2} u^\mathsf{T} u_j^{(2)} \\
\underset{p \gg 1}{\Leftrightarrow} & 2r_1^2 \left(1 - O\left(\sqrt{\frac{\log n}{p}}\right)\right) > r_1^2 + r_2^2 - 2r_1 r_2 O\left(\sqrt{\frac{\log n}{p}}\right) \\
\Leftrightarrow & \frac{1}{2}\left(1 + \frac{r_2}{r_1}\right) > O\left(\sqrt{\frac{\log n}{p}}\right).
\end{aligned}
$$

Sous cette hypothèse on voit donc que l'erreur de classification de points de l'hypersphère de grand rayon r_1 tend vers 100% lorsque n est fixe et p augmente. Pire, on voit surtout qu'il faut que n augmente de manière *exponentielle* avec p pour éviter ces erreurs systématiques. [1] Cette observation est reportée dans la simulation de la Figure 3.1.3 pour différentes valeurs de p et n. On observe bien le comportement attendu : pour p grandissant, on passe très vite d'une situation où aucune erreur de classification n'a lieu à une chute exponentiellement rapide de la qualité de classification.

Ce petit calcul est assez important et révèle un point fondamental, constant en apprentissage automatique, qui est celui de la *malédiction des dimensions* : **notre intuition trop portée vers la petite dimension (la seule que nous savons visualiser) nous trompe lorsqu'il s'agit de comprendre ce qu'il se passe en grandes dimensions**.

LISTING 3.2. Code Matlab de la Figure 3.1.1

```
P=50;
score=zeros(P,1);
n=200;
N=1e4;

r1=1;
r2=1/2;

for p=1:P
        % On tire n/2 points de chaque disque de rayons r1 et r2
        x1=randn(p,n/2);
        x1=r1*x1*diag(1./sqrt(diag(x1'*x1)));

        x2=randn(p,n/2);
```

1. Dans la dernière ligne, on a divisé par $r_1 - r_2$ qu'on a supposé d'ordre $O(1)$ (les rayons de changent pas avec p). Ceci explique l'étrange résultat final qui ne dépend pas de $r_1 - r_2$. Pour permettre à r_1, r_2 de dépendre de p, il faut être plus précautionneux sur ce calcul.

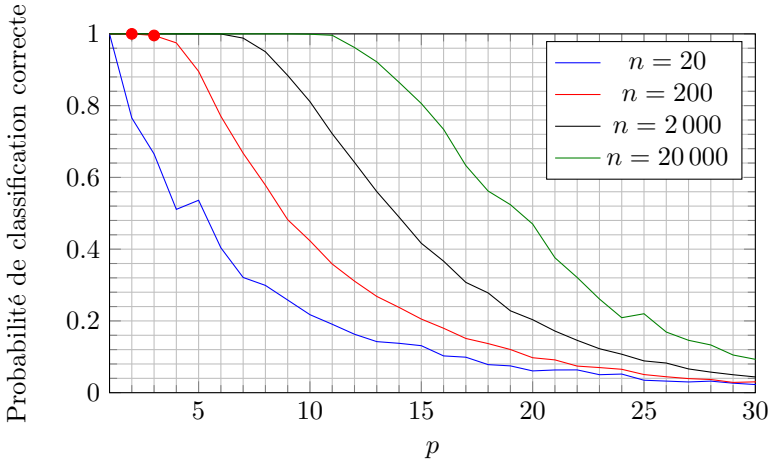

FIGURE 3.1.3. Les limitations de kNN : deux hypersphères de $\mathbb{R}^p$ de rayon $r_1 = 1$ et $r_2 = 1/2$ échantillonnées avec $n/2$ points par classe. On reporte ici la probabilité de classification correcte de 1-NN pour des points tirés de l'hypersphère de rayon r_1, en fonction de la dimension p, et pour différentes valeurs de n. En marqueurs rouges, on surligne les performances pour la configuration de la Figure 3.1.2 (100% pour $p = 2$ et 99% pour $p = 3$ lorsque $n = 200$).

```
        x2=r2*x2*diag(1./sqrt(diag(x2'*x2)));

        % On échantillonne densément le disque de rayon r1
        x=randn(p,N);
        x=x*diag(1./sqrt(diag(x'*x)));

        % On calcule le score de classification correcte de x par 1-NN
        for i=1:N
                score(p)=score(p)+(min(diag((x(:,i)*ones(1,n/2)-x1)'*(
                    x(:,i)*ones(1,n/2)-x1)))<min(diag((x(:,i)*ones(1,n
                    /2)-x2)'*(x(:,i)*ones(1,n/2)-x2))))/N;
        end
end

figure
plot(score);
```

La méthode k-NN est donc surtout appropriée, comme nous le disions, lorsque l'espace ambiant $\mathcal{X}$ est suffisamment densément échantillonné avec au moins $n \sim e^p$, [2] ce qui en pratique impose que l'espace des représentations des données soit de dimension p petite. On n'appliquera concrètement l'algorithme kNN que lorsque p ne dépasse pas la dizaine de représentants.

Un autre problème levé par kNN est celui du choix de la distance entre points : nous avons opté jusque là pour une distance euclidienne entre les points x_i, mais il pourrait être préférable d'adapter la distance en fonction de la situation : en particulier, si le représentant $[x_i]_1$ de x_i est bien plus informatif sur la classe de

2. Nous n'avons prouvé cette relation qu'à travers un exemple, mais cet exemple illustre parfaitement le problème de "densité" lors de l'échantillonnage des espaces en grande dimension.

x_i que $[x_i]_2$, il paraitrait pertinent de choisir une distance pondérée qui prenne cette information en compte. Malheureusement, comme il s'agit en apprentissage "automatique" de laisser la machine automatiser le calcul, il est dommage d'intervenir manuellement sur la distance. Dans la section suivante, nous allons justement discuté d'un algorithme qui a le don, en cherchant toujours à classer les données, de générer au passage une distance adaptée et a fortiori utile et interprétable pour l'utilisateur, entre les vecteurs de représentations.

3.1.2. Arbres de décision et forêts aléatoires. Les arbres de décision sont un concept assez élémentaire et qui, dans un sens, vivent à mi-chemin entre l'apprentissage automatique dans sa vision "boite noire" (à savoir, comme pour les emblématiques réseaux de neurones, la machine se charge de faire le travail sans jamais nous révéler ce qu'elle a vraiment fait) et la classification humaine naturelle et même sémantique de jeux de données.

L'approche des arbres de décision suit de fait un formalisme qui diffère a priori un peu du nôtre dans ce cours, mais que l'on peut rendre cohérent. Pour cela, on suppose que chaque donnée $x \in \mathcal{X}$ admet un système de représentation $\phi(x)$ discret et donc, pour simplifier les notations, on va à nouveau confondre x avec son système $\phi(x)$ de représentants. Ces représentants sont traités individuellement, comme ils consistent en une information sur la donnée ; ainsi, avec notre choix de notation, $[x]_1, \dots, [x]_p$ sont p caractéristiques de la donnée $x \in \mathcal{X}$ où $\dim(\mathcal{X}) = p$, et chacune apporte une information spécifique qui permet de classer ou régresser x.

Il s'agit alors de construire un arbre qui va successivement extraire, à chaque nœud, l'information parmi les p la plus discriminante pour classifier ou la plus pertinente pour régresser. À chaque étape de cette procédure, on élimine de la liste des p informations celles qui ont déjà servi, et on descend dans l'arbre jusqu'à ce qu'une classification parfaite soit atteinte (toutes les données des branches issues d'un nœud appartiennent à la même classe) ou qu'un score satisfaisant de régression soit obtenu. La métrique qui permet de sélectionner la représentation parmi les p la plus discriminante ou pertinente est laissée au choix de l'utilisateur, mais on agit en général de manière à forcer les branches issues de chaque nœud à être le plus homogène en classes possible (pour la classification) ou ayant une variance minimale (pour la régression).

Concrètement, le mécanisme pour un arbre binaire (de chaque nœud on extrait deux branches) pour un problème de classification à k classes ($\mathcal{Y} = \{1, \dots, k\}$) est le suivant :

(1) étant donné un ensemble d'apprentissage $(x_1, y_1), \dots, (x_n, y_n)$, avec x_i un vecteur à p éléments binaires (à savoir $[x_i]_j \in \{0, 1\}$), on tire de la racine de l'arbre deux branches : à chaque branche on associe un sous-ensemble des $\{x_i\}$ et son complémentaire de telle sorte que la probabilité de mauvaise classification d'un (x, y) arbitraire de l'ensemble de test (on suppose avoir accès à un tel ensemble ou on en extrait une partie de la base d'entrainement) soit minimale. Ce faisant, on maximise donc l'homogénéité des deux branches. Le sous-ensemble des $\{x_i\}$ n'est cependant pas choisi au hasard, et il ne s'agit évidemment pas d'énumérer les 2^n partitions possibles. Au contraire, on divisera l'ensemble $\{x_1, \dots, x_n\}$ suivant l'élément binaire parmi les p éléments des x_i qui minimise l'erreur de classification : il s'agit donc d'effectuer p tests et non 2^n.

(2) une fois les deux branches créées, on réitère le processus sur chaque nouvelle branche *ou* sur toutes les branches d'un même niveau (qui contiennent alors une partition de plus en plus affinée de l'ensemble $\{x_1, \dots, x_n\}$ de départ), et ce de manière itérative jusqu'à atteindre des feuilles dont les éléments x_i associés sont de la même classe (les y_i sont égaux).

Évidemment, quelques raffinements sont à apporter à cet algorithme général. Lorsque les entrées $[x_i]_j$ ne sont pas binaires mais réelles, on décidera d'un seuil (la valeur moyenne des $[x_i]_j$ par exemple) pour créer une décision binaire. On peut également créer des arbres ayant plus de deux branches par nœud, auquel cas des décisions multiples peuvent être effectuées ; selon le critère choisi, on peut même autoriser un nombre de branche qui évolue dynamiquement (si $[x_i]_1$ prend 2 valeurs et $[x_i]_2$ prend trois valeurs, selon le choix de la variable pertinente, on tirera deux ou trois branches du nœud en cours). Nous avons évoqué dans le deuxième point la possibilité de descendre dans l'arbre soit selon chaque branche indépendamment des autres, soit simultanément sur les différentes branches d'un même niveau : l'avantage du premier cas est celui d'assurer une profondeur locale la plus courte possible de l'arbre mais au détriment d'épuiser les p paramètres des données de manière différenciée (et plus on descend dans l'arbre, de manière assez aléatoire) selon la branche de l'arbre, tandis que le deuxième cas assure que l'on utilise à chaque niveau le même indice $[\cdot]_j$ des paramètres ayant pour effet de stabiliser l'arbre. Lorsqu'il s'agit non plus de classification mais de régression, le critère n'est plus une minimisation de l'erreur de classification, mais généralement une minimisation de la variance des y_i associés à chaque sous-branche.

L'avantage précieux des arbres de décision est que, lorsque les p représentants des données x ont un sens physique interprétable pour l'expérimentateur (par exemple, $[x]_j$ peut être la réponse à une question : "x est-il un homme ?", "x a-t-il plus de 30 ans ?"), l'arbre obtenu permettra visuellement de rendre compte des données les plus discriminantes (en haut de l'arbre) et des données les moins discriminantes (près des feuilles) de la catégorisation. C'est en ce sens qu'il semble plus pertinent d'optimiser chaque niveau de l'arbre conjointement et non indépendamment (item 2. précédent) pour garder un sens au processus séquentiel de filtrage. Il existe d'ailleurs des logiciels graphiques permettant de manière très élégante de visualiser des jeux de données parfois colossaux (plusieurs dizaines ou centaines de milliers de points) par regroupements choisis à différents niveaux d'un arbre de décision : on peut même visuellement faire apparaitre des distances différentes entre niveaux de l'arbre pour marquer la qualité de filtration de l'information à ce niveau (si un niveau est très éloigné du niveau inférieur, on comprend que le niveau inférieur effectue un filtrage bien plus marginal). L'autre avantage, commun à de nombreuses méthodes par arbres, est celle d'un gain logarithmique de traitement des données, une fois l'arbre construit : en effet, la profondeur typique d'un arbre construit sur p éléments discriminants est d'ordre $\log p$.

Par contre, la liste des limitations et inconvénients de la méthode est longue. L'outil des arbres binaires reste fortement heuristique : rien ne garantie notamment que l'arbre, qui est construit de manière itérative (ce qu'on appelle parfois des méthodes *gourmandes* ou *greedy* en anglais), est optimal en quelque sens que ce soit. Rien n'assure par exemple que l'arbre construit est le moins profond que l'on puisse obtenir si on avait optimisé *conjointement* et non successivement chaque couche. Pire, l'algorithme est très sensible au jeu de données d'apprentissage : l'ajout ou la délétion d'un seul élément de l'ensemble d'entrainement peut modifier complètement la structure de l'arbre, en changeant notamment l'ordre avec lequel les représentations sont tirées les unes après les autres ; par effet "boule de neige", tout l'arbre peut être énormément modifié. Également, proche des feuilles, le filtrage des informations pertinentes devient fortement dépendant des données de départ, et donc moins informatif, ou du moins moins fiable. Il est malheureusement difficile pour l'utilisateur d'évaluer cette fiabilité.

Pour remédier à ces problèmes de stabilité des arbres de décision, une technique qui fonctionne très bien est celle dite des *forêts aléatoires* (*random forests* en

anglais). Cette méthode nous donne l'opportunité d'introduire au passage une méthode très populaire en apprentissage automatique, que l'on appelle le *boostrapping*, aussi appelée *bagging* (il ne semble pas y avoir de traduction claire en français), et que nous introduisons formellement dans la remarque ci-dessous avant de revenir aux forêts aléatoires.

Remarque 7 (La méthode du bootstrapping). Nous l'avons longuement évoqué dans le chapitre précédent, une difficulté importante en apprentissage automatique consiste à limiter les effets de surapprentissage tout en assurer une bonne qualité de généralisation des algorithmes : c'est le fameux *compromis biais-variance* (on rappelle : le biais est induit par le manque d'équation des modèles $g \in \mathcal{G}$ des algorithmes à la fonction f à imiter, et la variance est induite au contraire par une trop bonne adéquation des $g(x_i)$ aux y_i de la base d'apprentissage lorsque la famille de modèles $\mathcal{G}$ est trop riche). Une façon de simultanément réduire le biais et la variance est d'effectuer un *bagging* des données. Cette technique peut sembler étrange : à partir des n données $X = [x_1, \dots, x_n]$, on va créer m échantillons de n données $X_1^\star, \dots, X_m^\star$ avec $X_i^\star = [x_{\varphi_i(1)}, \dots, x_{\varphi_i(n)}]$ où $\varphi_i \in \{1, \dots, n\}^n$ est un vecteur à entrées aléatoires et uniformes de $\{1, \dots, n\}$. C'est-à-dire qu'on va tirer m fois *avec remise* n points de $\{1, \dots, n\}$ pour créer des ensembles $X_i^\star$. On va alors utiliser chaque $X_i^\star$ comme s'il s'agissait du jeu de données d'entrainement et créer à partir de lui une fonction de décision $g_i^\star$. Pour une nouvelle donnée x, la décision finale du bootstrapping consistera alors en une moyenne (pour la régression) ou un vote majoritaire (pour la classification) des m valeurs $g_i^\star(x)$. Cette méthode est à nouveau très heuristique et n'apporte théoriquement rien sur des problèmes statistiques contrôlés, car en effet on n'apporte aucune information nouvelle, on se contente de rééchantillonner les mêmes points, qui plus est avec remise !, sans créer d'information supplémentaire. Mais le gain est en fait réel car il permet essentiellement deux choses :

- il réduit le surapprentissage en évitant de "coller" au jeu de données $X = [x_1, \dots, x_n]$ grâce à une diversification des entrées par les $X_i^\star$. En particulier, si x_j est un point de $X_i^\star$ et qu'on a forcé, de par la méthode choisie, $g_i^\star(x_j)$ à être très proche de y_j, il existe probablement des jeux de données $X_{i'}^\star$ qui ne contiennent pas x_j et pour lesquels $g_{i'}^\star(x_j)$ n'est pas très proche de y_j. En moyennant les sorties de ces jeux de données, on modère le surapprentissage ;
- il diminue potentiellement le biais de l'algorithme en créant une décision g qui, en tant que moyenne des $g_i^\star$, ne fait pas forcément partie de la classe $\mathcal{G}$ mais d'un ensemble "lissé" (une enveloppe convexe si on veut) des éléments de $\mathcal{G}$.

C'est en cela que, peut-être un peu paradoxalement à première vue, on peut améliorer un estimateur g de la fonction f recherchée en n'ajoutant aucune information supplémentaire, mais simplement en rééchantillonnant les données.

En fait, la méthode de bootstrapping va aujourd'hui plus loin, en proposant de "boostrapper" des algorithmes et non plus seulement les données d'apprentissage. L'idée consiste ici à moyenner différentes fonctions de décision $g_1, \dots, g_m$ associées à des algorithmes potentiellement radicalement différents. Ainsi, le surapprentissage induit par l'un de ces algorithmes peut être compensé par un autre, et la famille des fonctions g ainsi obtenues est maintenant l'enveloppe convexe d'un ensemble de m familles $\mathcal{G}_1, \dots, \mathcal{G}_m$ d'algorithmes. C'est assez "triste" théoriquement, mais il s'avère que beaucoup de compétitions en IA sont remportées par de tels *meta-algorithmes* qui mélangent par boostrapping un grand ensemble d'algorithmes, eux-mêmes possiblement bootstrappés par rééchantillonnage de leurs données. Par "triste", nous pointons ici le fait qu'au contraire de la plupart des disciplines scientifiques formellement établies et dont les optima et les moyens mathématiquement fins de

les atteindre sont souvent connus, en apprentissage automatique les méthodes qui "marchent bien" en pratique consistent en un mélange éclectique et sans réel fondement d'approches diverses.

Ayant introduit la méthode du bootstrapping, nous sommes maintenant en mesure de définir une *forêt aléatoire*. Par forêt aléatoire, on comprend tout simplement un ensemble, souvent très large, d'arbres de décision "bootstrappés". Précisément, pour un ensemble d'apprentissage $\{(x_1, y_1), \dots, (x_n, y_n)\}$ donné, que l'on divisera en général en données d'entrainement et données de test, on obtient de manière déterministe par la méthode décrite plus haut un arbre de décision. En boostrappant les données $x_1, \dots, x_n$ et en allouant aléatoirement les données d'entrainement et les données d'apprentissage, on peut obtenir un très grand nombre m d'arbres aléatoires, collectivement appelés *forêt aléatoire*. Pour une nouvelle donnée d'entrée x, l'estimation de la sortie y associée consiste alors simplement en la moyenne ou le vote majoritaire (c'est-à-dire le mode) des m issues des arbres de décision.

Cette méthode est connue comme étant très puissante parce qu'elle parvient notamment à "lisser" les effets aléatoires apparaissant au plus profond de chacun des arbres, réduisant ainsi la variance de la méthode (le surapprentissage). Prendre la moyenne des arbres n'est également pas équivalent à un unique arbre "conjoint", mais plutôt à une sorte d'arbre pondéré conjoint dans lequel, au lieu de ne considérer qu'un seul attribut par p à chaque descente d'un nœud au suivant, on effectue une combinaison linéaire de tous les attributs. Cette "régression par niveau" peut être apparentée (de loin) à une analyse en composantes principales du jeu de données (la composante dominante étant identifiée en haut de l'arbre, et les modes moins énergétiques plus bas). [3]

Malheureusement les forêts aléatoires font perdre certains des attributs utiles des arbres de décisions : que dire de la sélection de la variable "$1/3\times$ sexe de l'individu $+1/6\times$ poids de l'individu supérieur à 50kg $+\dots$" comme valeur discriminante en haut de l'arbre, suivie d'une combinaison entièrement différente dans le niveau inférieur, notamment quand p est grand ? On perd cette **explicabilité de l'IA, qui est d'ailleurs un domaine de recherche remis au goût du jour récemment**.

Conclusion de la section. Sans l'avoir explicitement mis en avant, on voit bien que les méthodes kNN et d'arbres de décision, sans être antagonistes, s'appliquent préférentiellement à des champs pratiques distincts : on souhaiterait plus naturellement utiliser des arbres de décision lorsqu'il existe des "attributs à décider", à savoir lorsque les entrées des vecteurs de données $x \in \mathcal{X}$ ont une signification physique ou sémantique claire et distinctes entrée-par-entrée. L'algorithme kNN lui est généralement indifférent du contenu des vecteurs de données (ou de leur représentation) et utilise une distance pré-établie dans l'espace $\mathcal{X}$: on aurait notamment du mal à comparer la distance entre l'attribut binaire "homme/femme" et entre les attributs continus "taille", "poids". On a également vu que kNN n'est pas adapté à des données de dimensions grandes, à moins que le nombre de ces données ne soit exponentiellement plus grand, et que la variabilité intra-classes ne soit pas trop grande (à savoir, que certaines classes ne soient pas petites et denses alors que d'autres sont grandes et mal échantillonnées) : mais même dans ce cas, est-il bien clair ce que signifie être proche ou loin dans un espace de dimension plus grande que $p = 10$?

Ce que ces méthodes ont cependant en commun, c'est une approche *statistique* des données, en ce sens que les données observées forment elles-mêmes "l'information complète" qu'il s'agit alors d'exploiter au mieux : on nomme souvent les algorithmes

3. La comparaison s'arrête là car les composantes en question n'ont pas à être orthogonales entre elles, et il n'est même pas bien clair que le bootstrapping pousse les premières branches à s'aligner dans de telles directions.

qui en résultent des méthodes *pilotée par les données* (ou *data-driven* en anglais). Si l'approche est intellectuellement "honnête" (on n'utilise que l'information à laquelle on a accès), elle pose des problèmes fondamentaux d'analyse de performances et de comparaison des méthodes (comme on n'a aucun accès à la mesure qui a engendré les données) et on reste alors dans un "flou heuristique". Dans la section suivante, nous allons plutôt considérer une approche probabiliste, généralement moins préférée en apprentissage, qui elle génère des algorithmes et méthodes *basées sur des modèles* (ou *model-based* en anglais) : ici on considèrera plutôt que les données ne sont que des tirages aléatoires d'une mesure sous-jacente que l'on cherchera à estimer (ou plutôt à "préciser") et sur laquelle on aura généralement un *a priori* de structure. Par exemple, comme on l'imagine bien, pour se simplifie la vie et les calculs, dans beaucoup de cas on fera l'hypothèse que les données sont des vecteurs gaussiens dont il s'agira d'estimer moyennes et covariances. Cette approche est plus satisfaisante mathématiquement en ce sens que, à la difficulté éventuelle de calcul près, si on connait les lois des jeux de données, on est capable en théorie d'évaluer les performances de nos algorithmes. Le bât blesse par contre lorsqu'il s'agit de justifier que le modèle *a priori* : pourquoi les représentations des images de chiens seraient bien modélisées par des vecteurs gaussiens ?

Comme nous le verrons brièvement à travers des remarques, les deux approches sont vraiment très différentes dans l'esprit et la philosophie de travail (et cela n'a eu de cesse de créer un fossé entre statisticiens et probabilistes de tous temps) mais **tendent à rejoindre lorsqu'il s'agit de traiter de données réelles de grandes dimensions**. C'est là l'un des points d'accroche des statistiques et probabilités modernes qui promettent peut-être d'unifier enfin le domaine, entraînant dans ce sillon une compréhension améliorée des algorithmes d'apprentissage modernes.

3.2. Vision probabiliste

Nous entamons donc cette section avec un regard différent sur les données : une vision probabiliste selon laquelle les données sont des observations issues d'une loi a priori, ou tout d'une moins d'une des lois composant une certaines famille de mesures : par exemple un vecteur gaussien mais de moyenne et covariance inconnus, ou alors plus généralement un vecteur dont la loi de probabilité satisfaisait une propriété de symétrie ou d'invariance (par exemple $p(x) = q(\|x - m\|)$ dans le cas d'une symétrie radiale).

3.2.1. Estimateur de Bayes. Pour simplifier les choses, on se place ici dans un contexte de classification où les sorties $y \in \mathcal{Y}$ sont discrètes avec $\mathcal{Y} = \{1, \dots, k\}$. Supposons que les données d'entrainement $(x_1, y_1), \dots, (x_n, y_n)$ soient des observations indépendantes d'une loi de probabilité μ_{xy} (une fonction mesurable de $\mathcal{X} \times \mathcal{Y}$ dans $[0, 1]$). On va supposer par ailleurs que $\mu_{xy} = \mu_{xy;\theta}$ fait partie d'une famille paramétrée de mesures $\{\mu_{xy;\tilde{\theta}}\}_{\tilde{\theta} \in \Theta}$, mais que θ est inconnu : c'est typiquement le cas par exemple lorsque x est gaussien de moyenne et variance rassemblées dans θ mais que ces moyenne et variance sont a priori non connues. Enfin, dernière hypothèse, on suppose que $\mu_{xy;\theta}$ est en fait un *modèle de mélange* de lois de probabilités de la forme

$$\mu_{xy;\theta} = \frac{1}{k} \sum_{i=1}^{k} \pi_i \mu_{x;\theta_i}$$

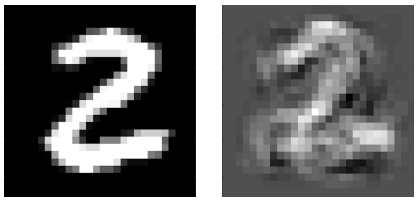

FIGURE 3.2.1. Image réelle de taille 28×28 ($= 784$ pixels) de la populaire base MNIST (chiffres écrits à la main) et une génération aléatoire d'un vecteur gaussien de taille $p = 784$ ayant les mêmes statistiques (moyenne et covariance) que l'image réelle (ces statistiques sont calculées empiriquement à partir des $\sim 10\,000$ images de la base de donnée). L'hypothèse gaussienne est clairement abusive, et pourtant, pour cette valeur assez grande de p, de nombreux algorithmes étudiés dans le cours ont exactement les mêmes performances de classification sur l'image réelle et sur le vecteur gaussien.

où $\pi_i = \mathbb{P}(y = i)$ et θ_i est le paramètre de la loi $\mu_{xy;\theta}$ lorsque $y = i$. Pour rester dans le cas gaussien, on peut par exemple supposer que

$$\mu_{xy;\theta} \sim \frac{1}{2}\mathcal{N}(\mu, I_p) + \frac{1}{2}\mathcal{N}(-\mu, I_p)$$

où ici $\pi_1 = \pi_2 = \frac{1}{2}$ et $\theta_1 = \mu$, $\theta_2 = -\mu$. Il s'agit du populaire et le plus élémentaire exemple de *mélange gaussien* (*Gaussian mixture model* en anglais).

REMARQUE 8 (Le mélange gaussien). Très très utilisé dans l'approche probabiliste de l'apprentissage automatisé, le *mélange gaussien*, ou *k-mélange gaussien*, suppose que la loi conjointe μ_{xy} de données (x, y) avec $x \in \mathcal{X} = \mathbb{R}^p$ et $y \in \mathcal{Y} = \{1, \dots, k\}$ s'écrit sous la forme

$$\mu_{xy} = \sum_{i=1}^{k} \pi_i \mathcal{N}(\mu_i, C_i)$$

pour $\pi_1 + \ldots + \pi_k = 1$, $\mu_1, \dots, \mu_k \in \mathbb{R}^p$ et $C_1, \dots, C_k \in \mathbb{R}^{p \times p}$ des matrices semi-définies positives. Ce modèle est souvent considéré comme un *modèle jouet* (*toy model* en anglais) en ce sens qu'il permet de faire des petits calculs de performances d'algorithmes, mais malheureusement en restant déconnecté de la réalité des données : on s'en convainc a priori facilement en visualisant en une ou deux dimensions ce mélange de lois en "cloches" (déformées par la covariance) qui n'a aucune raison de représenter une quelconque réalité des données. Néanmoins, lorsque la dimension p grandit, de manière très surprenante, bien que le modèle de mélange gaussien reste vraisemblablement irréaliste (un vecteur gaussien n'a toujours rien à voir avec une vraie donnée, même en grande dimension, comme on peut le voir en Figure 3.2.1), on peut démontrer que la plupart des algorithmes que nous allons étudier dans ce cours se comportent avec des vraies données *comme si elles étaient un simple mélange gaussien*. La Figure 5.4.2, étudiée plus tard lorsque nous aurons introduit plus formellement la notion de noyaux, permet de s'en rendre compte, ici pour des données encore plus proche de la réalité du domaine de la vision assistée par ordinateur.

Notons, et c'est un point important ici, que contrairement à la vision statistique où la sortie y d'un couple (x, y) est supposée être de la forme $y = f(x)$ pour une certaine fonction f potentiellement très complexe, on suppose dans cette vision probabiliste que y est un paramètre déterministe du modèle de mélange et que

la loi de x est donnée par la valeur de y (plutôt que l'inverse). Les deux notions ne sont en général pas compatibles, car les supports des lois $\mu_{x;\theta_i}$ peuvent être d'intersection non vide : et donc plusieurs x de lois $\mu_{x;\theta_i}$ distinctes peuvent prendre la même valeur : il n'existe alors pas de fonction f telles que $f(x) = y$; dans le cas mono ou multivarié gaussien, on a même pour intersection l'espace $\mathbb{R}^p$ tout entier.

Équipés de ces hypothèses, effectuons un petit calcul bayésien simple pour nous rendre déjà compte des difficultés qui apparaissent très vite dans ce cas. Comme nous l'avons évoqué, l'avantage du modèle probabiliste est qu'en théorie "tout se calcule". Notamment ici, sur l'observation d'une donnée x d'étiquette y inconnue, on va pouvoir choisir la valeur de y la plus probable au sens du maximum de vraisemblance, étant données toutes les observations connues. En l'occurrence, on obtient alors

$$\begin{aligned}
&\mathbb{P}(y|x,\{(x_i,y_i)\}) \\
&= \int_{\tilde{\theta}\in\Theta} \mathbb{P}(y|x,\{(x_i,y_i)\},\tilde{\theta})\mathbb{P}(\tilde{\theta}|x,\{(x_i,y_i)\})d\tilde{\theta} \\
&= \int_{\tilde{\theta}\in\Theta} \frac{\mathbb{P}(x|y,\{(x_i,y_i)\},\tilde{\theta})\mathbb{P}(y|\{(x_i,y_i)\},\tilde{\theta})}{\sum_{\tilde{y}=1}^{k}\mathbb{P}(x|\tilde{y},\{(x_i,y_i)\},\tilde{\theta})\mathbb{P}(\tilde{y}|\{(x_i,y_i)\},\tilde{\theta})} \frac{\mathbb{P}(x,\{(x_i,y_i)\}|\tilde{\theta})\mathbb{P}(\tilde{\theta})}{\int_{\tilde{\tilde{\theta}}\in\Theta}\mathbb{P}(x,\{(x_i,y_i)\}|\tilde{\tilde{\theta}})\mathbb{P}(\tilde{\tilde{\theta}})d\tilde{\tilde{\theta}}}d\tilde{\theta}
\end{aligned}$$

où nous n'avons fait qu'appliquer la règle de Bayes ici. Du fait de la structure du modèle, beaucoup de choses se simplifient : notamment, étant donné θ et y, la loi de x est parfaitement connue (c'est $\mu_{x;\theta_i}$). Comme par ailleurs les $\{(x_i,y_i)\}$ sont indépendants de (x,y), on se ramène simplement à

$$\begin{aligned}
\mathbb{P}(y|x,\{(x_i,y_i)\}) &= \int_{\tilde{\theta}\in\Theta} \frac{\mu_{x;\tilde{\theta}_y}(x)\pi_y}{\sum_{\tilde{y}=1}^{k}\mu_{x;\theta_{\tilde{y}}}\pi_{\tilde{y}}} \frac{\mathbb{P}(x,\{(x_i,y_i)\}|\tilde{\theta})\mathbb{P}(\tilde{\theta})}{\int_{\tilde{\tilde{\theta}}\in\Theta}\mathbb{P}(x,\{(x_i,y_i)\}|\tilde{\tilde{\theta}})\mathbb{P}(\tilde{\tilde{\theta}})d\tilde{\tilde{\theta}}}d\tilde{\theta} \\
&\propto \int_{\tilde{\theta}\in\Theta} \mu_{x;\tilde{\theta}_y}(x)\pi_y\mathbb{P}(x,\{(x_i,y_i)\}|\tilde{\theta})\mathbb{P}(\tilde{\theta})d\tilde{\theta}. \qquad (3.2.1)
\end{aligned}$$

Comment comprendre cette formule ? Pour une donnée x observée, on trouve tout simplement que le y associé est celui qui maximiserait la quantité $\mu_{x;\theta_y}(x)\pi_y$, ce qui est cohérent. Mais, l'ensemble des θ_y ($\{\theta_1,\dots,\theta_k\}$) des paramètres du modèle statistique n'est pas connu. Il faut donc les "tester tous" et les pondérer par leurs probabilités respectives étant donné ce qu'on connait, à savoir les $\{(x_i,y_i)\}$ de la base d'apprentissage. Ces $\{(x_i,y_i)\}$ nous servent donc ici, non pas comme référence pour créer une erreur d'entrainement qu'on chercherait à minimiser, mais plutôt comme références pour estimer le paramètre θ inconnu de la loi de (x,y).

Pour poursuivre le calcul, il faut cependant que la loi $\mathbb{P}(x,\{(x_i,y_i)\}|\tilde{\theta})$ soit connue, ce qui commence à devenir très artificiel. On pourrait par exemple demander, dans le cas du mélange gaussien, à ce que que $\tilde{\theta} = \{\mu,-\mu\}$ pour μ un vecteur tiré uniformément sur l'hypersphère de $\mathbb{R}^p$: dans ce cas, $\mathbb{P}(\tilde{\theta})$ est constant et $\prod_{i=1}^{n}\mathbb{P}(x_i|y_i,\tilde{\theta})$ (qui interviendrait dans le développement du terme résiduel) est simplement le produit de densités d'un vecteur gaussien. Mais il restera encore une intégrale multivariée à évaluer qui ne pourra s'effectuer que numériquement, ce qui devient très vite impossible dès que p est supérieur à trois ou quatre.

Autant dire que, même dans un cas très simple, l'optimal bayésien est compliqué à mettre en place. Pour des cas plus exotiques, la méthode est tout simplement exclue. Dans la suite, nous allons utiliser une façon détournée mais *a priori* pertinente, de simplifier le problème et d'éviter les intégrations, spécifiquement dans le cas d'un mélange gaussien (dont nous rappelons la définition en Remarque 8).

3.2.2. Analyse discriminante : LDA et QDA. Si on fait l'hypothèse d'un k-mélange gaussien pour les données, en suivant les notations de la Remarque 8, la mesure μ_{xy} des couples de données (x, y) est de la forme

$$\mu_{xy} \equiv \sum_{i=1}^{k} \pi_i \mu_{x;\theta_i} = \sum_{i=1}^{k} \pi_i \mathcal{N}(\mu_i, C_i)$$

avec $\theta_j = \{(\mu_j, C_j)\}$ et où, bien évidemment et c'est là toute la difficulté, les μ_j et C_j ne sont pas connus : seuls les couples (x_i, y_i) et l'observation x sont accessibles.

Et donc, d'après le calcul bayésien précédent, on aimerait idéalement pouvoir évaluer l'intégrale

$$\mathbb{P}(y|x, \{(x_i, y_i)\}) \propto \int_{\tilde{\theta}\in\Theta} \mu_{x;\tilde{\theta}_y}(x)\pi_y \mathbb{P}(x, \{(x_i, y_i)\}|\tilde{\theta})\mathbb{P}(\tilde{\theta})d\tilde{\theta}$$

où ici $\tilde{\theta}$ est l'ensemble des moyennes $\mu_1, \ldots, \mu_k$ et covariances $C_1, \ldots, C_k$ admissibles, et $\mathbb{P}(\tilde{\theta})$ représenterait une loi a priori de ces moyennes et covariances. En l'occurrence, ce calcul semble compliqué mais, surtout, il parait très artificiel de devoir générer une telle loi a priori pour les μ_i et C_i.

3.2.2.1. *QDA.* L'*analyse discriminante* part de l'idée suivante (c'est du moins une interprétation possible) : au lieu de calculer l'intégrale sur la mesure $\mathbb{P}(\tilde{\theta})$ de $\tilde{\theta}$, on remplace $\mathbb{P}(\tilde{\theta})$ par une mesure de Dirac en un point unique $\tilde{\theta}$ qui "estime au mieux" le vrai paramètre $\theta = \{\mu_1, \ldots, \mu_k, C_1, \ldots, C_k\}$ du modèle. Pour cela, une méthode très simple est d'estimer les μ_i par la moyenne empirique $\hat{\mu}_i$ des données de la classe i (à savoir l'ensemble des x_j tels que $y_j = i$)

$$\hat{\mu}_i = \frac{1}{|\{y_j = i\}|} \sum_{j|y_j=i} x_j$$

et les C_i par la covariance empirique

$$\hat{C}_i = \frac{1}{|\{y_j = i\}|(|\{y_j = i\}| - 1)} \sum_{j|y_j=i} (x_j - \hat{\mu}_i)\,(x_j - \hat{\mu}_i)^{\mathsf{T}}\,.$$

On obtient de fait une estimation assez "crue" de la probabilité a posteriori $\mathcal{P}(y|x, \{(x_i, y_i\})$, qu'on préfèrera même appeler une heuristique, à travers l'estimation dite *plug-in* (parce qu'on remplace dans la formule le paramètre inconnu directement par son estimée)

$$\mu_{x;\{\hat{\mu}_y,\hat{C}_y\}}\pi_y = \frac{\pi_y}{\sqrt{2\pi|\hat{C}_y|}} \exp\left(-\frac{1}{2}(x - \hat{\mu}_y)^{\mathsf{T}}\hat{C}_y^{-1}(x - \hat{\mu}_y)\right).$$

On choisira alors comme estimateur de l'étiquette y la valeur parmi $\{1, \ldots, k\}$ qui maximise $\mu_{x;\{\hat{\mu}_y,\hat{C}_y\}}\pi_y$. Pour identifier la valeur de y atteignant le maximum, on peut évaluer tous les $\mu_{x;\{\hat{\mu}_y,\hat{C}_y\}}\pi_y$, avec $y = 1, \ldots, k$, ou alternativement effectuer une comparaison paire-à-paire (ce qui est le plus pertinent lorsqu'on a affaire à $k = 2$ classes). Dans ce cas, disons pour simplifier les notations que nous souhaitons comparer les hypothèses $y = 1$ et $y = 2$; on sera alors amenés à identifier le signe de

$$\begin{aligned}\mu_{x;\{\hat{\mu}_1,\hat{C}_1\}}\pi_1 - \mu_{x;\{\hat{\mu}_2,\hat{C}_2\}}\pi_2 = &\frac{\pi_1}{\sqrt{2\pi|\hat{C}_1|}} \exp\left(-\frac{1}{2}(x - \hat{\mu}_1)^{\mathsf{T}}\hat{C}_1^{-1}(x - \hat{\mu}_1)\right)\\ &- \frac{\pi_2}{\sqrt{2\pi|\hat{C}_2|}} \exp\left(-\frac{1}{2}(x - \hat{\mu}_2)^{\mathsf{T}}\hat{C}_2^{-1}(x - \hat{\mu}_2)\right).\end{aligned}$$

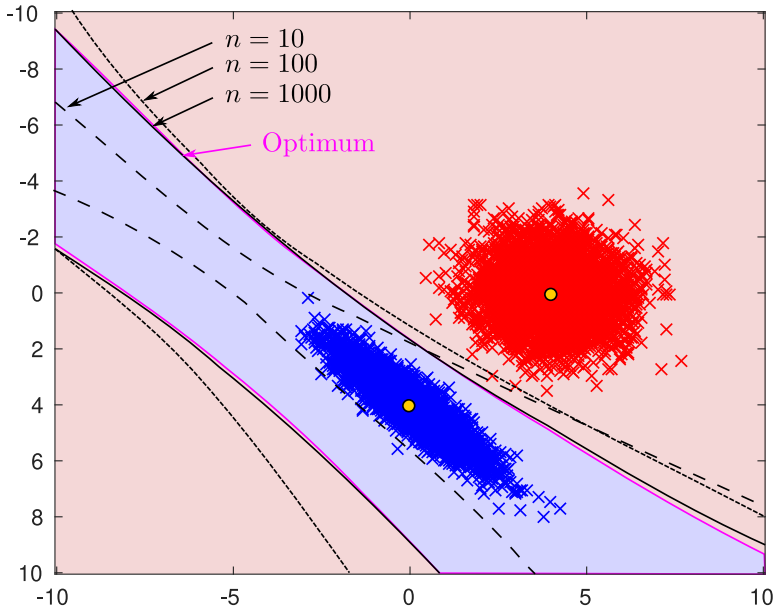

FIGURE 3.2.2. Performance de la méthode QDA pour un mélange de deux gaussiennes bi-variées. Les zones de décision en pointillés montrent que n doit être très grand (ici au delà de 10^3) pour obtenir des seuils de décision précis.

On aura notamment une *frontière de décision* lorsque ce terme est nul : une limite en deçà de laquelle l'hypothèse $y = 2$ est favorisée et au delà de laquelle l'hypothèse $y = 1$ sera retenue. On retiendra donc $y = 1$ par rapport à $y = 2$ si

$$\frac{\pi_1}{\sqrt{2\pi|\hat{C}_1|}} \exp\left(-\frac{1}{2}(x-\hat{\mu}_1)^\mathsf{T}\hat{C}_1^{-1}(x-\hat{\mu}_1)\right) > \frac{\pi_2}{\sqrt{2\pi|\hat{C}_2|}} \exp\left(-\frac{1}{2}(x-\hat{\mu}_2)^\mathsf{T}\hat{C}_2^{-1}(x-\hat{\mu}_2)\right)$$

$$\Leftrightarrow \log\frac{\pi_1}{\pi_2} - \frac{1}{2}\log\frac{|\hat{C}_1|}{|\hat{C}_2|} - \frac{1}{2}(x-\hat{\mu}_1)^\mathsf{T}\hat{C}_1^{-1}(x-\hat{\mu}_1) + \frac{1}{2}(x-\hat{\mu}_2)^\mathsf{T}\hat{C}_2^{-1}(x-\hat{\mu}_2) > 0$$

où nous avons pris le logarithme pour atteindre la deuxième ligne. En distribuant les formes quadratiques et en rassemblant les termes, on obtient finalement

$$\frac{1}{2}x^\mathsf{T}(\hat{C}_2^{-1} - \hat{C}_1^{-1})x + x^\mathsf{T}\left(\hat{C}_2^{-1}\hat{\mu}_2 - \hat{C}_1^{-1}\hat{\mu}_1\right) + \frac{1}{2}\left(\hat{\mu}_2^\mathsf{T}\hat{C}_2^{-1}\hat{\mu}_2 - \hat{\mu}_1^\mathsf{T}\hat{C}_1^{-1}\hat{\mu}_1 + 2\log\frac{\pi_1}{\pi_2} - \log\frac{|\hat{C}_1|}{|\hat{C}_2|}\right) > 0.$$

Au niveau de la frontière, on obtient donc une forme de type

$$g(x) \equiv x^\mathsf{T}Ax + x^\mathsf{T}b + c = 0.$$

qui est une forme quadratique : la région de décision n'est donc ici plus linéaire mais quadratique et on parle d'*analyse discriminante quadratique* (*quadratic discriminant analysis* ou *QDA* en anglais).

La Figure 3.2.2 illustre les performances de la méthode QDA en fonction du nombre d'échantillons n dans le cadre de la classification de $k = 2$ classes d'un mélange gaussien en $p = 2$ dimensions. Comme on le constate, déjà ici, le nombre de données d'apprentissage n doit être plusieurs ordres de grandeurs supérieur à p pour obtenir des performances satisfaisantes.

LISTING 3.3. Code Matlab de la Figure 3.2.2

```
% Tirage des données et détermination de la fonction de décision
n=1000;
mu1=[0;4];
mu2=[4;0];
rho1=.9;
C1=[1 rho1;rho1 1];
rho2=.1;
C2 = [1 rho2;rho2 1];

X1=mu1*ones(1,n/2)+C1^.5*randn(2,n/2);
X2=mu2*ones(1,n/2)+C2^.5*randn(2,n/2);

hat_mu1=mean(X1')';
hat_mu2=mean(X2')';
hat_C1=cov(X1');
hat_C2=cov(X2');

A=1/2*(inv(hat_C2)-inv(hat_C1));
b=inv(hat_C2)*hat_mu2-inv(hat_C1)*hat_mu1;
c=1/2*(hat_mu2'*inv(hat_C2)*hat_mu2-hat_mu1'*inv(hat_C1)*hat_mu1-log(
    det(hat_C1)/det(hat_C2)));

g=@(x) x'*A*x+b'*x+c;

% Création de la carte des classes
x1s=-10:1e-1:10;
x2s=-10:1e-1:10;
score=zeros(length(x1s),length(x2s));

ix1=1;
for x1=x1s
        ix2=1;
        for x2=x2s
                score(ix1,ix2)=g([x1;x2]);
                ix2=ix2+1;
        end
        ix1=ix1+1;
end

figure
imagesc(x1s,x2s,score>0);
hold on;
plot(X1(1,:),X1(2,:),'bx');
plot(X2(1,:),X2(2,:),'rx');
```

Un certain nombre de commentaires méritent d'être effectués à ce niveau. Un point fondamental de la méthode QDA a trait à son optimalité : sous l'hypothèse où les données x_i sont réellement issues d'un tel mélange gaussien (à savoir, si le modèle est parfaitement conforme aux données, ce qui est souvent discutable), lorsque $n_1, \dots, n_k \to \infty$, on obtient presque sûrement $\hat{\mu}_j \to \mu_j$ et $\hat{C}_j \to C_j$, de

sorte que la fonction de décision $g(x)$ *est optimale au sens du maximum de vraisemblance*. Ainsi, si la mesure de mérite de la classification d'une donnée x inconnue consiste à maximiser la probabilité a posteriori de y sachant x, dans la limite où $n_1, \dots, n_k \to \infty$, aucun algorithme d'apprentissage ne pourra faire mieux que QDA. On peut comprendre cette optimalité par le fait que le terme $\mathbb{P}(x, \{(x_i, y_i)\}|\tilde{\theta})$ de l'intégrale (3.2.1) se "concentre" quand $n_1, \dots, n_k \to \infty$ vers une mesure de Dirac en $\theta = \{(\mu_1, \dots, \mu_k, C_1, \dots, C_k\}$. Néanmoins, *cela ne signifie malheureusement pas* que QDA soit "presque optimal" pour des $n_1, \dots, n_k$ arbitraires fixes, même lorsque les x_i sont en effet réellement gaussiens. Bien d'autres méthodes, que l'on préfèrera de fait à QDA, atteignent des performances bien meilleures lorsque les n_j ne sont pas très grands ; et ce, à nouveau, même sur des données réellement gaussiennes. C'est le cas des SVMs, mais aussi de manière surprenante de LDA que nous allons évoquer par la suite.

Comme précisé plus haut, on obtient en Figure 3.2.2 une région de décision qui prend une forme quadratique et non pas linéaire comme ce sera le cas de la méthode LDA que nous allons étudier ensuite mais aussi et surtout des machines à vecteurs de support (linéaires). Ainsi au lieu de se contenter de séparer l'espace des données au moyen d'un hyperplan, on a accès à une séparation plus "flexible" par des paraboles et hyperboles. Pour parvenir au même effet, les machines à vecteurs de support devront s'équiper des représentations plus riches (via un noyau au moins quadratique $\phi(\cdot)$) des données. Néanmoins ce découpage quadratique de l'espace des données induit des artefacts gênants, directement observables dans la Figure 3.2.2. On remarque en effet que la zone "rouge" associée à la classe des échantillons centrés sur $(4, 0)$ se scinde ici en deux, et en particulier intègre la zone inférieure gauche dont les points sont plus proches des échantillons bleus que des échantillons rouges. Ceci n'a rien de surprenant et est simplement lié au fait que, sous l'hypothèse gaussienne bi-variée, les échantillons bleus ont une moindre variance dans la direction "bas-gauche" que les échantillons rouges, qu'on risque donc de trouver plus probablement à ce niveau. Mais du point de vue de l'apprentissage, ce résultat reste perturbant et vraisemblablement peu souhaitable : si des données "à la marge" (les dits *outliers*) sont découverts dans cette zone, on souhaite plus probablement les associer à la classe bleue qu'à la classe rouge.

Mais le point le plus délicat à gérer avec l'algorithme QDA est celui de sa forte instabilité numérique. En effet, on observe dans la Figure 3.2.2 des variations très fortes de la zone de décision prédite lorsque n n'est pas très grand par rapport à p. Une des raisons au cœur du problème est celui de l'*utilisation des estimateurs plug-in* des matrices de covariances C_1 et C_2 dans la formule de la fonction de décision $g(x)$. En effet, celles-ci apparaissent sous la forme d'inverses $\hat{C}_j^{-1}$ et de déterminants $\log|\hat{C}_j|$. Or, par définition, $\hat{C}_j \propto \sum_{i|y_i=1}(x_i - \hat{\mu}_j)(x_i - \hat{\mu}_j)^\intercal$ est une matrice de rang au plus $n_j = |\{i|y_i = 1\}|$. Son inverse n'est donc déjà pas défini lorsque $p > n_j$ et son déterminant nul, ce qui apparait ainsi lorsque la dimension des données p est grande ou qu'au moins une classe est très sous-représentée. Mais en fait, même si $p < n_j$, un argument simple de la *théorie des matrices aléatoires* permet de montrer que l'ensemble des valeurs propres de $\hat{C}_j$ peut être très étalé par rapport à celles de C_j. Les machines à vecteurs de support, qui n'utilisent ni inversions matricielles, ni calculs de déterminants, ne souffriront pas de cette limitation.

Remarque 9 (De la difficulté d'estimer les matrices de covariance). Supposons simplement que $C_j = I_p$ et donc que toutes les valeurs propres de C_j égalent 1. Alors la théorie des matrices matrices permet de démontrer que, dans la limite où $n_j, p \to \infty$ de telle sorte que $p/n_j \to c_j > 0$ constant, la plus petite valeur propre $\lambda_{\min}(\hat{C}_j)$ converge presque sûrement vers $(1-\sqrt{c_j})^2$ et la plus grande $\lambda_{\max}(\hat{C}_j)$ vers $(1+\sqrt{c_j})^2$.

Et donc, même en prenant $p = 10n_j$, on trouve $\lambda_{\min}(\hat{C}_j) \simeq .46$ et $\lambda_{\max}(\hat{C}_j) \simeq 1.73$, qui diffèrent fortement de 1 ! En fait, même pour $n_j = 100p$, on trouve $\lambda_{\min}(\hat{C}_j) \simeq .81$ et $\lambda_{\max}(\hat{C}_j) \simeq 1.21$, ce qui n'est pas une erreur négligeable par rapport à 1. La Figure 3.2.3 propose une représentation visuelle de ce phénomène. Ainsi, en pratique, d'une part on estime très mal C_j mais en plus l'inverse $\hat{C}_j^{-1}$ et le logarithme du déterminant $\log|\hat{C}_j|$ peuvent très vite créer d'importantes erreurs dans l'estimateur final de QDA. C'est une limite fondamentale de cette approche, et de beaucoup d'autres méthodes qui utilisent aussi des estimateurs *plug-in* de matrices de covariances.

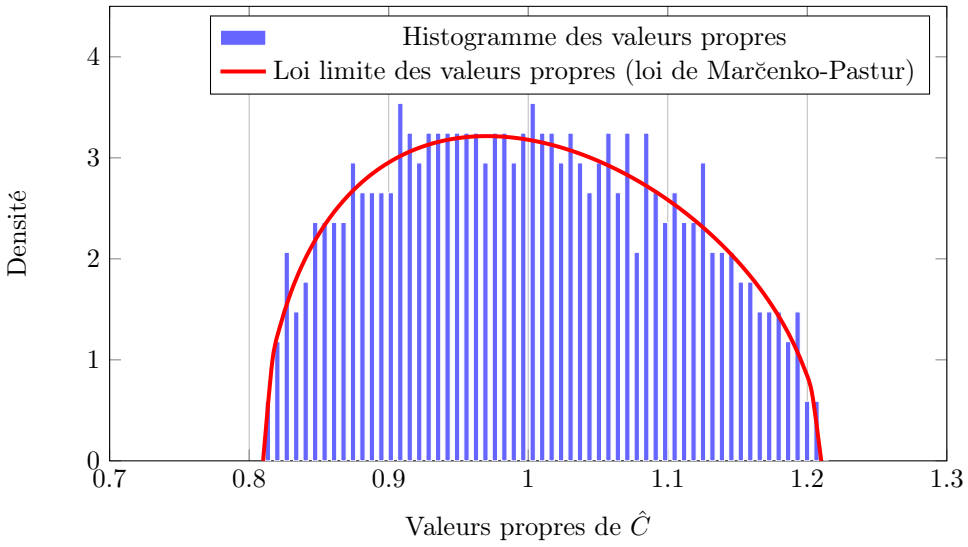

FIGURE 3.2.3. Histogramme de la distribution empirique des valeurs propres de la covariance empirique $\hat{C}$ issue de n données indépendantes $x_i \sim \mathcal{N}(0, I_p)$ comparé à la loi limite de Marčenko-Pastur. Ici pour $p = 500$ et $n = 50\,000$. On observe que les valeurs propres s'étalent ici de .81 à 1.21, au lieu d'être concentrées autour de 1.

Des solutions à cette instabilité ont été envisagées dans la littérature qui consistent à "régulariser" l'estimateur $\hat{C}_j$ en y ajoutant une matrice αI_p pour conserver l'inversibilité, mais ceci crée un biais et ne résout pas le problème de fond.

En fait, comme nous allons le voir, et de manière assez surprenante à première vue, la stratégie la plus "sure" reste encore de contourner le problème en supposant que les deux classes ont la même matrice de covariance $C_1 = C_2$, même lorsque cela est faux ! Ceci donne lieu, comme nous allons le voir tout de suite, à la méthode appelée *analyse discriminante linéaire* ou LDA, pour *linear discriminant analysis*, en anglais.

3.2.2.2. *LDA.* Ce qui fait conceptuellement souffrir la méthode QDA, nous l'avons vu, réside dans l'erreur systématique d'évaluation des matrices de covariance C_1 et C_2, tandis que les moyennes μ_1 et μ_2 sont bien mieux évaluées. Ceci se comprend en fait bien : on a accès, pour chaque $\mu_j \in \mathbb{R}^p$ et $C_j \in \mathbb{R}^{p\times p}$ à n_j échantillons $x_j \in \mathbb{R}^p$; en terme de "degrés de libertés", c'est donc np degrés de libertés (les $n \times p$ entrées de tous les vecteurs x_i) utilisés pour estimer, respectivement, les p entrées de μ_j et les p^2 entrées de C_j. Si $n_j \sim p$, on a donc en moyenne n_j valeurs

par scalaire à estimer dans μ_j mais seulement n_j/p valeurs par scalaire à estimer dans C_j.

L'idée de LDA est de supposer que les covariances $C_1, \ldots, C_k$ du mélange gaussien sont toutes identiques, égales à une covariance unique C, et d'estimer alors C au lieu des C_j. Ceci ne semble a priori pas résoudre le problème car on a toujours une covariance à estimer. Cependant, au contraire de QDA pour lequel la fonction de décision $g(x)$ dépend de manière quadratique de $\hat{C}_j^{-1} - \hat{C}_{j'}^{-1}$ et introduit un biais lié à $\log(|\hat{C}_j|/|\hat{C}_{j'}|)$ (qui ont toutes les chances de produire des résultats désastreux), dans le cas de LDA, les erreurs d'estimation de $\hat{C}$ n'affecteront que marginalement $g(x)$ dont la décision se fera surtout sur la base des $\hat{\mu}_j - \hat{\mu}_{j'}$.

Pour bien s'en rendre compte, reprenons donc les équations de la section précédente, mais maintenant avec le modèle de mélange

$$\mu_{xy} \equiv \sum_{i=1}^{k} \pi_i \mu_{;\theta_i} = \sum_{i=1}^{k} \pi_i \mathcal{N}(\mu_i, C)$$

pour une certaine matrice de covariance C commune à toutes les classes, et $\theta_i = \{(\mu_i, C)\}$. On estime toujours μ_i par la moyenne empirique $\hat{\mu}_i = 1/|\{y_j = i\}| \sum_{j|y_j=i} x_j$ mais cette fois-ci on va estimer la covariance commune via l'ensemble des données :

$$\hat{C} = \frac{1}{n(n-1)} \sum_{i=1}^{k} \sum_{j|y_j=i} (x_j - \hat{\mu}_i)(x_j - \hat{\mu}_i)^\mathsf{T}.$$

À supposer que l'hypothèse de la covariance commune est à peu près correcte, l'estimée de C est déjà meilleure que celle des C_i distincts comme on utilise ici l'ensemble des n données au lieu des données restreintes à chaque classe.

Tout comme pour QDA, on effectue alors un *plug-in* de cet estimateur dans la densité a posteriori de toute donnée x de classe y inconnue, pour obtenir

$$\mu_{x;\{\hat{\mu}_y,\hat{C}\}}\pi_y = \frac{\pi_y}{\sqrt{2\pi|\hat{C}|}} \exp\left(-\frac{1}{2}(x - \hat{\mu}_y)^\mathsf{T}\hat{C}^{-1}(x - \hat{\mu}_y)\right)$$

qu'il s'agira donc d'évaluer pour chaque valeur candidate de $y \in \{1, \ldots, k\}$. En comparant les hypothèses $y = 1$ et $y = 2$, on obtient cette fois

$$\begin{aligned}\mu_{x;\{\hat{\mu}_1,\hat{C}\}}\pi_1 - \mu_{x;\{\hat{\mu}_2,\hat{C}\}}\pi_2 = &\frac{\pi_1}{\sqrt{2\pi|\hat{C}|}} \exp\left(-\frac{1}{2}(x - \hat{\mu}_1)^\mathsf{T}\hat{C}^{-1}(x - \hat{\mu}_1)\right) \\ &- \frac{\pi_2}{\sqrt{2\pi|\hat{C}|}} \exp\left(-\frac{1}{2}(x - \hat{\mu}_2)^\mathsf{T}\hat{C}^{-1}(x - \hat{\mu}_2)\right)\end{aligned}$$

de sorte que l'hypothèse $y = 1$ est retenue sur l'hypothèse $y = 2$ si

$$\begin{aligned}&\pi_1 \exp\left(-\frac{1}{2}(x - \hat{\mu}_1)^\mathsf{T}\hat{C}_1^{-1}(x - \hat{\mu}_1)\right) > \pi_2 \exp\left(-\frac{1}{2}(x - \hat{\mu}_2)^\mathsf{T}\hat{C}_2^{-1}(x - \hat{\mu}_2)\right) \\ \Leftrightarrow &\log\frac{\pi_1}{\pi_2} - \frac{1}{2}(x - \hat{\mu}_1)^\mathsf{T}\hat{C}^{-1}(x - \hat{\mu}_1) + \frac{1}{2}(x - \hat{\mu}_2)^\mathsf{T}\hat{C}^{-1}(x - \hat{\mu}_2) > 0\end{aligned}$$

de sorte que les déterminants des covariances disparaissent de l'expression. Après simplification, on obtient cette fois

$$x^\mathsf{T}\hat{C}^{-1}(\hat{\mu}_2 - \hat{\mu}_1) + \frac{1}{2}\left(\hat{\mu}_2^\mathsf{T}\hat{C}^{-1}\hat{\mu}_2 - \hat{\mu}_1^\mathsf{T}\hat{C}^{-1}\hat{\mu}_1 + 2\log\frac{\pi_1}{\pi_2}\right) > 0.$$

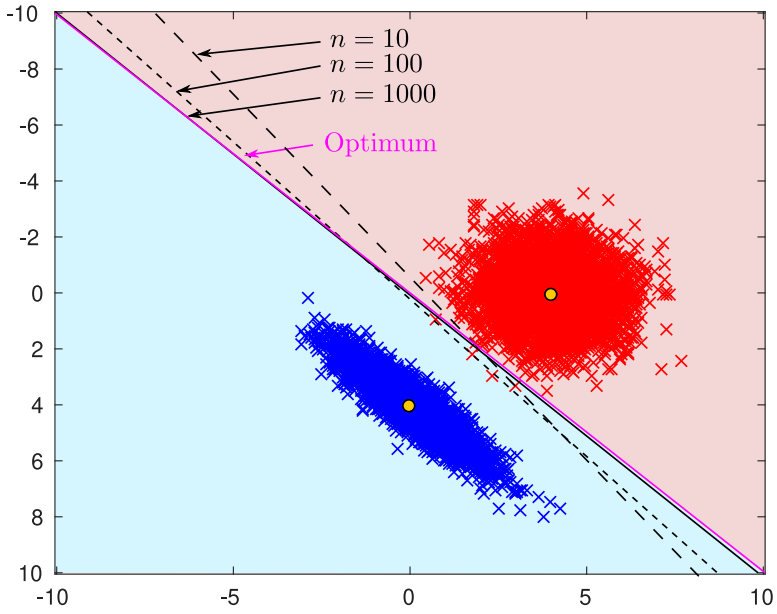

FIGURE 3.2.4. Performance de la méthode LDA pour un mélange de deux gaussiennes bi-variées de covariances C_1 et C_2 distinctes mais supposées égales par la méthode. La frontière de décision est ici un hyperplan séparateur dont l'approximation est assez correcte lorsque n est petit (ici 10^2). Néanmoins, on observe que l'hypothèse non vérifiée $C_1 = C_2$ induit un décalage clair de la frontière optimale obtenue par QDA dans la limite $n \to \infty$.

L'équation de la frontière de décision est donc maintenant *linéaire*, et non plus quadratique, et prend la forme

$$g(x) \equiv x^\mathsf{T} b + c = 0$$

à savoir l'équation d'un *hyperplan*, ce qui n'est pas sans rappeler la méthode LSSVM que nous avons évoquée dans l'introduction et sur laquelle nous reviendrons plus longuement par la suite.

La Figure 3.2.4 reproduit la même simulation que dans la Figure 3.2.2 mais sous l'hypothèse, ici erronée donc, de l'égalité $C_1 = C_2 = C$ de sorte à pouvoir utiliser la méthode LDA. Au contraire de QDA, on observe, comme cela était prévu, une division de l'espace des solutions par un hyperplan de vecteur normal $b = \hat{C}^{-1}(\hat{\mu}_2 - \hat{\mu}_1)$. On remarque par ailleurs que, dès $n = 100$, l'hyperplan optimal est déjà bien approché : en fait, il faut voir ici que l'estimation du vecteur b est surtout guidée par l'estimation de $\hat{\mu}_2 - \hat{\mu}_1$, qu'on sait être très bonne en générale, ainsi que par une "rotation" par $\hat{C}^{-1}$ qui affecte marginalement la direction $\hat{\mu}_2 - \hat{\mu}_1$, si tant est que l'estimation de C^{-1} par $\hat{C}^{-1}$ n'est pas trop altérée (ce qui est en général le cas si C est bien conditionnée).

Cependant, il est important de relever, et cela se voit assez bien visuellement en Figure 3.2.4, que l'erreur d'hypothèse $C_1 = C_2 = C$ peut induire une position sous-optimale de l'hyperplan "objectif" (celui obtenu pour $n \to \infty$). En somme LDA permet de modifier le *compromis biais-variance* de QDA en réduisant la *variance* (induite par les erreurs d'estimation des $\hat{C}_j^{-1}$) mais au prix, lorsque les C_j sont trop différents, d'une augmentant du *biais* (induit par une erreur de modèle).

LISTING 3.4. Code Matlab de la Figure 3.2.4

```
% Tirage des données et détermination de la fonction de décision
n=1000;
mu1=[0;4];
mu2=[4;0];
rho1=.9;
C1=[1 rho1;rho1 1];
rho2=.1;
C2 = [1 rho2;rho2 1];

X1=mu1*ones(1,n/2)+C1^.5*randn(2,n/2);
X2=mu2*ones(1,n/2)+C2^.5*randn(2,n/2);

hat_mu1=mean(X1')';
hat_mu2=mean(X2')';
hat_C=1/2*(cov(X1')+cov(X2'));

b=inv(hat_C)*(hat_mu2-hat_mu1);
c=1/2*(hat_mu2'*inv(hat_C)*hat_mu2-hat_mu1'*inv(hat_C)*hat_mu1;

g=@(x) b'*x+c;

% Création de la carte des classes
x1s=-10:1e-1:10;
x2s=-10:1e-1:10;
score=zeros(length(x1s),length(x2s));

ix1=1;
for x1=x1s
        ix2=1;
        for x2=x2s
                score(ix1,ix2)=g([x1;x2]);
                ix2=ix2+1;
        end
        ix1=ix1+1;
end

figure
imagesc(x1s,x2s,score>0);
hold on;
plot(X1(1,:),X1(2,:),'bx');
plot(X2(1,:),X2(2,:),'rx');
```

Conclusion de la section. Après avoir introduit les approches heuristiques populaires que sont l'algorithme des plus proches voisins et les arbres de décisions, dont nous avons établi les difficultés d'analyse et les limitations (notamment lorsqu'il s'agit de traiter de données hétérogènes pour kNN ou en grandes dimensions pour kNN et les arbres de décisions), nous nous sommes tournés dans cette section vers des outils probabilistes (estimation bayésienne et analyses discriminantes) reposant sur un socle théorique plus solide mais fondés sur des hypothèses généralement très irréalistes.

Comparer la valeur des deux types d'approches est de fait compliqué et dépend largement des données et des problèmes visés. On peut néanmoins établir des petites règles simples :

- **données hétérogènes, binaires, nominales, etc.** : lorsque les vecteurs de données $x_i \in \mathcal{X}$ (ou leurs représentations) consistent en des collections d'attributs de natures variées de l'item i (en sciences sociales, ces attributs peuvent être des sexe, âge, poids, réponses à des enquêtes, etc.), il est bien difficile de modéliser l'espace $\mathcal{X}$ et encore plus d'y induire des opérateurs et métriques. On aura tendance dans ce cas à vouloir traiter les p entrées des x_i de manière isolée : les arbres de décisions sont évidemment préférés dans ce cas de figure, chaque niveau de l'arbre résultant conservant d'ailleurs un sens physique utile à l'expérimentateur ;
- **données complexes, difficilement modélisables ou éloignées du monde physique** : proposer un modèle probabiliste simple des données (mélange gaussien, mélange de lois discrètes, etc.) suppose souvent que l'expérimentateur a une vague idée de la nature physique des données. C'est notamment le cas dans un vaste champ des sciences physiques (mécanique, électromagnétique, sciences de l'univers, télécommunications, etc.). En dehors de ce cadre, les modèles portent peu de sens (qu'est-ce que modéliserait un bruit gaussien dans un problème de regroupement automatique de pixels sur une image arbitraire en deux dimensions ?). On se tournera ici plutôt vers des méthodes heuristiques simples et robustes, qui évitent les biais criants des modèles. L'algorithme kNN est l'une de ces solutions dont, si on ne sait analyser ses performances théoriquement, au moins on maitrise facilement le comportement attendu. N'oublions cependant pas que ces algorithmes peuvent souffrir d'un phénomène de *malédiction de la dimension* qui les rend fortement instables lorsque p est grand (déjà lorsque $p \sim 10$).
- **données "naturelles", physiquement modélisables** : on doit comprendre ici l'ensemble des signaux, des séries temporelles, et même de certaines représentations bien choisies d'images, qu'il est crédible de modéliser par des vecteurs aléatoires. Le spectre est assez large car, même une image de chiffre manuscrit (ceux de la populaire base MNIST) peut être vue comme une déformation bruitée d'une image de référence ; il n'est en particulier pas rare que l'hypothèse de mélange gaussien soit très adaptée, non pas forcément pour "bien modéliser" les données, mais au moins pour comprendre et anticiper le comportement des algorithmes qui traitent ces données. Pour cette large famille de données, les approches probabilistes ou les approches statistiques relevant de l'optimisation d'un critère bien identifié seront privilégiées.

Dans le cadre de la suite du cours, dans un objectif tourné vers les applications modernes de l'apprentissage automatique (aux grandes bases de données et aux données elles-mêmes de grandes dimensions), nous allons désormais nous concentrer sur cette troisième famille de méthodes. Nous évoquerons en premier lieu et comme point de référence les populaires machines à vecteurs de support (SVMs), dont nous établirons qu'elles ont en fait fortement liées à des méthodes de régressions simples, et d'une certaine manière à la méthode LDA. Par cette connexion, nous discuterons en particulier de la méthode LSSVM, dérivée des SVMs, qui nous permettra ensuite de faire le lien avec les méthodes de projections aléatoires non-linéaires qui nous ferons replonger dans le contexte des réseaux de neurones. Le pont entre SVMs et réseaux de neurones s'effectuera plus précisément par le biais des méthodes à noyaux, les *réseaux de neurones aléatoires* implémentant indirectement une famille particulière de noyaux. Nous conclurons le cours spécifiquement par une étude des

réseaux de neurones récurrents dits "echo-state" très utiles pour la prédiction simple de séries temporelles complexes, telles que celles rencontrées en statistiques pour la finance, et qui elles-aussi se résument à une simple méthode de régression linéaire.

3.3. Exercices

Exercice 7 (Implémentation des méthodes LDA/QDA). Les codes des Listings 3.3–3.4 proposent une implémentation des méthodes LDA et QDA en petite dimension ($p = 2$). L'exercice consiste à réimplémenter le code (en Matlab ou en Python) pour des tailles arbitraires de données p. On pourra notamment simuler un modèle de données issues d'un mélange à deux classes $\mathcal{N}(\mu_1, C_1)$ et $\mathcal{N}(\mu_2, C_2)$ où μ_1 et μ_2 pourront être des vecteurs tirés aléatoirement sur la sphère unitaire de $\mathbb{R}^p$, multipliés par un scalaire d'amplitude (déterministe ou aléatoire), et où les C_i seront pris :

- soit Toeplitz de la forme $[C_i]_{ab} = \alpha^{|a-b|}$ pour un certain $\alpha \in [0, 1]$: dans ce cas, pour $\alpha \to 0$, C_i tend vers la matrice identité, tandis que pour $\alpha \to 1$, C_i tend vers une matrice de rang 1 alignée au vecteur 1_p (toute la variation aléatoire des données est donc dans cette même direction) ;
- soit *Wishart* aléatoire, à savoir $C_i = Z_i Z_i^\mathsf{T}/q$, pour $Z_i \in \mathbb{R}^{p\times q}$ avec $[Z_i]_{ab} \sim \mathcal{N}(0, 1)$ et q à choisir : ici, pour $q \to \infty$, C_i tend vers la matrice identité alors que pour $q \ll p$ petit, C_i est de rang faible aligné sur des directions aléatoires.

Simuler ces différentes configurations pour n données d'apprentissage en attachant de l'importance : (i) à l'impact du rapport p/n sur la qualité de QDA, (ii) à l'impact de la proximité des matrices C_1 et C_2, (iii) à l'impact de la "structure" faible ou forte (à savoir proche ou éloigné de la matrice identité) des matrices de covariances, (iv) à l'impact relatif des distances entre moyennes et covariances. Conclure quant aux cas d'utilisation favorables de LDA ou de QDA, en validant notamment les remarques élaborées dans le cours.

Exercice 8 (Algorithme kNN pour d'autres métriques). Dans beaucoup de problème d'apprentissage, les classes des éléments peuvent être statistiquement définies via leur structure de covariance. C'est le cas de certaines séries temporelles (par exemple en finance). L'idée de l'exercice est de simuler le scénario d'un mélange deux classes de données $x_1, \dots, x_n \in \mathbb{R}^p$ avec $x_i \sim \mathcal{N}(0, C_1)$ ou $x_i \sim \mathcal{N}(0, C_2)$ selon la classe. Pour simuler des corrélations de séries stationnaires, on prendra $C_\ell = \{\alpha_\ell^{|i-j|}\}_{i,j=1}^p$ pour $\ell \in \{1, 2\}$ et un choix libre des $\alpha_\ell \in [0, 1)$: ce paramètre α_ℓ traduit la corrélation entre deux éléments consécutifs de la série temporelle et modélise ainsi un processus dit autorégressif du premier ordre.

Afin de classer les séries temporelles, une idée peut consister à utiliser un algorithme des plus proches voisins dont la métrique est la distance entre matrices de covariances des échantillons. On va donc supposer ici avoir accès, pour chaque échantillon x_i à m observations indépendantes (par exemple en m périodes distinctes) $x_i^{(1)}, \dots, x_i^{(m)}$ de x_i. Ces m observations vont fournir un estimateur empirique $\frac{1}{m}\sum_{j=1}^m x_i^{(j)} x_i^{(j)\mathsf{T}}$ de la matrice de covariance de x_i. Pour comparer les différentes covariances empiriques, on utilisera alors l'algorithme des k plus proches voisins pour une certaine distance entre matrices de covariances, telles que :

- la distance de Frobenius $\|A - B\|_{\mathrm{Fro}} = \sqrt{\mathrm{tr}((A - B)^2)}$ qui est l'extension matricielle de la distance euclidienne pour les vecteurs ;
- la distance de Fisher $\|A - B\|_{\mathrm{F}} = \sqrt{\sum_{i=1}^p \log^2 \lambda_i(A^{-1}B)}$ qui est la distance de la géodésique liant A à B dans l'espace des matrices définies positives ;

— la divergence de Kullbach-Liebler $d_{\mathrm{KL}}(A, B) = \mathrm{tr}(A^{-1}B) - p + \log\det(A^{-1}B)$ qui évalue la distance entre deux distribution gaussienne centrées au sens de l'entropie ;
— etc., etc.

Pour toutes ces distances et métriques, implémenter un algorithme des k plus proches voisins que l'on appliquera au jeu de données synthétique ci-dessus, en jouant sur les différents paramètres n, p, k notamment. Relever les performances dans chaque situation et commenter les résultats obtenus.

Chapitre 4

Méthodes de minimisation du risque empirique (SVM, LSSVM, régression logistique, etc.)

Nous avons déjà proposé une vue globale des machines à vecteurs de support (SVM) dans l'introduction du cours (en Section 1.3.2) et allons consacrer ce chapitre à aller plus dans le détail, mais avec un angle de vue différent qui nous permettra de mieux connecter les SVMs à d'autres algorithmes de l'apprentissage automatique.

Rappelons que dans la Section 1.3.2, nous avons discuté de l'approche "historique" géométrique des SVMs "durs" (*hard-margin SVM*) qui consistent à trouver trois hyperplans affines $\mathcal{H}_1$, $\mathcal{H}_{-1}$ et $\mathcal{H}_0$ parallèles et définis par $\mathcal{H}_j : \{z \in \mathbb{R}^p | (w^\star)^\mathsf{T} z + j = 0\}$ (et donc simultanément définis par le vecteur normal $w^\star$), de telle sorte que :

- l'ensemble des vecteurs x_i de la base d'entrainement $\{(x_i, y_i)\}_{i=1}^n$, où $y_i \in \mathcal{Y} = \{-1, +1\}$ est binaire, se trouvent à l'extérieur des hyperplans $\mathcal{H}_1$ et $\mathcal{H}_{-1}$ (aucun point ne se trouve entre $\mathcal{H}_1$ et $\mathcal{H}_{-1}$)
- les vecteurs x_i de même classe y se trouvent tous dans le même demi-espace induit par l'hyperplan $\mathcal{H}_y$
- parmi tous les triplets $\{(\mathcal{H}_1, \mathcal{H}_0, \mathcal{H}_{-1})\}$ solutions (sauf cas très particulier, en général il y en a une infinité ou aucun), on retiendra celle qui induit la plus grande distance entre $\mathcal{H}_1$ et $\mathcal{H}_{-1}$. Pour cela, on cherchera, parmi tous les $w^\star$ valides, celui dont la norme $\|w^\star\|$ est minimale. Rappelons en effet que la distance d'un point $z_1 \in \mathcal{H}_1$ à son projeté $z_{-1} = z_1 + 2w^\star/\|w^\star\|^2 \in \mathcal{H}_{-1}$ vaut $\|z_1 - z_{-1}\| = 1/\|w^\star\|$ qui est donc maximal lorsque $\|w^\star\|$ est minimale.

Cette approche géométrique est illustrée dans la Figure 4.0.1.

4.1. Minimisation du risque empirique régularisé

Nous allons cependant opter ici pour une vision plus large, plus générale, et qui permet de voir les SVMs (durs) comme un algorithme particulier dans la famille plus riche des *algorithmes de minimisation du risque empirique*. Rappelons en effet que l'objectif d'un algorithme d'apprentissage est de générer une fonction $g : \mathcal{X} \to \mathcal{Y}$ ayant pour objectif de minimiser une certaine erreur de généralisation $E_{\text{gen}}(g) = \mu_{xy}(\ell(g(x), y))$ où μ_{xy} est la mesure inconnue qui produit les paires $(x, y) \in \mathcal{X} \times Y$ et $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}^+$ est une distance ou mesure de *risque* (*risk* ou *loss* en anglais) entre y et son estimation $g(x)$. Pour parvenir à estimer E_{gen}, étant donnée une base de données $(x_1, y_1), \ldots, (x_n, y_n)$ d'apprentissage connue, on passera par exemple par de la validation croisée en découpant l'ensemble des données en une base d'entrainement $(x_1, y_1), \ldots, (x_{\text{train}}, y_{\text{train}})$ et une base de test $(x_{\text{train}+1}, y_{\text{train}+1}), \ldots, (x_n, y_n)$ et en approximant $E_{\text{gen}}(g)$ par $E_{\text{test}}(g) = \frac{1}{n_{\text{test}}} \sum_{i=n_{\text{train}}+1}^{n} \ell(g(x_i), y_i)$. Quant à la base d'entrainement $(x_1, y_1), \ldots, (x_{\text{train}}, y_{\text{train}})$, elle servira à construire g, et cela en minimisant le *risque empirique* :

$$E_{\text{train}}(g) = \frac{1}{n_{\text{train}}} \sum_{i=1}^{n_{\text{train}}} \ell(g(x_i), y_i).$$

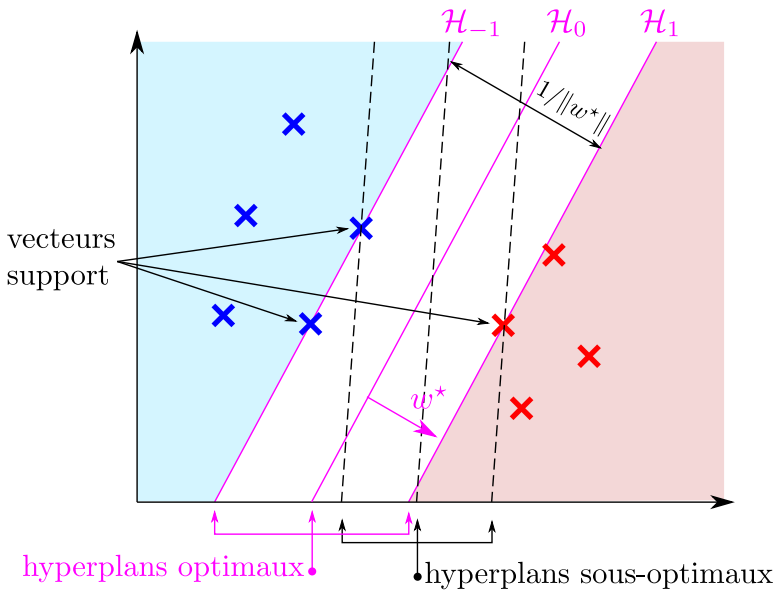

FIGURE 4.0.1. Représentation schématique d'un SVM en dimension $p = 2$: des vecteurs support sous-tendent les hyperplans optimaux de distance $1/\|w^\star\|$ minimale.

Nous l'avons déjà maintes fois évoqué, minimiser $E_{\text{train}}(g)$ seul comporte un fort risque : celui du surapprentissage qui est dû au fait que g, par construction, est une fonction de la base $(x_1, y_1), \ldots, (x_{\text{train}}, y_{\text{train}})$, et que l'évaluation de $E_{\text{train}}(g)$, à cause de la dépendance induite par les (x_i, y_i), peut être un piètre estimateur de $E_{\text{gen}}(g)$.

Cependant, passer par un découpage des données d'apprentissage en bases d'entrainement et de test est un sacrifice que l'on n'est généralement peu enclins à payer. Une manière de passer outre ce découpage, ou tout du moins, de réduire autant que possible la taille $n_{\text{test}} = n - n_{\text{train}}$ de la base de test, tout en limitant le problème du surapprentissage, consiste à *relaxer* le coût d'entrainement $E_{\text{train}}(g)$ en le *pénalisant* ou en le *régularisant* par un terme qui évite la génération de fonctions g trop "exotiques" (repensons à l'approximation polynomiale de la Figure 4 pour laquelle on souhaiterait ici pénaliser les coefficients des ordres les plus élevés du polynôme interpolateur). Ce qu'on demandera typiquement est de minimiser pour $g \in \mathcal{G}$ une fonction de la forme :

$$E(g) = \frac{1}{n} \sum_{i=1}^{n} \ell(g(x_i), y_i) + \gamma h(g)$$

où $h : \mathcal{G} \to \mathbb{R}^+$ impose une contrainte sur les fonctions g acceptables et $\gamma \geq 0$ est un hyperparamètre de contrôle. En particulier, lorsque $\gamma = 0$, on retrouve $E(g) = E_{\text{train}}(g)$ qui risque d'être sensible au surapprentissage, tandis que pour $\gamma \to \infty$, on choisira une fonction g qui minimise la contrainte $h(g)$ mais qui devient alors indépendante des données.

Nous allons nous concentrer dans cette partie sur une famille très simple mais pourtant déjà très riche, comme nous le verrons, de fonctions $g \in \mathcal{G}$ de *régression affine*, à savoir :

$$\mathcal{G} = \left\{ g : \mathcal{X} \subset \mathbb{R}^p \to \mathbb{R},\ g(x) = w^\mathsf{T} x + b,\ (w, b) \in \mathbb{R}^p \times \mathbb{R} \right\}.$$

On abuse ici des notations en autorisant l'image de tels g à être réelle et non réduite à $\mathcal{Y} = \{-1, +1\}$ mais évidemment la décision de classification finale y d'une donnée $x \in \mathcal{X}$ sera donnée par le signe de $g(x)$. Avec cette remarque, il apparait notamment que pour un couple de données typique (x, y), on souhaitera que $yg(x)$ soit positif (si $g(x) > 0$ alors $y > 0$, et si $g(x) < 0$ alors $y < 0$). Ainsi, dans ce cas particulier, on peut écrire le coût $E(g)$ introduit formellement auparavant comme :

$$E(g) = \frac{1}{n} \sum_{i=1}^{n} \rho(y_i(w^\mathsf{T} x_i + b)) + \gamma h(g)$$

où g est paramétrée par le couple $\{w, b\}$, et on a écrit ici $\ell(g(x), y) = \rho(y(w^\mathsf{T} x + b))$ pour une certaine fonction ρ que l'on choisira telle que :

- $\rho(t)$ est (fortement) décroissante sur $(-\infty, 0)$, de sorte à (fortement) pénaliser les valeurs négatives de t qui, pour $t = y(w^\mathsf{T} x + b) < 0$ correspondent à une erreur de classification du couple (x, y) ;
- $\rho(t)$ est soit minimale en $t = 1$, de sorte à forcer les couples (x, y) à vérifier $g(x) \simeq y$, ou alternatiement $\rho(t)$ est décroissante sur $(0, \infty)$ (et donc finalement, d'après le point précédent, sur $\mathbb{R}$ tout entier) de sorte à assurer que $yg(x)$ soit aussi éloigné que possible de valeurs négatives.

Parmi les fonctions ρ communément choisies, on trouve :

- $\rho(t) = \log(1 + e^{-t})$ qui va donner lieu à l'algorithme de *régression logistique*, que nous étudierons par la suite ;
- $\rho(t) = \frac{1}{2}(t-1)^2$ que l'on rencontrera dans les méthodes de *régression linéaire* ou LSSVM, également étudiés dans ce chapitre ;
- $\rho(t) = \imath(\{t \geq 1\})$, où $\imath(A) = 0$ si l'événement A est vrai et $\imath(A) = +\infty$ sinon ; c'est précisément le choix pris par l'algorithme des *machines à vecteurs de support*, au centre de nos préoccupations dans ce chapitre ;
- $\rho(t) = \max\{0, 1 - t\}$ qui est le choix pris pour la version dites *souple* des machines à vecteurs de support ;
- d'autres coûts tels que $\rho(t) = e^{-t}$ sont également utilisés dans certaines méthodes dites de *boosting*.

Pour se faire une idée, Figure 4.1.1 donne une représentation de ces fonctions.

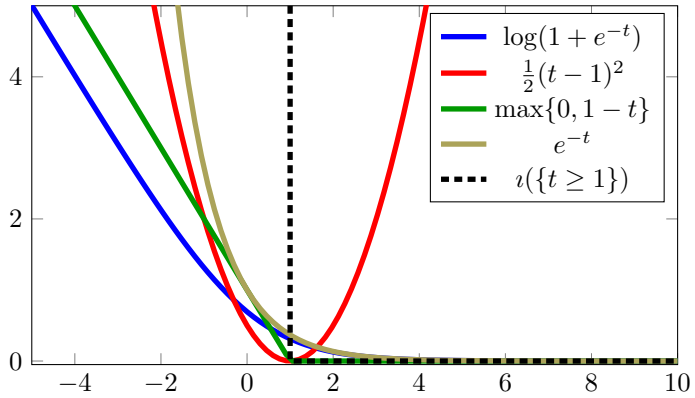

FIGURE 4.1.1. Fonctions de coût $\rho(t)$ communément employées dans la littérature.

Quant à la fonction $h(g)$ qui est donc ici une fonction des paramètres (w, b) qui paramétrisent h, on prendra bien souvent :

- un coût L^q avec $q > 1$ sur le vecteur de régression, à savoir $h(g) = (\sum_{i=1}^{p} w_i^q)^{1/q}$. Sous cette hypothèse, pour des q assez grand, on cherche surtout à éviter

que w n'ait des entrées trop grandes qui rendraient le régresseur très instable et sujet au surapprentissage (pour $q \to \infty$, on cherchera notamment à borner l'amplitude des plus grandes entrées de w) ;
— un coût L^q avec $q \leq 1$, qui au contraire tendra à répartir l'énergie du vecteur de régression w dans un nombre limité de ses entrées. L'idée est ici de chercher à rendre w assez "creux" de sorte que peu de variables des données x soient "explicatives" du phénomène $x \mapsto y$ que l'on cherche à caractériser. En particulier, dans la limite $q \to 0$, on cherchera à prendre w avec le moins d'entrées non nulles possibles, ce qu'on appelle aussi le contrainte de *parcimonie*.

Le choix de $h(g)$ dépend donc du problème considéré. Généralement cependant, pour des raisons numériques ou analytiques, on se contentera de deux choix standard :
— un coût quadratique $h(g) = \frac{1}{2}\|w\|^2$ (on ne prendra en général pas la racine et on divise par 2 par convention de sorte que $dh(g)/dw = w$) qui maitrise l'explosion des entrées de w : avec ce choix, on le verra, certains algorithmes, tels que le LSSVM deviennent analytiquement tractable ;
— un coût L_1, à savoir $h(g) = \sum_{i=1}^{p} |w_i|$, qui a été démontré récemment, dans le domaine en vogue de l'acquisition comprimée (*commpressive sensing* en anglais), être un très bon *ersatz* convexe du coût (non convexe) L_0 (voir les cours d'optimisation convexe moderne).

Pour notre part, nous nous contenterons dans ce cours de travailler avec la régularisation $h(g) = \|w\|^2$.

4.2. Machine à vecteurs de support (SVM)

4.2.1. Minimisation du risque et lien avec la vision géométrique. Comme nous venons de l'évoquer, dans la vision qui est la notre ici, les SVMs sont l'une des nombreuses formes d'algorithmes de minimisation du risque empirique régularisé, que l'on écrit donc ici :

$$E(g) = \frac{1}{n}\sum_{i=1}^{n} \imath(\{y_i(w^\mathsf{T} x_i + b) \geq 1\}) + \frac{\gamma}{2}\|w\|^2.$$

Pour bien comprendre de quoi il s'agit, remarquons que $E(g) = +\infty$ si pour au moins un couple (x_i, y_i), on a $y_i(w^\mathsf{T} x_i + b) < 1$ donc si ce couple (x_i, y_i) est du "mauvais côté" de l'hyperplan $\mathcal{H}_1 = \{z,\ w^\mathsf{T} z + b = 1\}$ (quand $y = 1$) ou de l'hyperplan $\mathcal{H}_{-1} = \{z,\ w^\mathsf{T} z + b = -1\}$ (quand $y = -1$). De tels choix de (w, b) sont donc invalides. Parmi ceux qui restent, si la régularisation $\frac{\gamma}{2}\|w\|^2$ n'était pas ajoutée, tous les coûts $E(g)$ seraient nuls et équivalents. Grâce à la régularisation cependant, pour g tel que (w, b) est valide ($E(g) < +\infty$), on obtient

$$E(g) = \frac{\gamma}{2}\|w\|^2$$

et il s'agit donc de retenir comme solutions les vecteurs w ayant la norme la plus petite dans cette famille. Ceci coïncide donc avec la vision géométrique de la *marge maximale* entre les hyperplans $\mathcal{H}_1$ et $\mathcal{H}_{-1}$. En régularisant le problème, on sélectionne donc en somme la fonction g la plus "souple" qui satisfasse les contraintes.

Le cas des SVMs est assez particulier en cela que la fonction $\rho(t)$ est constante par variation d'échelle : $\alpha\rho(t) = \rho(t)$ pour tout $\alpha > 0$ de sorte que le paramètre γ de la minimisation de $E(g)$ n'a aucun impact et est en cela inopérante. On peut donc plus simplement réécrire le problème sous la forme

$$\mathrm{argmin}_{(w,b)}\, E(g) = \mathrm{argmin}_{(w,b)}\, \frac{1}{2}\|w\|^2 + \frac{1}{n}\sum_{i=1}^{n} \imath(\{y_i(w^\mathsf{T} x_i + b) \geq 1\})$$

ou si on préfère, et c'est parfaitement équivalent

$$\operatorname{argmin}_{(w,b)} E(g) = \operatorname{argmin}_{(w,b)} \frac{1}{2}\|w\|^2, \quad \text{tel que} \quad \forall i \in \{1,\dots,n\},\ y_i(w^\mathsf{T} x_i + b) \geq 1$$

de sorte à retrouver la forme introduite dans la Section 1.3.2.

4.2.2. Résolution des SVMs. Ayant énoncé le problème des SVMs à la fois sous une forme géométrique en Section 1.3.2 et sous la forme d'une *régression affine* (non linéaire) équivalente à l'instant, on comprend désormais mieux la pertinence "intuitive" ou heuristique (vision géométrique) mais aussi plus formelle (dans sa forme de minimisation d'un risque empirique normalisé) de cet algorithme.

Cherchons maintenant à la résoudre, et à clarifier sa dénomination de *machine à vecteurs de support*, qui a des propriétés pratiques extrêmement intéressantes. Nous l'avons vu dans la Section 1.3.2, la formulation du problème SVM est celui de la minimisation d'une fonction fortement convexe en (w,b) ($\frac{1}{2}\|w\|^2$) sous des contraintes d'inégalités linéaires en (w,b) : il s'agit donc d'un problème d'optimisation convexe ayant *au plus une solution*. S'il n'est pas *réalisable* (à savoir s'il n'existe aucun couple (w,b) satisfaisant toutes les contraintes), il n'a pas de solution. Sinon, si au moins un tel couple (w,b) existe, cela permet de définir $(w^\star, b^\star)$ comme "la solution" du problème.

Supposons pour le moment le problème réalisable. En tant que problème d'optimisation quadratique sous contraintes linéaires, on sait alors qu'il est possible de le résoudre directement par des méthodes de *programmation (quadratique) dynamiques* (en anglais *quadratic dynamic programming* ou simplement *quadratic programming*). Ces méthodes sont de diverses nature :

— historiquement, la plus connue des méthodes est celle des *points intérieurs*, aussi appelée *méthode de la barrière*. Il s'agit pour ces approches d'effectuer une simple descente de gradient sur la fonction objectif (ici $\frac{1}{2}\|w\|^2$ dont le gradient vaut simplement w) tout en évitant de rester "bloqué" dans les contraintes (une fois qu'un point courant w de la descente atteint le cas d'égalité d'une des contraintes d'inégalités, la progression de la descente cesse). La méthode consiste à créer une pénalité, généralement logarithmique, qui serait ici de la forme $\eta \sum_i \log(y_i(w^\mathsf{T} x_i + b) - 1)$ pour un certain $\eta < 0$, qui devient arbitrairement élevée lorsqu'on s'approche du cas critique où $y_i(w^\mathsf{T} x_i + b) = 1$ pour un certain indice i. On joue alors sur η en le réduisant séquentiellement pour minimiser l'impact de ce coût artificiel. La méthode est illustrée en Figure 4.2.1. Cet outil est cependant assez difficile à manipuler, notamment parce qu'il faut intelligemment régler parallèlement le pas de la descente de gradient et la valeur de η, et ce, tout en contrôlant un critère d'arrêt de la descente qu'on ne veut pas être trop rapide.

— plus récemment, une élégante approche, celle des *méthodes proximales*, a permis de résoudre des problèmes d'optimisation convexes de la forme

$$\min_w f_1(w) + f_2(w)$$

où f_1 et f_2 sont deux fonctions convexes dont on sait déterminer *l'opérateur proximal*. Il s'avère que c'est le cas des fonctions $\|w\|^2$ et $\imath(A(w))$ pour $A(w)$ des contraintes linéaires en w. Des algorithmes, tels que le *forward-backward splitting* ou *l'algorithme de Douglas-Rachford*, permettent alors par le biais de la très élémentaire *méthode du point fixe* (on itère une équation de la forme $w^{(t+1)} = F(w^{(t)})$ jusqu'à convergence) de trouver $w^\star$; voir les cours modernes d'optimisation pour l'introduction à ces notions. La Figure 4.2.2 décrit brièvement l'idée de la méthode proximale.

Ces méthodes de programmation dynamique ne sont cependant pas à reproduire manuellement en général et sont bien implémentées dans les bibliothèques de Matlab, Python, Julia, etc. Il n'est pas non plus dans notre intérêt présent de creuser ces approches, mais juste de savoir qu'elles existent et savent résoudre le problème des SVMs. Néanmoins, avant de conclure sur ces outils, il est important de savoir que ces derniers deviennent coûteux en temps de calcul et généralement bien moins stables lorsque le nombre n de contraintes augmente : c'est un point assez gênant dans notre contexte, notamment lorsque nous devons utiliser un nombre important de données d'entrainement.

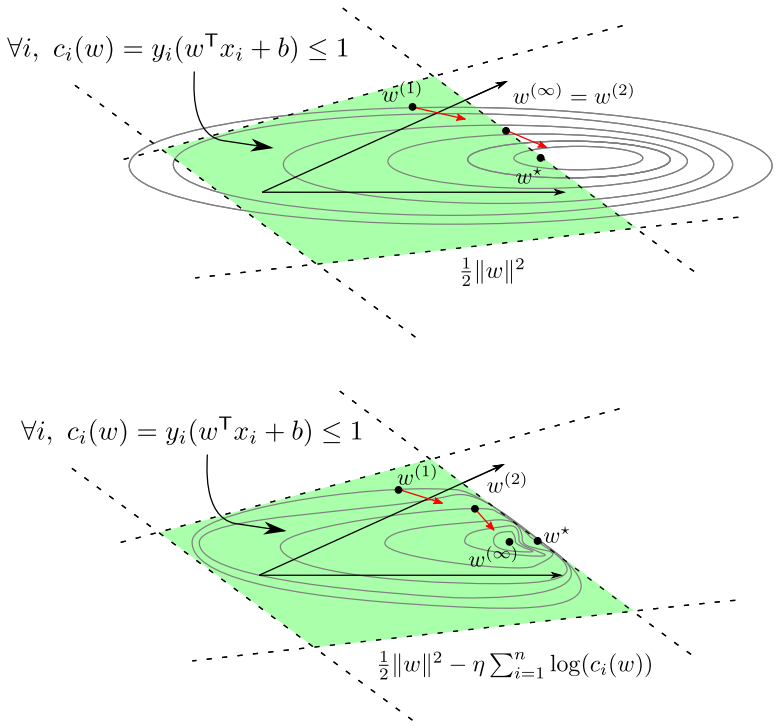

FIGURE 4.2.1. Illustration de la méthode de la barrière pour résoudre des problèmes d'optimisation convexes sous contraintes d'inégalités. En haut, une descente de gradient classique qui se heurte en deux pas à la contrainte et ne progresse plus. En bas, modification de la fonction objectif en prenant en ajoutant des "barrières" : la fonction s'en trouve déformée, la solution approximative, mais la descente de gradient ne s'arrête pas trop tôt.

4.2.3. Représentation duale. Mais revenons un moment sur le problème d'optimisation lui-même. En tant que problème convexe, il admet un problème *dual* qu'on obtient via le lagrangien

$$\mathcal{L}(\alpha) = \frac{1}{2}\|w\|^2 - \sum_{i=1}^{n} \alpha_i (y_i(w^\mathsf{T} x_i + b) - 1)$$

où $\alpha_1, \ldots, \alpha_n \geq 0$ sont les *multiplicateurs de Lagrange* (n'oublions pas : les α_i sont signés, positifs ou nuls, puisqu'on traite des contraintes *d'inégalités* et non d'égalité). On sait alors, c'est le principe de dualité forte dans le cas de problèmes

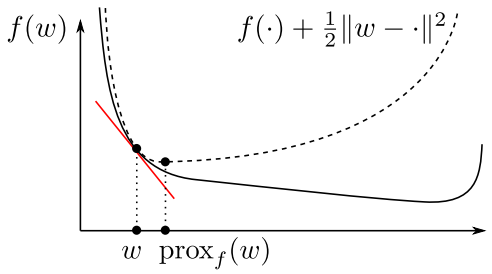

FIGURE 4.2.2. Définition de l'opérateur proximal $\mathrm{prox}_f(w) = \min_z f(z) + \frac{1}{2}\|z - w\|^2$ d'une fonction $f(w)$ convexe : même si f n'est pas différentiable (son gradient n'est pas défini) comme par exemple pour la fonction $\imath(A(w))$, son proximal l'est toujours et fournit une bonne approximation locale par régularisation. Les algorithmes proximaux exploitent cette propriété, à condition que prox_f ait une forme analytique connue (ce qui n'est malheureusement pas toujours le cas).

convexes, que

$$\sup_{\alpha \geq 0} \inf_{(w,b)\in\mathbb{R}^p\times\mathbb{R}} \mathcal{L}(\alpha) = \inf_{w|\forall i, y_i(w^\mathsf{T} w_i + b)\geq 1} \frac{1}{2}\|w\|^2.$$

Pour rappel, cette identité se prouve facilement géométriquement (du moins dans le cas d'une seule contrainte d'inégalité) : la Figure 4.2.3 propose une illustration.

Comme on sait au moins résoudre $\inf_{(w,b)\in\mathbb{R}^p\times\mathbb{R}} \mathcal{L}(\alpha)$ dont la solution est donnée par $w = \sum_{i=1}^n \alpha_i y_i x_i$ et $0 = \sum_{i=1}^n \alpha_i y_i$ (qui permet de retrouver b), il ne reste alors plus qu'à résoudre la maximisation externe sur les α_i après avoir remplacé w par sa valeur, de sorte qu'on est amené à résoudre

$$\sup_{\alpha\geq 0} -\frac{1}{2}\sum_{i,j=1}^n \alpha_i\alpha_j y_i y_j x_i^\mathsf{T} x_j + \sum_{i=1}^n \alpha_i = \sup_{\alpha\geq 0} -\frac{1}{2}\alpha^\mathsf{T} D_y X^\mathsf{T} X D_y \alpha + \alpha^\mathsf{T} 1_n$$

sous la contrainte additionnelle $0 = \sum_{i=1}^n \alpha_i y_i$, où dans la deuxième ligne on a utilisé une forme vectorielle plus élégante en écrivant $X = [x_1, \ldots, x_n] \in \mathbb{R}^{p\times n}$, $D_y = \mathrm{diag}(y_1, \ldots, y_n)$, $\alpha = [\alpha_1, \ldots, \alpha_n]^\mathsf{T}$ et $1_n = [1, \ldots, 1]^\mathsf{T} \in \mathbb{R}^n$. C'est le problème *dual* associé à l'optimisation *primale* étudiée jusqu'ici.

Sous cette forme, il est clair qu'on a affaire à nouveau à un problème d'optimisation quadratique en le vecteur α sous contrainte : (i) d'une égalité $0 = \sum_{i=1}^n \alpha_i y_i$ et (ii) de n inégalités $\alpha_1, \ldots, \alpha_n \geq 0$. Contrairement au problème primal, on a donc une contrainte d'égalité supplémentaire ; celle-ci peut cependant être éliminée par exemple en écrivant que, sous cette contrainte, $\alpha = P_y\alpha$ où $P_y = I_n - \frac{1}{n}yy^\mathsf{T}$ est un projecteur orthogonal au vecteur $y = [y_1, \ldots, y_n]^\mathsf{T}$, et on peut alors remplacer $-\frac{1}{2}\alpha^\mathsf{T} D_y X^\mathsf{T} X D_y \alpha + \alpha^\mathsf{T} 1_n$ par $-\frac{1}{2}\alpha^\mathsf{T} P_y D_y X^\mathsf{T} X D_y P_y \alpha + \alpha^\mathsf{T} P_y 1_n$ et optimiser sur $\alpha \in \mathbb{R}^n$ tel que $P_y\alpha \geq 0$. À partir de là, on peut à nouveau utiliser une des méthodes de points intérieurs ou proximale discutée précédemment.

Passer par le dual est en général utile pour résoudre plus simplement un problème d'optimisation, notamment en réduisant le nombre de variables à optimiser. Ici, on passe au contraire d'un problème à $p+1$ variables (celles de w et b) à un problème à n variables (celle de α), et donc, étant donné que n est généralement bien plus grand que p, cette résolution semble inefficace. Mais cette formulation a en fait un autre intérêt, dont nous avons préalablement discuté, qui est premièrement

d'observer que le vecteur normal $w^\star$ solution au problème s'écrit sous la forme :

$$w^\star = \sum_{i=1}^{n} \alpha_i^\star y_i x_i$$

avec $\alpha^\star$ la solution du dual. Et donc, l'hyperplan séparateur recherché (à la valeur de b près) est entièrement déterminé par une combinaison linéaire des données $y_i x_i$.

Mais, plus important encore, rappelons (voir le descriptif de la Figure 4.2.3 pour un rappel visuel) que le multiplicateur de Lagrange α_i n'est non nul *que* si la contrainte associée $y_i(x_i^\mathsf{T} w^\star + b^\star) \geq 1$ est *active*, à savoir si et seulement si $y_i(x_i^\mathsf{T} w^\star + b^\star) = 1$. Or, si les points x_i sont distribués arbitrairement dans $\mathbb{R}^p$, cette équation en les $p+1$ variables de $(w^\star, b^\star)$ ne peut être valable que pour un maximum de $p+1$ vecteurs x_i (en forme vectorielle, on aurait $[X^\mathsf{T}\ 1_n]\left[\begin{smallmatrix} w^\star \\ b^\star \end{smallmatrix}\right] = y$, qui n'est inversible en général que si $[X^\mathsf{T}\ 1_n]$ est de taille $(p+1)\times(p+1)$ et donc $n = p+1$ au maximum).

Ainsi, on ne peut avoir plus de $p+1$ valeurs solutions non nulles $\alpha_i^\star$ au problème dual, et $w^\star$ se trouve être une combinaison linéaire d'un nombre généralement réduit de $y_i x_i$. Ces vecteurs x_i, se trouvant par définition *sur l'hyperplan séparateur* sont appelés *vecteurs support* car, de fait, ils "supportent" (ou définissent) les hyperplans parallèles $\mathcal{H}_1$, $\mathcal{H}_0$ et $\mathcal{H}_{-1}$ recherchés. Un petit calcul géométrique nous confirme en effet ce résultat : pour définir un hyperplan affine de $\mathbb{R}^p$, il nous faut p vecteurs ($p-1$ pour définir l'hyperplan de référence, ou sa normale, et 1 vecteur supplémentaire pour définir l'ordonnée à l'origine) ; il nous faut de plus un vecteur additionnel pour définir la distance entre les hyperplans $\mathcal{H}_1$ et $\mathcal{H}_{-1}$, et donc bien $p+1$ vecteurs au total. On peut s'en rendre à nouveau visuellement compte dans la Figure 4.0.1, ici pour $p = 2$.

Autre avantage crucial, nous le verrons par la suite, de la formulation $w^\star = \sum_{i=1}^n \alpha_i^\star y_i x_i$ (dont la somme est effectivement réduite généralement à $p+1$ termes), est que la classification d'un nouveau vecteur x consiste alors à évaluer le signe de :

$$(w^\star)^\mathsf{T} x + b^\star = \sum_{i | x_i \text{ vecteur support}} \alpha_i^\star y_i x_i^\mathsf{T} x + b^\star$$

de sorte qu'il suffit d'évaluer les $(p+1)$ produits scalaires $x_i^\mathsf{T} x$ pour tous les x_i vecteurs support. Lorsque nous travaillerons, non plus avec les données x_i elles-mêmes mais avec des vecteurs de représentation $\phi(x_i)$ potentiellement de grande dimension, le produit scalaire $\phi(x_i)^\mathsf{T}\phi(x)$ pourra être plus aisément évalué grâce à *l'astuce du noyau* que nous avons déjà brièvement évoqué. Il en va de même pour les $\alpha_i^\star$ et $b^\star$ qui ne dépendent ici que des produits scalaires $x_i^\mathsf{T} x_j$ (rappelons que α est solution du problème de minimisation sous contrainte $-\frac{1}{2}\alpha^\mathsf{T} D_y X^\mathsf{T} X D_y \alpha + \alpha^\mathsf{T} 1_n$).

4.2.4. Limitations et SVM *soft margin*. Nous avons longuement discuté de SVMs sous leur version dite "dure" (*hard margin*). Celle-ci, rappelons le, présuppose l'existence d'une solution aux contraintes "dures" (on serait tenté de dire "excessivement dures") $y_i(w^\mathsf{T} w_i + b) \geq 1$ pour tout $i \in \{1, \ldots, n\}$. Cela n'est pas toujours possible et il est simple de s'en convaincre en prenant dans $\mathbb{R}^2$ l'ensemble des (x_i, y_i) suivant : $([1,1],1), ([-1,-1],1)$ pour la classe 1 et $([1,-1],-1), ([-1,1],-1)$ pour la classe -1. Ces vecteurs ne sont pas linéairement séparable par un hyperplan de sorte que le SVM "dur" n'a aucune solution admissible (le problème d'optimisation associé n'est pas réalisable).

Si on souhaite malgré tout produire un hyperplan "séparateur", on devra donc autoriser la présence d'erreur de classification de certains vecteurs de la base d'apprentissage. Une façon d'opérer est de pénaliser ces points, mais d'un coût inférieur à l'infini (comme c'est le cas pour le SVM dur). L'approche suivie par les *SVMs à*

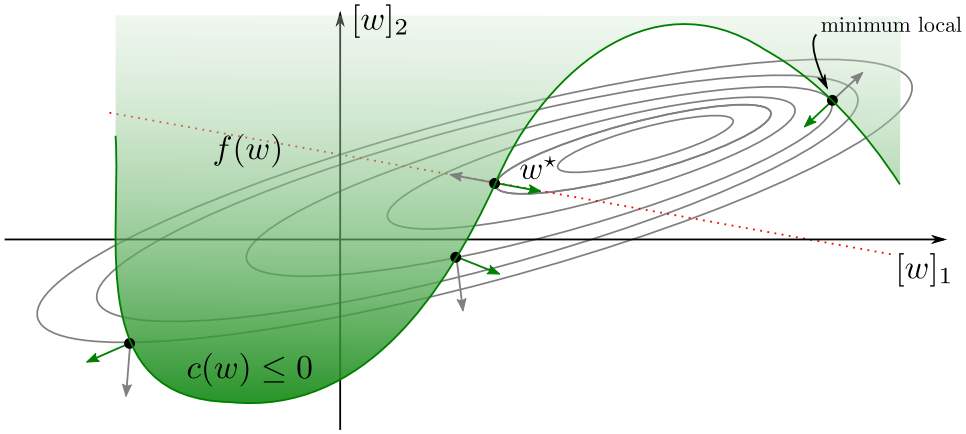

FIGURE 4.2.3. Schéma de preuve de la formule de dualité de Lagrange dans le cas de l'optimisation de $f(w)$ sous une contrainte d'inégalité $c(w) \neq 0$: en tout minimum local $w^\star$ de $f(w)$ (unique dans le cas convexe), le gradient de f s'oppose au gradient de c et donc $\nabla_w\{f(w)+\alpha c(w)\} = 0$ pour un certain $\alpha \geq 0$. Si un minimum local *non contraint* $w^\star$ de $f(w)$ satisfait la contrainte $c(w^\star) \neq 0$, alors $\nabla f(w^\star) = 0$ et alors $\alpha = 0$: la contrainte est *inactive*. Par contre, tout minimum local $w^\star$ contraint de $f(w)$ qui n'est pas un minimum local non contraint (pour lequel donc $\nabla f(w^\star) < 0$) induit $\alpha > 0$: la contrainte est *active*.

marge souple (*soft-margin SVM*) est d'écrire le problème d'optimisation toujours sous la forme d'une minimisation du risque empirique $\sum_{i=1}^n \ell(g(x_i), y_i)+h(g)$, mais cette fois-ci avec une fonction de coût ℓ ne prenant pas des valeurs infinies : précisément, on cherchera à résoudre

$$\min_{(w,b)} \frac{1}{2}\|w\|^2 + \gamma \sum_{i=1}^n \max\{0, 1 - y_i(w^\mathsf{T} x_i + b)\}$$

pour $\gamma > 0$ un hyperparamètre du problème (qu'il pourra s'agir d'optimiser lui aussi). En utilisant la fonction $\max\{0, z\}$, on ne pénalise donc que les valeurs positives de z, en l'occurrence seulement les indices i pour lesquelles $1-y_i(w^\mathsf{T} x_i+b) \geq 0$ ou de manière équivalente $y_i(w^\mathsf{T} x_i+b) \leq 1$: c'est-à-dire les vecteurs x_i qui seraient du mauvais côté de l'hyperplan séparateur. La pénalisation pour ces points est proportionnelle à γ, de sorte que, pour $\gamma \to \infty$, on devrait retrouver la solution SVM "dure", alors que pour $\gamma \to 0$, on donnera énormément de liberté aux vecteurs à ne pas être correctement classés.

La fonction $(w, b) \mapsto \max\{0, 1 - y_i(w^\mathsf{T} x_i + b)\}$ est convexe, de sorte que le problème d'optimisation général reste convexe, mais elle n'est cependant pas différentiable. On ne peut donc pas la résoudre numériquement par une simple descente de gradient. On peut cependant faire appel à des algorithmes proximaux qui savent gérer cette situation (voir la section précédente). Avant l'avènement de ces approches proximales, une astuce pour retomber sur un problème d'optimisation différentiable était (paradoxalement) de recréer des contraintes dures en ajoutant des variables qu'on appelle *slack variables*, et de réécrire le problème précédent sous la forme

$$\min_{(w,b,\{\xi_i\}_{i=1}^n)\in\mathbb{R}^p\times\mathbb{R}\times\mathbb{R}_+^n} \frac{1}{2}\|w\|^2 + \gamma \sum_{i=1}^n \xi_i, \quad \text{tel que} \quad \forall i,\ y_i(w^\mathsf{T} x_i + b) \geq 1 - \xi_i.$$

On retombe dans ce cas essentiellement dans un problème de la forme des SVM "durs", que l'on pourra résoudre avec les mêmes outils. Sous sa forme duale, ce problème s'écrit de manière très similaire au cas "dur" :

$$\max_{\alpha \geq 0} -\frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j y_i y_j x_i^\mathsf{T} x_j + \sum_{i=1}^{n} \alpha_i$$

mais maintenant avec à la fois la contrainte $\sum_{i=1}^n \alpha_i y_i = 0$ mais également, pour tout $i \in \{1, \dots, n\}$, les contraintes supplémentaires $\alpha_i \leq \gamma$. On recouvre bien le cas des SVMs durs lorsque $\gamma \to \infty$ où ces dernières contraintes n'existent effectivement pas. À nouveau, le problème se résout assez simplement en utilisant les outils d'optimisation convexe évoqués dans la section précédente.

Il est intéressant de remarquer que, sous ce nouveau jeu de contraintes, $\alpha_i = 0$ lorsque la contrainte $y_i(w^\mathsf{T} w_i + b) \geq 1 - \xi_i$ est vérifiée au sens strict, à savoir $y_i(w^\mathsf{T} w_i + b) > 1 - \xi_i$. Dans ce cas, si la variable ξ_i est non nulle (donc strictement positive), il est possible de la réduire et de faire baisser le coût relaxé $\sum_{j=1}^n \xi_j$, et donc d'obtenir une meilleure solution : ainsi, $\alpha_i = 0 \Rightarrow \xi_i = 0$. Au contraire, si $\alpha_i > 0$, cela signifie que $y_i(w^\mathsf{T} x_i + b) = 1 - \xi_i$ et donc : (i) si $\xi_i = 0$, on a affaire à un vecteur de support, (ii) si $\xi_i > 0$, on a affaire à un vecteur du mauvais côté de l'hyperplan séparateur. Pour ces derniers vecteurs mal classés, comme $w = \sum_j \alpha_j y_j x_j$ et que

$$y_i(w^\mathsf{T} x_i + b) = y_i(\sum_j \alpha_j y_j x_j^\mathsf{T} x_i + b) = y_i(\sum_{j \neq i} \alpha_j y_j x_j^\mathsf{T} x_i + b) + \alpha_i \|x_i\|^2 = 1 - \xi_i$$

on pourrait à nouveau choisir de diminuer la valeur de ξ_i en augmenter par compensation celle de α_i, ce qui améliorerait le résultat l'optimisation. Par conséquent, cela signifie que $\alpha_i = \gamma$ dans ce cas (on augmente α_i jusqu'à sa contrainte, ou alors c'est que ξ_i n'est pas positif !). Pour résumer, on se trouve finalement dans la situation suivante :

- si $\alpha_i = 0$, le vecteur x_i est bien classé ;
- si $\alpha_i \in (0, \gamma)$, le vecteur x_i est "juste" bien classé : c'est un vecteur support de l'hyperplan séparateur ;
- si $\alpha_i = \gamma$, le vecteur x_i est mal classé.

Il faut bien sûr faire attention de ne pas croire qu'en augmentant la valeur de γ, on assurerait de bien classer plus de vecteurs. De fait, cela est vrai pour des petites valeurs de γ, mais si le problème n'est *pas réalisable*, le nombre de vecteurs qui peuvent être correctement classé reste borné, et un effet plateau apparaitra.

4.3. Régression des moindres carrés et LSSVM

Nous avons vu dans la section précédente que les machines à vecteurs de support, sous forme de marge stricte ou souple, sont des instances d'algorithmes de minimisation du risque empirique, pour les fonctions de coût $\imath(\{t > 1\})$ (marge stricte) et $\max\{0, 1 - t\}$ (marge souple). Les deux méthodes sont assez intuitives mais donnent lieu à un problème d'optimisation convexe dont la solution n'est pas explicite et qu'il est parfait délicat de résoudre numériquement. C'est notamment le cas lorsque le nombre de données n est relativement grand.

Une façon simple d'obtenir une solution explicite est d'opter pour un *coût quadratique*, à savoir $\frac{1}{2}(t-1)^2$. Précisément, on cherchera ici, non pas à être "du bon côté" de l'hyperplan séparateur $\mathcal{H}_1$ ou $\mathcal{H}_{-1}$, mais plutôt à être "au plus près" de chaque hyperplan. On sort de fait quelque peu de l'esprit initial de séparation de l'espace en deux classes distinctes (séparées par une grande marge) pour plutôt chercher à "concentrer" l'espace *autour* de deux hyperplans qui porteraient eux-mêmes (ou plutôt dans leur proche voisinage) les vecteurs de chaque classe.

En opérant de la sorte, on obtient alors le problème d'optimisation suivant

$$\min_{(w,b)} \frac{1}{2}\|w\|^2 + \frac{\gamma}{2}\sum_{i=1}^{n}\left(y_i(w^\mathsf{T}x_i+b)-1\right)^2 = \min_{(w,b)} \frac{1}{2}\|w\|^2 + \frac{\gamma}{2}\sum_{i=1}^{n}\left(w^\mathsf{T}x_i+b-y_i\right)^2$$

à nouveau pour $\gamma > 0$ un hyperparamètre qui pénalise les points les plus éloignés des hyperplans. L'avantage fondamental de cette formulation est qu'elle donne lieu à une solution explicite. En effet, il s'agit d'un problème convexe en (w, b) et quadratique en ces variables. Il suffit donc d'annuler le gradient, de sorte à obtenir (en utilisant plutôt la deuxième forme du minimum) :

$$0 = w + \gamma\sum_{i=1}^{n} x_i\left(x_i^\mathsf{T}w+b-y_i\right) = \left(I_p+\gamma XX^\mathsf{T}\right)w + \gamma X(b1_n - y)$$

$$0 = \sum_{i=1}^{n}\left(x_i^\mathsf{T}w+b-y_i\right) = nb + 1_n^\mathsf{T}(X^\mathsf{T}w - y).$$

On obtient ainsi

$$\begin{aligned} w &= \gamma\left(I_p+\gamma XX^\mathsf{T}\right)^{-1}X(y-b1_n) = \left(XX^\mathsf{T}+\gamma^{-1}I_p\right)^{-1}X(y-b1_n)\\ &= X\left(X^\mathsf{T}X+\gamma^{-1}I_n\right)^{-1}(y-b1_n)\\ b &= \frac{1}{n}1_n^\mathsf{T}y - \frac{1}{n}1_n^\mathsf{T}X^\mathsf{T}w = \frac{1}{n}1_n^\mathsf{T}y - \frac{1}{n}1_n^\mathsf{T}X^\mathsf{T}X\left(X^\mathsf{T}X+\gamma^{-1}I_n\right)^{-1}(y-b1_n)\\ &= \frac{1}{n}1_n^\mathsf{T}y - \frac{1}{n}1_n^\mathsf{T}(y-b1_n) + \frac{\gamma^{-1}}{n}1_n^\mathsf{T}\left(X^\mathsf{T}X+\gamma^{-1}I_n\right)^{-1}(y-b1_n) \end{aligned}$$

où, pour passer de la première à la seconde ligne de l'expression de w on a utilisé le fait que $A(BA+I)^{-1} = (AB+I)^{-1}A$ (qui se vérifie facilement en multipliant les deux côtés de l'égalité par $BA+I$ par la droite), et pour passer de la première à la seconde ligne de l'expression de b, on a utilisé $A(A+I)^{-1} = (A+I-I)(A+I)^{-1} = I-(A+I)^{-1}$. De l'expression de b, on obtient finalement, en rassemblant les termes

$$0 = 1_n^\mathsf{T}\left(X^\mathsf{T}X+\gamma^{-1}I_n\right)^{-1}(y-b1_n)$$

ou de manière équivalente

$$b = \frac{1_n^\mathsf{T}\left(X^\mathsf{T}X+\gamma^{-1}I_n\right)^{-1}y}{1_n^\mathsf{T}\left(X^\mathsf{T}X+\gamma^{-1}I_n\right)^{-1}1_n}.$$

On obtient donc finalement la solution explicite des hyperplans $\mathcal{H}_0:\ w^\mathsf{T}z+b=0$, $\mathcal{H}_1:\ w^\mathsf{T}z+b=1$ et $\mathcal{H}_{-1}:\ w^\mathsf{T}z+b=-1$, pour (w, b) donnés par

$$\begin{aligned} w &= X\left(X^\mathsf{T}X+\gamma^{-1}I_n\right)^{-1}(y-b1_n)\\ b &= \frac{1_n^\mathsf{T}\left(X^\mathsf{T}X+\gamma^{-1}I_n\right)^{-1}y}{1_n^\mathsf{T}\left(X^\mathsf{T}X+\gamma^{-1}I_n\right)^{-1}1_n}. \end{aligned}$$

On retrouve donc finalement, et cela n'est pas trop étonnant, une simple régression linéaire, normalisée par l'hyperparamètre $\gamma > 0$. Plusieurs remarques sont intéressantes à formuler :

— **le paramètre** b : comme notre problème consiste à classer les données x_i de $\mathbb{R}^p$, le problème n'est pas affecté si on translate tous les x_i d'un même vecteur constant. En particulier, si on "recentre" les x_i par leur moyenne empirique $\frac{1}{n}\sum_i x_i = \frac{1}{n}X1_n$, cela revient à changer dans la solution ci-dessus la matrice X par $X - \frac{1}{n}X1_n1_n^\mathsf{T} = XP$ avec $P = I_n - \frac{1}{n}1_n1_n^\mathsf{T}$ (dont on vérifie bien que la i-ème colonne est $x_i - \frac{1}{n}X1_n$). Mais dans ce cas, il n'est pas difficile de vérifier que 1_n est un vecteur propre de $(PX^\mathsf{T}XP+\gamma^{-1}I_n)^{-1}$

de valeur propre γ, de sorte que, injecté dans l'expression de b, on trouve simplement

$$b = \frac{1}{n} 1_n^\mathsf{T} y$$

qui est la moyenne des valeurs de y. Ainsi, si les x_i sont centrées empiriquement, l'ordonnée à l'origine de l'hyperplan solution est nécessairement égale à la moyenne des étiquettes y_i. En particulier, si les classes sont équilibrées (le nombre n_1 d'éléments de la classe 1 est égal au nombre n_{-1} d'éléments de la classe -1), on trouve simplement $b = 0$ et l'hyperplan affine central est un hyperplan linéaire. Par ailleurs, si les données X sont recentrées et donc égales à XP, alors 1_n est un vecteur propre droit de $XP(PX^\mathsf{T}XP + \gamma^{-1} I_n)^{-1}$ associé à la valeur propre 0, de sorte que w devient indépendant de b et s'écrit plus directement

$$w = XP\left(PX^\mathsf{T}XP + \gamma^{-1} I_n\right)^{-1} y.$$

— **le paramètre** γ : si on prend la limite $\gamma \to \infty$, on va chercher à forcer les données à être proches de leurs hyperplans $\mathcal{H}_1$ et $\mathcal{H}_{-1}$ respectifs. Supposons pour simplicité que tout échantillon de taille $p-1$ des x_i est linéairement indépendant (c'est par exemple le cas avec probabilité un si les x_i sont tirés i.i.d. dans $\mathbb{R}^p$ selon une loi à densité). En se rappelant que X est de taille $p \times n$ et $X^\mathsf{T}X$ de taille $n \times n$ mais de rang maximal $\max\{p, n\}$, deux cas de figure peuvent alors se présenter :

— si $p \geq n$, alors $X^\mathsf{T}X$ est de rang n et on peut donc sans crainte prendre la limite $\gamma \to \infty$ dans les équations de w et b, pour obtenir

$$w^\infty = X(X^\mathsf{T}X)^{-1}(y - b^\infty 1_n)$$
$$b^\infty = \frac{1_n^\mathsf{T}(X^\mathsf{T}X)^{-1} y}{1_n^\mathsf{T}(X^\mathsf{T}X)^{-1} 1_n}$$

(attention, dans ce cas par contre, on ne peut plus aussi simplement recentrer les données comme évoqué ci-dessus puisque $PX^\mathsf{T}XP$ est de range $n-1$). On obtient alors un régresseur des *moindres carrés* (*least squares* en anglais) très classique.

— si $p < n$, $X^\mathsf{T}X$ est de rang $p < n$ et n'est plus inversible. On utilisera dans ce cas le fait que $w = (XX^\mathsf{T} + \gamma^{-1} I_p)^{-1} X(y - b1_n)$ et, si on recentre les données comme évoqué précédemment, on aura alors simplement

$$w^\infty = (XPX^\mathsf{T})^{-1} XPy$$
$$b^\infty = \frac{1}{n} 1_n^\mathsf{T} y.$$

C'est à nouveau une forme de régression linéaire, mais sur les lignes plutôt que les colonnes de la matrice X.

Si on prend au contraire la limite $\gamma \to 0$ (en voulant fortement éviter le surapprentissage par exemple), on trouve

$$w^0 \sim \gamma X(y - b^0 1_n)$$
$$b^0 = \frac{1}{n} 1_n^\mathsf{T} y$$

qui nous amène plutôt maintenant vers une solution de type *filtrage adapté* (*matched filter* en anglais). À savoir, w^0 ne consiste ici qu'en une simple combinaison linéaire des x_i pondérés par leurs étiquettes y_i.

— **les vecteurs support ?** : comme pour les SVMs à marge souple, on pourrait choisir d'écrire le problème d'optimisation sous la forme

$$\min_{(w,b,\{\xi_i\}_{i=1}^n)} \frac{1}{2}\|w\|^2 + \frac{\gamma}{2}\sum_{i=1}^n e_i^2, \quad \text{tel que} \quad \forall i,\ e_i = w^\mathsf{T} x_i + b - y_i.$$

Dans ce cas, en utilisant les multiplicateurs de Lagrange $\alpha = (\alpha_1, \ldots, \alpha_n)$ associés aux n contraintes d'égalités, on trouverait

$$w = \sum_{i=1}^n \alpha_i x_i = X\alpha$$

et donc à nouveau (comme pour les SVMs à marges) une combinaison linéaire des x_i avec cette fois-ci les coefficients

$$\alpha = \left(X^\mathsf{T} X + \gamma^{-1} I_n\right)^{-1} (y - b1_n).$$

Et donc, contrairement aux SVMs à marges, ici en général les α_i sont non nuls et sont donc *tous des vecteurs support*. C'est un point qui, numériquement, joue en la défaveur de LSSVM de par sa nature "non économe" en les données x_i qu'il utilise toutes.

Une fois établi le régresseur (w, b), la fonction de décision $g(x)$ pour une donnée x non étiquetée est donnée par :

$$\begin{aligned} g(x) &= x^\mathsf{T} w + b \\ &= x^\mathsf{T} X \left(X^T X + \gamma^{-1} I_n\right)^{-1} (y - b1_n). \end{aligned} \tag{4.3.1}$$

En particulier, cette forme permet de calculer facilement l'erreur d'entrainement $E_{\text{train}}(g)$. En effet, pour $\ell(g(x_i), y_i) = (g(x_i) - y_i)^2$, et en renommant n ci-dessus en n_{train} (pour les besoins de différentiation avec n_{test} que l'on définira par la suite) :

$$\begin{aligned} E_{\text{train}}(g) &= \frac{1}{n_{\text{train}}} \sum_{i=1}^{n_{\text{train}}} (g(x_i) - y_i)^2 \\ &= \frac{1}{n_{\text{train}}} \sum_{i=1}^{n_{\text{train}}} (w^\mathsf{T} x_i + b - y_i)^2 \\ &= \frac{1}{n_{\text{train}}} (X^\mathsf{T} w + b1_{n_{\text{train}}} - y)^\mathsf{T} (X^\mathsf{T} w + b1_{n_{\text{train}}} - y). \end{aligned}$$

En calculant explicitement $X^\mathsf{T} w + b1_{n_{\text{train}}} - y$, on obtient alors

$$\begin{aligned} X^\mathsf{T} w + b1_{n_{\text{train}}} - y &= X^\mathsf{T} X (X^\mathsf{T} X + \gamma^{-1} I_{n_{\text{train}}})^{-1} (y - b1_{n_{\text{train}}}) + b1_{n_{\text{train}}} - y \\ &= -\gamma^{-1} (X^\mathsf{T} X + \gamma^{-1} I_{n_{\text{train}}})^{-1} (y - b1_{n_{\text{train}}}) \end{aligned}$$

où dans la deuxième ligne, on a à nouveau utilisé l'astuce $A(A+I)^{-1} = (A + I - I)(A+I)^{-1} = I - (A+I)^{-1}$. Et donc finalement,

$$\begin{aligned} E_{\text{train}}(g) &= \frac{\gamma^{-2}}{n} (y - b1_{n_{\text{train}}})^\mathsf{T} \left(X^\mathsf{T} X + \gamma^{-1} I_{n_{\text{train}}}\right)^{-2} (y - b1_{n_{\text{train}}}) \\ &= \frac{1}{n_{\text{train}}} (y - b1_{n_{\text{train}}})^\mathsf{T} \left(I_{n_{\text{train}}} + \gamma X^\mathsf{T} X\right)^{-2} (y - b1_{n_{\text{train}}}). \end{aligned}$$

De la même manière, pour un ensemble de données de test $\tilde{X} = [\tilde{x}_1, \dots, \tilde{x}_{n_{\text{test}}}]$ associées à des étiquettes $\tilde{y} = [\tilde{y}_1, \dots, \tilde{y}_{n_{\text{test}}}]^\mathsf{T}$, on trouve

$$\begin{aligned} E_{\text{test}}(g) &= \frac{1}{n_{\text{test}}} \sum_{i=1}^{n_{\text{test}}} (g(\tilde{x}_i) - \tilde{y}_i)^2 \\ &= \frac{1}{n_{\text{test}}} \sum_{i=1}^{n_{\text{test}}} (w^\mathsf{T} \tilde{x}_i + b - \tilde{y}_i)^2 \\ &= \frac{1}{n_{\text{test}}} (\tilde{X}^\mathsf{T} w + b 1_{n_{\text{test}}} - \tilde{y})^\mathsf{T} (\tilde{X}^\mathsf{T} w + b 1_{n_{\text{test}}} - \tilde{y}) \end{aligned}$$

avec

$$\tilde{X}^\mathsf{T} w + b 1_{n_{\text{test}}} - \tilde{y} = \tilde{X}^\mathsf{T} X (X^\mathsf{T} X + \gamma^{-1} I_{n_{\text{train}}})^{-1} (y - b 1_{n_{\text{train}}}) + b 1_{n_{\text{test}}} - \tilde{y}.$$

Ces deux expressions explicites de $E_{\text{train}}(g)$ et $E_{\text{test}}(g)$ sont intéressantes en cela qu'elles sont théoriquement interprétables et numériquement très simples à évaluer. On remarque notamment que $E_{\text{train}}(g)$ est une fonction monotone décroissante de γ avec pour limites (toujours en faisant l'hypothèse que tout échantillon de taille p des x_i est linéairement indépendant)

$$E_{\text{train}}(g) \underset{\gamma \to \infty}{\longrightarrow} \begin{cases} 0 & , \ p \geq n_{\text{train}} \\ \frac{1}{n_{\text{train}}} (y - b 1_{n_{\text{train}}})^\mathsf{T} \Pi_X^\perp (y - b 1_{n_{\text{train}}}) & , \ p < n_{\text{train}} \end{cases}$$

où $\Pi_X^\perp$ est le projecteur orthogonal à l'espace engendré par les p lignes de la matrice X (avec $p < n_{\text{train}}$). Pour se convaincre de la forme prise par la limite lorsque $p < n$, il suffit de diagonaliser $X^\mathsf{T} X$ sous la forme $X^\mathsf{T} X = U \Lambda U^\mathsf{T}$ où $\Lambda = \text{diag}(\lambda_1, \dots, \lambda_p, 0, \dots, 0)$ et U la base orthonormale associée : on a ainsi $I_{n_{\text{train}}} + \gamma X^\mathsf{T} X = \gamma U (I_{n_{\text{train}}} + \gamma \Lambda) U^\mathsf{T}$ avec $I_{n_{\text{train}}} + \gamma \Lambda = \text{diag}(1 + \gamma \lambda_1, \dots, 1 + \gamma \lambda_n, 1, \dots, 1)$. Dans la limite $\gamma \to \infty$, en prenant l'inverse, $(I_{n_{\text{train}}} + \gamma \Lambda)^{-1} \to \text{diag}(0, \dots, 0, 1, \dots, 1)$, et le résultat suit.

Ce résultat n'est pas étonnant : lorsque $p \geq n_{\text{train}}$, la régression par moindres carrés (dans la limite $\gamma \to \infty$) permet d'inverser exactement les $n_{\text{train}} \leq p$ contraintes $w^\mathsf{T} x_i + b = y_i$, de sorte que l'erreur est nulle. Au contraire, lorsque $p < n_{\text{train}}$, il y a trop de contraintes à satisfaire et la solution par moindres carrés "laisse de l'énergie" : on remarque d'ailleurs que si les x_i, y_i étaient tirés uniformément d'une loi invariante par rotation, alors pour n_{train} assez grand on aurait $\frac{1}{n_{\text{train}}} (y - b 1_{n_{\text{train}}})^\mathsf{T} \Pi_X^\perp (y - b 1_{n_{\text{train}}}) \simeq (n_{\text{train}} - p)/n_{\text{train}} = 1 - p/n_{\text{train}}$, faisant ainsi apparaitre le rapport p/n_{train} dans l'énergie "délaissée" par le régresseur.

Attention par contre, la limite $E_{\text{train}}(g) \to_{\gamma \to \infty} 0$ n'est pas du tout une bonne chose ! Elle montre bien que nous sommes en présence d'un cas de surapprentissage total, qui risque d'induire des valeurs très élevées de $E_{\text{test}}(g)$ associées. Pour bien s'en rendre compte, la Figure 4.3.1 présente les performances de l'algorithme dans le cadre de la classification d'un mélange de deux gaussiennes $\mathcal{N}(\pm \mu, I_p)$ avec $\mu = (4, 0, \dots, 0)^\mathsf{T}$, en fonction des différents paramètres du problème. On observe bien la décroissance de $E_{\text{train}}(g)$ à la fois lorsque $p < n_{\text{train}}$ ou $p > n_{\text{train}}$, mais avec une limite nulle lorsque $p > n_{\text{train}}$. Quant aux erreurs $E_{\text{test}}(g)$, elles diffèrent de comportement : admettant un minimum local lorsque $\gamma \simeq 10^{-2}$ ou décroissant avec γ.

La figure permet d'ailleurs bien de se rendre compte ici de l'importante différence entre $E_{\text{train}}(g)$ et $E_{\text{test}}(g)$ qui sont pourtant des erreurs évaluées sur des données statistiquement équivalentes ! Ce qui change cependant, et dont nous avons déjà parlé, c'est la dépendance en les données d'entrainement de g qui rend $E_{\text{train}}(g)$ statistiquement très différent de $E_{\text{test}}(g)$ (ce dernier étant l'objectif à minimiser).

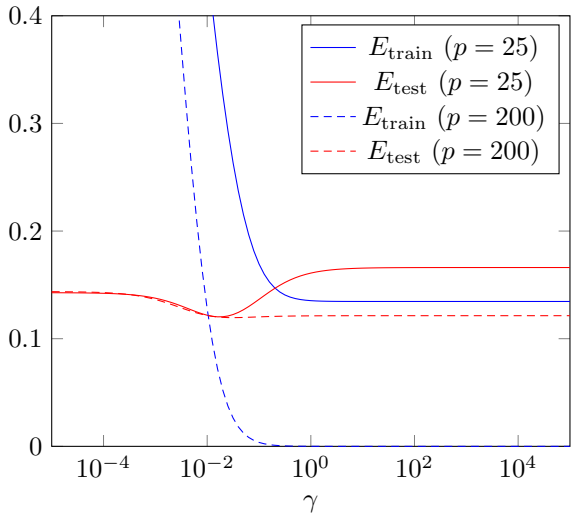

FIGURE 4.3.1. Performance E_{train} et E_{test} de l'algorithme LSSVM pour différentes valeurs de γ et p. Ici pour $x_i \sim \mathcal{N}(\pm\mu, I_p)$ i.i.d. avec $\mu = (1, 0, \ldots, 0)^\mathsf{T}$ avec $2n_{\text{train}} = 2n_{\text{test}} = n = 100$.

LISTING 4.1. Code Matlab de la Figure 4.3.1.

```
n=100;
p=25;

% Génération des données N(+/-mu,I)
mu=[4;zeros(p-1,1)];
y=sign(randn(n,1));
X=randn(p,n)+mu*y';

% Découpage en 'train' et 'test'
ntrain=n/2;
ntest=n-ntrain;
Xtrain=X(:,1:ntrain);
ytrain=y(1:ntrain);
Xtest=X(:,ntrain+1:end);
ytest=y(ntrain+1:end);

% Design de l'hyperplan LSSVM
b=@(gamma)sum(inv(Xtrain'*Xtrain+1/gamma*eye(ntrain))*ytrain)/sum(sum(
    inv(Xtrain'*Xtrain+1/gamma*eye(ntrain))));
w=@(gamma)Xtrain*inv(Xtrain'*Xtrain+1/gamma*eye(ntrain))*(ytrain-b(
    gamma)*ones(ntrain,1));gammas=10.^(-5:1e-1:5);

Etrain=zeros(1,length(gammas));
Etest =zeros(1,length(gammas));

% Calcul des erreurs
igamma=1;
gammas = 10.^(-5:1e-2:5);
for gamma=gammas
```

```
28          Etrain(igamma)=1/ntrain*norm(Xtrain'*w(gamma)+b(gamma)*ones(
                ntrain,1)-ytrain)^2;
29          Etest(igamma) =1/ntest *norm(Xtest' *w(gamma)+b(gamma)*ones(
                ntest ,1)-ytest )^2;
30          igamma=igamma+1;
31  end
32
33  figure
34  semilogx(gammas,[Etrain;Etest]);
```

4.4. Régression logistique

Nous avons montré que les algorithmes SVM (à marge douce ou dure) ainsi que LSSVM découlent tous d'une forme spécifique de la méthode de la *minimisation du risque empirique régularisé*, seulement pour des fonctions de coûts différentes. Nous avons également montré qu'il s'agissait dans tous les cas d'une forme plus ou moins élaborée d'une problème de régression, où il s'agit de découvrir un régresseur $w \in \mathbb{R}^p$ (ainsi qu'un terme de biais $b \in \mathbb{R}$) des données d'entrée $x_1, \dots, x_n \in \mathbb{R}^p$ sur les sorties $y_1, \dots, y_n \in \mathbb{R}$. Ce vecteur w ainsi que le biais b ont également une interprétation géométrique d'hyperplan séparateur, qui est, il faut l'admettre, la vision plus largement répandue des SVMs.

Néanmoins, si on garde la vision "régression" pour un moment, on trouve d'autres algorithmes d'apprentissage qui s'expriment aussi sous la forme de la *minimisation du risque empirique régularisé*. La plus populaire de ces méthodes est celle de la *régression logistique*. Celle-ci se base sur un modèle gaussien $\mathcal{N}(\pm\mu, C)$ pour les données x_i (attention, C est bien commun aux deux classes). Cependant, au contraire de l'analyse discriminante (LDA, QDA), il ne s'agira pas ici d'estimer les moyennes et covariances directement avant d'effectuer un test de maximum de vraisemblance. L'approche suivie par l'algorithme découle de l'observation suivante : pour un couple (x, y) pour lequel y est inconnu, la probabilité a posteriori de y sachant x est donnée, sous le modèle de mélange gaussien binaire par

$$\mathbb{P}(y|x) = \frac{\mathbb{P}(y)\mathbb{P}(x|y)}{\mathbb{P}(x)} = \frac{\mathbb{P}(y)\mathbb{P}(x|y)}{\mathbb{P}(x|y)\mathbb{P}(y) + \mathbb{P}(x|-y)\mathbb{P}(-y)}.$$

En développant le calcul et en faisant l'hypothèse que $\mathbb{P}(y) = \mathbb{P}(-y) = 1/2$, on obtient alors

$$\begin{aligned}\mathbb{P}(y|x) &= \frac{\exp(-\frac{1}{2}(x-y\mu)^\mathsf{T} C^{-1}(x-y\mu))}{\exp(-\frac{1}{2}(x-y\mu)^\mathsf{T} C^{-1}(x-y\mu)) + \exp(-\frac{1}{2}(x+y\mu)^\mathsf{T} C^{-1}(x+y\mu))} \\ &= \frac{1}{1+\exp(-2y\mu^\mathsf{T} C^{-1}x)} \\ &= \sigma(y\mu^\mathsf{T} C^{-1}x)\end{aligned}$$

si on nomme $\sigma(t) = (1+\exp(-t))^{-1}$ qu'on appelle communément la *fonction logistique sigmoïde* (ou *fonction logistique* ou *sigmoïde*) du fait de sa forme.

Ainsi, si on avait accès au vecteur $C^{-1}\mu$, il serait possible d'évaluer la vraisemblance de y et $-y$ sachant x. Une manière d'estimer $C^{-1}\mu$, dénotons w son estimateur, est d'utiliser la connaissance des couples d'entrainement (x_i, y_i) : en particulier, l'estimateur w de maximum de vraisemblance étant données les (x_i, y_i) s'écrit :

$$\sup_w \mathbb{P}\left(\{(x_i, y_i)\}_{i=1}^n | w\right) \propto \sup_w \mathbb{P}\left(\{y_i)\}_{i=1}^n | w, \{x_i\}_{i=1}^n\right) = \sup_w \prod_{i=1}^n \sigma(y_i w^\mathsf{T} x_i)$$

en utilisant l'indépendance des couples (x_i, y_i). En passant au logarithme et en rappelant que $\sigma(t) = (1 + \exp(-t))^{-1}$, on trouve alors que w minimise

$$\inf_w \sum_{i=1}^{n} \log\left(1 + \exp(-y_i w^\mathsf{T} x_i)\right).$$

C'est la fonction que nous chercherons donc à minimiser sur w.

Placé dans le contexte de la minimisation du risque empirique régularisé, on cherchera alors à résoudre :

$$\inf_{w \in \mathbb{R}^p} \frac{1}{2}\|w\|^2 + \gamma \sum_{i=1}^{n} \log\left(1 + \exp(-y_i w^\mathsf{T} x_i)\right).$$

Notons ici que, sous cette forme, le paramètre b des régresseurs précédent a disparu (ou est simplement fixé et égal à zéro).

Il n'est pas difficile de vérifier que la fonction $\rho(t) = \log(1 + \exp(-t))$ est bien convexe (on peut par exemple dériver deux fois pour obtenir $\rho'(t) = -\exp(-t)/(1+\exp(-t)) = -1 + 1/(1+\exp(-t))$ puis $\rho''(t) = \exp(-t)/(1+\exp(-t))^2 > 0$). On a donc à nouveau affaire à un problème d'optimisation convexe dont la solution est unique.

Un fort avantage de cette optimisation est que la fonction objectif est ici différentiable et soumise à aucune contrainte : elle peut donc être résolue par simple descente de gradient, en utilisant notamment le fait que

$$\nabla_w \left\{ \frac{1}{2}\|w\|^2 + \gamma \sum_{i=1}^{n} \log\left(1 + \exp(-y_i w^\mathsf{T} x_i)\right) \right\} = w - \gamma \sum_{i=1}^{n} \frac{y_i x_i}{1 + \exp(-y_i w^\mathsf{T} x_i)}.$$

Un exemple d'implémentation de la méthode est fournie dans le Listing 4.2.

4.5. Comparaison des techniques

Il est difficile de comparer les méthodes SVM, LSSVM et régression logistique théoriquement, étant donné qu'à la fois SVM et régression logistique n'admettent pas de solution explicite. Il est aussi délicat de comparer les fonctions de coût E_{train} et E_{test} qui sont basées sur différents objectifs (pour SVM une satisfaction des contraintes de barrière, pour régression logistique un maximum de vraisemblance, et pour LSSVM une distance euclidienne minimale aux hyperplans). Il est cependant possible dans le cadre de classification qui nous concerne de comparer, non plus les coûts "objectif" mais plutôt la probabilité de classification correcte induite par les méthodes. Celles-ci sont évidemment comparables entre toutes les familles d'algorithmes que l'on a considéré jusqu'ici. Cette probabilité de classification correcte sera dénotée $\mathbb{P}_{\text{train}}(g)$ (pour les données d'entrainement) et $\mathbb{P}_{\text{test}}(g)$ (pour les données de test), et est définie (si on divise comme précédemment les n données x_i en $x_1, \ldots, x_{\text{train}}$ pour la base d'entrainement et $n_{\text{train}} + 1, \ldots, n$ pour la base de test, par

$$\mathbb{P}_{\text{train}}(g) = \frac{1}{n_{\text{train}}} \sum_{i=1}^{n_{\text{train}}} 1_{\{y_i g(x_i) > 0\}}$$

$$\mathbb{P}_{\text{test}}(g) = \frac{1}{n_{\text{test}}} \sum_{i=n_{\text{train}}+1}^{n} 1_{\{y_i g(x_i) > 0\}}.$$

Dans la même configuration de mélange gaussien que dans la Figure 4.3.1, la Figure 4.5.1 propose une comparaison entre les performances $\mathbb{P}.$ des méthodes SVM, LSSVM, régression logistique ainsi également de kNN (méthode des k plus proches voisins). Pour ce type de données, il apparait clairement que kNN, bien que sous une

configuration très isotrope, a une performance très réduite, alors que SVM, LSSVM et régression logistique sont toutes trois très compétitives, avec néanmoins un fort avantage ici pour la régression logistique (ce qui n'est sûrement pas étonnant car elle se base sur une hypothèse gaussienne des données). Il est d'ailleurs notable que la valeur optimale pour γ se trouve être à peu près la même pour SVM et LSSVM, même si à nouveau les fonctions de coût diffèrent largement et toute interprétation, sans plus d'analyse, serait plutôt hasardeuse.

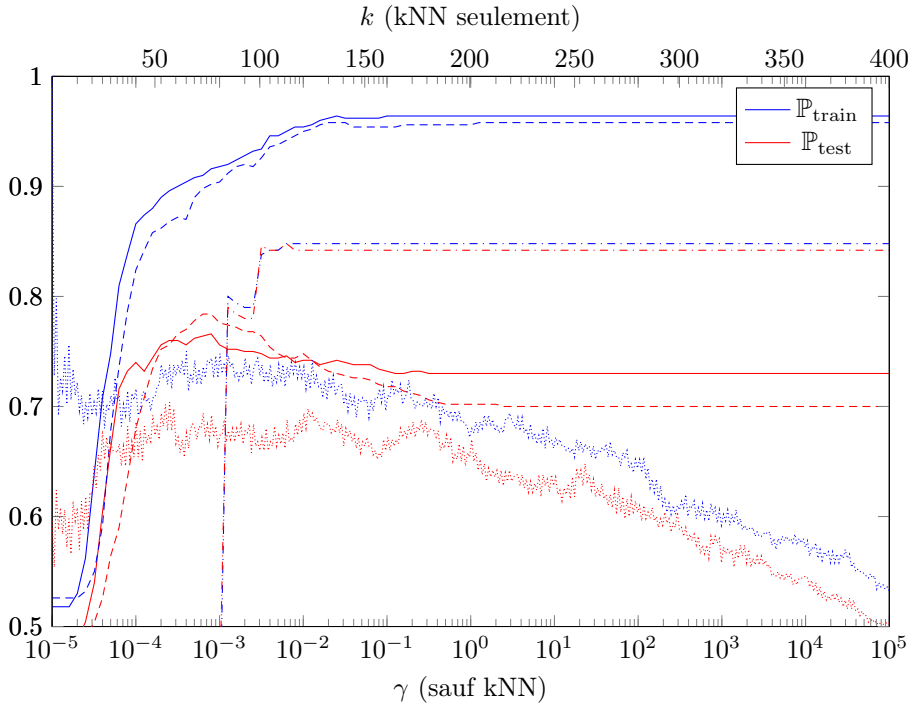

FIGURE 4.5.1. Performance $\mathbb{P}_{\text{train}}$ et $\mathbb{P}_{\text{test}}$ des algorithmes SVM (ligne pointillée), LSSVM (ligne pleine), régression logistique (tirets et points), kNN (points) pour différentes valeurs de γ (régularisation) ou k (nombre de voisins). Ici pour $x_i \sim \mathcal{N}(\pm\mu, I_p)$ i.i.d. avec $\mu = (1, 0, \ldots, 0)^\mathsf{T}$, $p = 250$, $n = 1\,000$ avec $n_{\text{train}} = n_{\text{test}} = n/2$.

LISTING 4.2. Code Matlab de la Figure 4.5.1.

```
n=100;
p=25;

% Génération des données N(+/-mu,I)
mu=[4;zeros(p-1,1)];
y=sign(randn(n,1));
X=randn(p,n)+mu*y';

% Découpage en 'train' et 'test'
ntrain=n/2;
ntest=n-ntrain;
Xtrain=X(:,1:ntrain);
ytrain=y(1:ntrain);
```

```
Xtest=X(:,ntrain+1:end);
ytest=y(ntrain+1:end);

% Design de l'hyperplan LSSVM, SVM et rég. log.

% LSSVM
b=@(gamma)sum(inv(Xtrain'*Xtrain+1/gamma*eye(ntrain))*ytrain)/sum(sum(
    inv(Xtrain'*Xtrain+1/gamma*eye(ntrain))));
w=@(gamma)Xtrain*inv(Xtrain'*Xtrain+1/gamma*eye(ntrain))*(ytrain-b(
    gamma)*ones(ntrain,1));gammas=10.^(-5:1e-1:5);

% SVM
SVM=@(gamma) fitcsvm(Xtrain',ytrain','BoxConstraint',gamma);
kNN=@(N) fitcknn(Xtrain',ytrain','NumNeighbors',N);

% Rég. log.: fminunc résout une minimisation par descente de gradient
    en recevant ici la fonction, son gradient, et le point initial de
    la descente (on aide ici l'algorithme en partant de mu inconnu,
    mais ça ne doit rien changer en théorie)
options = optimoptions('fminunc','Algorithm','trust-region','
    SpecifyObjectiveGradient',true);
w_rl=@(gamma) fminunc( {@(z) 1/2*norm(z)^2+gamma*sum(log(1+exp(-diag(
    ytrain)*Xtrain'*z))),@(z) z-gamma*sum(Xtrain*diag(1+exp(-diag(
    ytrain)*Xtrain'*z)),2)},mu,options);

gammas=10.^(-5:1e-2:5);
Ns = 1:400;

Ptrain_lssvm=zeros(1,length(gammas));
Ptest_lssvm =zeros(1,length(gammas));
Ptrain_svm=zeros(1,length(gammas));
Ptest_svm =zeros(1,length(gammas));
Ptrain_rl=zeros(1,length(gammas));
Ptest_rl =zeros(1,length(gammas));
Ptrain_kNN=zeros(1,length(Ns));
Ptest_kNN =zeros(1,length(Ns));


% Calcul des erreurs
igamma=1;
for gamma=gammas
        Ptrain_lssvm(igamma)=1/ntrain*sum( (Xtrain'*w(gamma)+b(gamma)*
            ones(ntrain,1)).*ytrain > 0 );
        Ptest_lssvm(igamma) =1/ntest *sum( (Xtest' *w(gamma)+b(gamma)*
            ones(ntest ,1)).*ytest  > 0 );
        Ptrain_svm(igamma)  =1/ntrain*sum(predict(SVM(gamma),Xtrain')
            ==ytrain);
        Ptest_svm(igamma)   =1/ntest *sum(predict(SVM(gamma),Xtest')
            ==ytest );
        Ptrain_rl(igamma)   =1/ntrain*sum( (Xtrain'*w_rl(gamma)).*
            ytrain > 0 );
```

```
52          Ptest_rl(igamma)    =1/ntest *sum( (Xtest' *w_rl(gamma)).*
                ytest  > 0 );
53          igamma=igamma+1;
54  end
55
56  iN=1;
57  for N=Ns
58          Ptrain_kNN(iN) =1/ntrain*sum(predict(kNN(N),Xtrain')==ytrain);
59          Ptest_svm(iN)  =1/ntest *sum(predict(kNN(N),Xtest') ==ytest );
60          igamma=iN+1;
61  end
62
63  figure
64  semilogx(gammas,[Ptrain_lssvm;Ptest_lssvm;Ptrain_svm;Ptest_sm;
        Ptrain_rl;Ptest_rl]);
65
66  figure
67  plot(Ns,[Ptrain_kNN;Ptest_kNN]);
```

4.6. Exercices

Exercice 9 (Apprentissage semi-supervisé). À travers l'exemple du LSSVM, nous avons vu dans cette section un moyen simple d'obtenir un algorithme d'apprentissage supervisé en résolvant un problème d'optimisation quadratique. Il existe une manière similaire d'utiliser une optimisation quadratique dans un cadre dit semi-supervisé, à savoir lorsque, parmi les données $\{(x_i, y_i)\}_{i=1}^n$ d'apprentissage, certains y_i ne sont pas connus et qu'il s'agit alors d'identifier ces valeurs. *Il faut bien comprendre qu'on cherche ici, non plus à utiliser seulement les données (x_i, y_i) complètes (avec y_i connu) lors d'une première phase d'apprentissage avant d'appliquer l'algorithme sur chaque x_i dont le y_i associé n'est pas connu, mais plutôt d'utiliser simultanément toutes les données et de recouvrer l'ensemble des y_i (connus et non connus) ; la différence est majeure comme elle permet ici d'utiliser plus d'information (la partie non-supervisée) que dans le cas purement supervisé.* Pour ce faire, une approche très populaire consiste à résoudre le problème d'optimisation suivant :

$$\min_{\hat{y}_1,\dots,\hat{y}_n\in\mathbb{R}} \sum_{i=1}^{n} K_{ij}(\hat{y}_i - \hat{y}_j)^2$$

tel que $\hat{y}_i = y_i$ pour tous les y_i connus et où $K_{ij} \geq 0$ est une fonction d'"affinité" entre x_i et x_j, généralement fonction décroissante $K_{ij} = f(\|x_i - x_j\|)$ de la distance $\|x_i - x_j\|$. Montrer que le problème est équivalent à

$$\min_{\hat{y}\in\mathbb{R}^n} \hat{y}^\mathsf{T}(D - K)\hat{y}$$

où $K = \{K_{ij}\}_{i,j=1}^n$ et D la matrice diagonale définie par

$$D_{ii} = \sum_{j=1}^{n} K_{ij}.$$

Expliquer la logique de cette optimisation : (i) étudier précisément le produit $K_{ij}(\hat{y}_i - \hat{y}_j)^2$ et expliquer, dans le contexte où $y_i \in \{\pm 1\}$, la logique de la minimisation du produit, (ii) pourquoi est-il en particulier fondamental que K_{ij} soit positif ?

Résoudre le problème d'optimisation (en utilisation par exemple la méthode des multiplicateurs de Lagrange ou plus simplement en remplaçant les $\hat{y}_i$ connus

par leurs valeurs y_i). Montrer que la solution s'exprime à nouveau sous la forme d'une méthode de régression linéaire. Écrire la solution sous une forme matricielle "élégante" en définissant la matrice $K = \{K_{ij}\}_{i,j=1}^n$ qu'on segmentera sous la forme

$$K = \begin{bmatrix} K_{[ee]} & K_{[en]} \\ K_{[ne]} & K_{[nn]} \end{bmatrix}$$

où $K_{[ee]}$ contient l'ensemble des K_{ij} pour x_i, x_j **é**tiquetés (y_i, y_j connus), $K_{[nn]}$ l'ensemble des K_{ij} pour x_i, x_j **n**on-étiquetés (y_i, y_j inconnus), etc., ainsi qu'en découpant D sous la forme $D = \mathrm{diag}(D_{[ee]}, D_{[nn]})$ avec des notations évidentes (matrice qu'on appelle communément la matrice des *degrés* du *graphe d'affinité* K). La solution prendra une forme linéaire $\hat{y}_{[n]} = Ay_{[e]}$ où $y_{[e]}$ est l'ensemble des y_i étiquetés et $\hat{y}_{[n]}$ l'estimation des y_i non étiquetés.

Donner une interprétation physique à la formule de régression ainsi obtenue : quelle est l'opération effectuer par la matrice inverse ? quelle est l'opération effectuée par le terme extérieur ?

Il s'avère que l'algorithme décrit ainsi ne fonctionne pas bien lorsque les x_i sont de grandes dimensions p, ceci étant dû à un biais introduit par la méthode. Sans détailler les raisons qui ne sont pas totalement évidentes, il a été démontré qu'une meilleure façon d'opérer est de résoudre plutôt le problème d'optimisation normalisé :

$$\min_{\hat{y}_1,\dots,\hat{y}_n} \sum_{i=1}^n K_{ij}(D_{ii}^{-1}\hat{y}_i - D_{jj}^{-1}\hat{y}_j)^2$$

tel que $\hat{y}_i = y_i$ pour tous les y_i connus.

Résoudre à nouveau ce problème et donner la solution sous forme matricielle du type $\hat{y}_{[n]} = Ay_{[e]}$ comme précédemment. En réécrivant intelligemment cette équation matricielle sous une forme $B\hat{y} = \hat{y}$ avec $\hat{y} = [y_{[e]}^\mathsf{T}, \hat{y}_{[n]}^\mathsf{T}]^\mathsf{T}$, montrer que le problème peut se résoudre numériquement par une approche itérative de "propagation des étiquettes sur le graphe d'affinité K". À savoir, en itérant successivement une certaine matrice sur le vecteur des étiquettes.

Implémenter la méthode en Matlab ou Python et comparer les performances des première et seconde méthodes développées dans l'exercice, pour le mélange gaussien $x_i \sim \mathcal{N}(\pm\mu, I_p)$. Implémenter également l'approche par propagation des étiquettes.

Exercice 10 (Introduction au regroupement spectral (*spectral clustering*)). En reprenant l'Exercice 9, considérons maintenant le cas *non-supervisé* où aucun $y_i \in \{\pm 1\}$ n'est connu parmi les données $(x_1, y_1), \dots, (x_n, y_n)$. On considère donc une configuration dite non supervisée où il s'agit ici, non pas de découvrir les y_i (ce qui est impossible sans aucune donnée), mais de "regrouper" les données $x_1, \dots, x_n$ en un nombre donné de classes distinctes. Pour cela, on résout le même problème que dans l'Exercice 9 mais sans contrainte du type $\hat{y}_i = y_i$ évidemment. Cependant, comme on souhaite $\hat{y}_i \in \{\pm 1\}$, on va a priori imposer cette contrainte, et donc :

$$\min_{\hat{y} \in \{\pm 1\}^n} \hat{y}^\mathsf{T}(D - K)\hat{y}.$$

Ce problème est malheureusement discret et donc combinatoire et trop complexe à résoudre numériquement. L'approche dite du *regroupement spectral* consiste à "relaxer" le problème en imposant seulement que $\|\hat{y}\|^2 = n$. Montrer qu'alors le problème devient équivalent à un problème aux valeurs et vecteurs propres. Quelle en est la solution ? En déduire une méthode d'allocation des classes des x_i.

Implémenter la méthode en Matlab ou Python pour des données x_i issues du mélange gaussien $x_i \sim \mathcal{N}(\pm\mu, I_p)$ (on pourra utiliser intelligemment la *méthode de*

la puissance pour extraire seulement le vecteur propre dominant au lieu de faire une décomposition spectrale complète, très coûteuse).

Exercice 11 (Implémentation manuelle de la régression logistique). Nous avons introduit au cours de ce chapitre une implémentation Matlab (Listing 4.2) de la méthode de régression logistique, que nous rappelons ci-dessous :

```
1 options = optimoptions('fminunc','Algorithm','trust-region','
      SpecifyObjectiveGradient',true);
2 w_rl=@(gamma) fminunc( {@(z) 1/2*norm(z)^2+gamma*sum(log(1+exp(-diag(
      ytrain)*Xtrain'*z))),@(z) z-gamma*sum(Xtrain*diag(1+exp(-diag(
      ytrain)*Xtrain'*z)),2)},mu,options);
```

Il s'agit dans cet exercice dans un premier temps de comprendre cette implémentation (se référer à la documentation Matlab de `fminunc`). Réimplémenter ensuite la méthode de manière exhaustive (en Matlab et Python), en effectuant manuellement la descente de gradient (attention à bien régler le pas de la descente). Tester la méthode sur le mélange gaussien $x_i \sim \mathcal{N}(\pm\mu, I_p)$.

Chapitre 5

Méthodes à noyaux et leur lien avec SVM

Nous avons évoqué dans le chapitre d'introduction le fait que les données $x \in \mathbb{R}^p$ sont soit des données brutes (par exemple une série temporelle, ou pour des images de taille $d_1 \times d_2$ une version vectorisée en dimension $p = d_1 d_2$), soit des représentations $\phi(x)$ pour une certaine fonction $\phi : \mathcal{X} \to \mathbb{R}^p$ (avec $p \in \{1, \dots, \infty\}$, ou même plus généralement $\phi : \mathcal{X} \to \mathcal{H}$ avec $\mathcal{H}$ un espace de Hilbert (muni d'un produit scalaire) : donc pas exemple des fonctions paramétrées par x, ou des valeurs dans un alphabet binaire muni d'opérations logiques. Cette fonction ϕ a pour objectif d'"extraire" des représentations de x (à savoir diverses fonctions de ses entrées) qui, dans l'idéal, sont suffisamment expressives pour permettre des différentier les classes (ou identifier des critères de régression) des données. En particulier, on cherchera à trouver des représentants des données qui les rendent *linéairement séparables* de sorte que, dans le cadre d'une classification à deux classes notamment, un hyperplan puisse diviser l'espace image $\{\phi(x)\}_{x \in \mathcal{X}}$ en deux dans suivant les classes.

Nous allons voir dans ce chapitre des exemples simples de noyaux et allons mettre l'accent sur l'intérêt d'avoir un espace de représentations de taille p aussi grande que possible, quitte même à générer des représentations $\phi_1(x), \phi_2(x), \dots$ complètement au hasard par un mécanisme de *projection aléatoire*. Enfin, nous verrons que, grâce à *l'astuce du noyau*, on pourra parfois prendre p arbitrairement grand (même infini) tout en étant capable de numériquement effectuer les calculs en temps fini : spécifiquement lorsque le produit scalaire $\phi(x)^\mathsf{T}\phi(x')$ peut s'évaluer sous la forme d'une fonction, dite *noyau* $\kappa(x, x') = \phi(x)^\mathsf{T}\phi(x')$ facile à évaluer. On se trouvera alors même dans une situation assez perturbante (car assez abstraite) où on ne choisira plus forcément $\phi(x)$ comme un ensemble de "bons représentants" mais on choisira directement un noyau $\kappa(x, x')$ simple à évaluer et suffisamment "non linéaire".

Les méthodes à noyau, nous le verrons, ont deux avantages fondamentaux : (i) grâce à l'astuce du noyau, elles s'implémentes très naturellement au sein d'un SVM ou d'un LSSVM (mais aussi de nombreuses autres méthodes en apprentissage), ce qui les rend donc très appropriées aux contextes de classification – d'ailleurs, il s'avère les SVMs sont aujourd'hui utilisés en pratique quasi-exclusivement sous leur version *SVM à noyau* et non pas *SVM linéaire* – et (ii) à travers la vision "projection aléatoire", elles créent un pont élégant entre SVMs, régression linéaire, et réseaux de neurones, ce qui nous fera un joli point d'entrée vers le dernier chapitre du cours.

5.1. Notions élémentaires de représentations non linéaires

Nous avons vu dans les chapitres précédents, dans le cadre de la classification binaire, que les méthodes de minimisation du risque empirique régularisée, sous la forme "régressive" que nous avons étudiées, ont toutes pour objectif de construire un vecteur $w \in \mathbb{R}^p$ (éventuellement accompagné d'un terme de biais $b \in \mathbb{R}$), tel que le signe de $x^\mathsf{T}w$ (ou $x^\mathsf{T}w + b$) détermine la classe y de x. Ceci implique qu'on puisse effectivement "diviser l'espace" en deux sous-espaces séparables par un hyperplan.

Il s'agit donc, si $\mathcal{X}$ n'est pas linéairement séparable (ce qui en effet n'est le cas en général), de trouver une fonction ϕ telle que l'image $\phi(\mathcal{X})$ le soit. Prenons

par exemple l'illustration proposée en partie gauche de la Figure 5.1.1 où il s'agit de séparer deux classes de données qui sont des anneaux concentriques dans $\mathbb{R}^2$. Ces données ne sont clairement pas linéairement séparables car il est impossible de passer un hyperplan dans le plan $\mathbb{R}^2$ qui isolerait les deux jeux de données : il s'agit donc ici de trouver une fonction ϕ qui rendent les données séparables. Étant donnée la configuration assez élémentaire ici, une telle fonction ϕ est très vite trouvée : il suffit de prendre $\phi : \mathbb{R}^2 \to \mathbb{R}, x \mapsto \|x\|$; en effet, les deux anneaux concentriques sont composés de points qui diffèrent essentiellement par leur distance à l'origine (petite pour l'anneau rouge, grande pour l'anneau bleu). Ainsi, $\|x\|$ est une représentant suffisante qui permet de classer les données linéairement (il suffit de considérer le test $\|x\| \lessgtr .75$) ; notons d'ailleurs ici $\phi(\mathcal{X})$ est plus petite en dimension que $\mathcal{X}$ lui-même, ce qui ne nous arrivera pas souvent !

Mais imaginons justement que nous n'ayons pas, comme ce sera souvent le cas, une vision aussi claire de situation géométrique des vecteurs de l'espace d'origine $\mathcal{X}$. Comme il est clair que n'importe quelle fonction linéaire ou affine (de type $Ax + b \in \mathbb{R}^q$, même pour q arbitrairement grand) ne peut séparer des vecteurs de $\mathbb{R}^p$ initialement non séparables, on va naturellement chercher des représentants *non-linéaires* des données $x \in \mathcal{X}$. La première idée qui vient en tête est de considérer des polynômes des entrées $x[1], \ldots, x[p]$ de x, à commencer par la réprésentation bi-dimensionelle $\phi(x) = [x[1]^2, x[2]^2]^\intercal$. C'est ce que nous représentons dans la partie droite de la Figure 5.1.1. Évidemment ici, comme $x[1]^2 + x[2]^2 = \|x\|^2$, il existe une combinaison linéaire (et donc un hyperplan séparateur) quasi optimale pour ce jeux de données. Il n'en reste pas moins que, dans un contexte général de données mal identifiées, créer une fonction de représentation $\phi(x)$ à base de polynômes est un choix initial intéressant.

Mais allons un peu plus loin : rappelons nous que, si on peut écrire $y = f(x)$ pour une certaine fonction $f; \mathcal{X} \to \mathcal{Y}$ inconnue et que cette fonction est assez lisse (continue bornée notamment) alors, par le théorème de Weierstass, pour tout compact $C \subset \mathcal{X}$ et tout $\varepsilon > 0$, il existe un $q \in \mathbb{N}$ et un polynôme (ou multinôme si x est multivarié) $Q(x)$ d'ordre q tel que $\sum_{x \in C} |f(x) - Q(x)| < \varepsilon$. Comme $Q(x)$ peut être vu comme une *combinaison linéaire* des monômes $x[1], \ldots, x[p], x[1]^2, x[2]^2, x[1]x[2], \ldots, x[p]^q$ d'ordres inférieurs ou égaux à q, le vecteur $\phi(x) = [x[1], \ldots, x[p], x[1]^2, x[2]^2, x[1]x[2], \ldots, x[p]^q]^\intercal$ projette x dans un espace où les données sont, à ε près, linéairement séparables. Cette remarque nous enseigner deux choses :

- **bases de représentation** : il est utile, idéalement, de choisir un vecteur de représentations des données qui forment une *base* d'un espace assez riche de fonctions, idéalement adapté au jeu de données ;
- **taille et "richesse" du vecteur de représentations** : on se rend bien compte qu'il est pertinent d'avoir un système de représentations très "riche" en ce sens qu'il permet de représenter de nombreuses fonctions par combinaisons linéaires de ses entrées. Malheureusement cela pose très vite un problème : pour l'exemple précédent, le nombre de multinômes élémentaires jusqu'à un ordre q explose très vite avec la dimension initiale de l'espace $\mathcal{X}$ et il sera en pratique totalement inconcevable d'avoir des représentations aussi fines qui impliqueraient que $\phi(x)$ vive dans un espace à milliers ou milliards de dimensions. On devra donc en général se restreindre à des fonctions bien choisies ou, comme nous le verrons, bénéficier de l'*astuce du noyau*.

  Notons par ailleurs que cette idée d'utiliser un espace $\phi(\mathcal{X})$ de grande dimension est très naturel, mais pose un second problème : pour séparer n données d'entrainement $x_1, \ldots, x_n$ comme le font la plupart des algorithmes d'apprentissage, on sait qu'il est suffisant d'utiliser une représentation des points dans un espace de dimension $p > n$. En effet, dans ce cas, il est

toujours possible de trouver un hyperplan séparateur, tout simplement en effectuant une régression linéaire sur les données. Cela semble magique à première vue... mais cache une réalité toute autre : celle d'un cas terrible de surapprentissage ! Comme on peut nécessairement séparer les données, quelqu'elles soient, on n'apprend ici effectivement rien sur ces données si on les sépare par un hyperplan canonique (d'ailleurs si $p \gg n$, il en existe une infinité). Avoir un système de représentation $\phi(x)$ de grande dimension ouvre donc la porte au danger du surapprentissage dont il sera d'autant plus délicat de se prémunir que p est grand. En effet, même en présence d'une base de test, si $p \gg n_{\text{train}} + n_{\text{test}}$, on prend même le risque de surapprendre les données de test. C'est un problème généralement délicat à régler.

Avant de passer à l'implémentation, grâce à l'astuce du noyau, de telles représentations non-linéaires au sein des SVMs, concluons cette section par un l'exemple très populaire en apprentissage des "deux lunes imbriquées" de la partie gauche de la Figure 5.1.2, et qui est un bon exercice pour comprendre comment on peut générer "à la main" un bon jeu de représentations. Dans ce cas, il n'est pas facile de déterminer un bon noyau séparateur. Une première idée est de constater que les deux lunes peuvent être vues comme des morceaux de paraboles, et donc un noyau quadratique semble tout indiqué ; cependant, si on trace des hyperboles portées par chaque lune, on traverse nécessairement l'autre lune, ce qui n'est pas souhaité ! (on les couperait même pile au milieu...) On doit alors envisager des polynômes d'ordre plus élevé : un ordre quatre devrait suffire pour faire passer une courbe avec trois extrema locaux ; pour que cela fonctionne, ici dans $\mathbb{R}^2$, il faut prendre pour assurer de trouver le bon polynôme

$$\phi(x) = [1, x[1], x[2], x[1]x[2], x[1]^2x[2], x[1]x[2]^2, x[1]^2x[2]^3, \dots, x[2]^4]^\mathsf{T}$$

ce qui au total fait un vecteur de dimension $p = 19$. En voyant que l'image par ϕ^{-1} de l'hyperplan séparateur doit passer par $(0, 0)$, on peut se restreindre aux polynômes d'ordres impairs et éliminer à peu près la moitié des termes. Au lieu de cela, on peut également se dire que les deux lunes sont des "bouts" de sinusoïdes décalées et de phases différentes. En imaginant que chacune est une combinaison linéaire particulière de sinusoïdes de fréquences distinctes, on peut alors choisir pour $\phi(x)$ une combinaison de sinusoïdes en les variables $x[1]$ et $x[2]$: c'est ce qui est proposé dans la partie droite de la Figure 5.1.2 où on observe effectivement une séparabilité linéaire des deux lunes par la transformation $\phi(x) = [\sin(x[1]), \sin(3x[2])]^\mathsf{T}$ qui reste en dimension raisonnable $p = 2$. Notons ici que, grâce à la transformation ϕ^{-1} (qui, pour chaque x, est en général un ensemble de points car ϕ n'a aucune raison d'être inversible), on retrouve en Figure 5.1.1 comme en Figure 5.1.2 l'ensemble des points générant l'hyperplan séparateur.

LISTING 5.1. Code Matlab de la Figure 5.1.1

```
n=100;
p=2;
phi=@(z)[z(1,:).^2;z(2,:).^2];

% Génération des données sur des cercles bruités
x1=randn(p,n);
x1=x1*diag(1./sqrt(diag(x1'*x1)))+.05*randn(p,n);
x2=randn(p,n);
x2=.5*x2*diag(1./sqrt(diag(x2'*x2)))+.05*randn(p,n);

% Génération des représentants
z1=phi(x1);
```

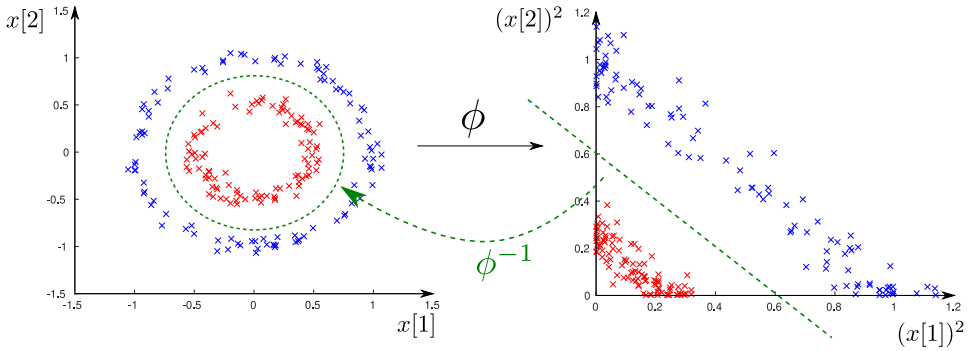

FIGURE 5.1.1. Visualisation de la méthode des noyaux : ici pour $\phi(x) = [x[1]^2, x[2]^2]^\mathsf{T}$. Dans l'espace d'origine (à gauche), les deux classes ne sont pas séparables. Dans l'espace image par ϕ (à droite), les données sont séparables linéairement, par exemple à l'aide d'un SVM. L'hyperplan séparateur a comme pré-image par ϕ^{-1} une courbe qui sépare les deux classes dans l'espace originel.

```
13 z2=phi(x2);
14
15 % Espace originel
16 figure
17 hold on;
18 plot(x1(1,:),x1(2,:),'bx');
19 plot(x2(1,:),x2(2,:),'rx');
20
21 % Espace image
22 figure
23 hold on;
24 plot(z1(1,:),z1(2,:),'bx');
25 plot(z2(1,:),z2(2,:),'rx');
```

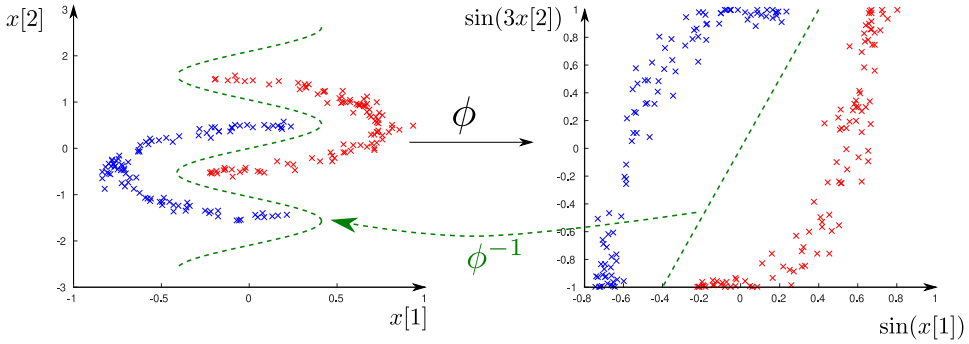

FIGURE 5.1.2. Visualisation de la méthode des noyaux : ici pour $\phi(x) = [\sin(x[1]), \sin(3x[2])]^\mathsf{T}$. Dans l'espace d'origine (à gauche), les deux classes ne sont pas séparables. Dans l'espace image par ϕ (à droite), les données sont séparables linéairement, par exemple à l'aide d'un SVM. L'hyperplan séparateur a comme pré-image par ϕ^{-1} une courbe qui sépare les deux classes dans l'espace originel.

LISTING 5.2. Code Matlab de la Figure 5.1.1

```
n=100;
p=2;
phi=@(z)[sin(z(1,:));sin(3*z(2,:))];

% Génération des demi-lunes bruitées
x1=randn(p,n);
x1=x1*diag(1./sqrt(diag(x1'*x1)))+.05*randn(p,n);
x1(1,:)=-x1(1,:).*sign(x1(1,:));
x1=x1+[.25;-.5]*ones(1,n);

x2=randn(p,n);
x2=x2*diag(1./sqrt(diag(x2'*x2)))+.05*randn(p,n);
x2(1,:)=sign(x2(1,:)).*x2(1,:);
x2=x2+[-.25;.5]*ones(1,n);

% Génération des représentants
z1=phi(x1);
z2=phi(x2);

% Espace originel
figure
hold on;
plot(x1(1,:),x1(2,:),'bx');
plot(x2(1,:),x2(2,:),'rx');

% Espace image
figure
hold on;
plot(z1(1,:),z1(2,:),'bx');
plot(z2(1,:),z2(2,:),'rx');
```

5.2. Projections aléatoires, et le lien avec les réseaux de neurones

Dans la section précédente, nous avons spécifiquement travaillé sur des exemples "jouets", généralement en deux dimensions ($\dim(\mathcal{X}) = 2$ et aussi $\dim(\phi(\mathcal{X})) = 2$). Cependant, les jeux de données modernes (images, sons, graphes, données issues de l'explosion du bigdata) sont généralement de grandes, voire très grandes dimensions.

Si les données ont une structure ou des propriétés d'invariances particulière, il est très pertinent d'utiliser cette information *a priori* pour construire des représentations efficaces. Par exemple, dans le cadre de sons et d'images, on peut utiliser le fait que ces données ont une représentation fréquentielle généralement de petite dimension : on peut alors utiliser les premiers modes de leurs transformées de Fourier. Dans le cadre d'images plus complexes, la transformée de Fourier n'est pas très appropriée (pour différencier un chien d'un chat par exemple), et on utilisera d'autres propriétés des images : soit des transformées plus complexes (telles que des transformées en ondelettes), soit des propriétés angulaires des images (un ballon a des angles dans toutes les directions alors qu'une maison a des angles dans des directions spécifiques, propres à la classe "maisons") comme dans la populaire méthode dite de l'*histogramme des gradients orientés* (en anglais *HOG* pour *histogram of oriented gradients*). Des méthodes plus élaborées, telle que la méthode *SURF* (pour *speed-up robust features*) utilisée en traitement d'images, cherchent

des points caractéristiques de l'image et conserve les paramètres de ces points (en utilisant des filtres par ondelettes).

Ces méthodes ont l'avantage d'être très génériques, applicables systématiquement sur toute base d'images, mais souffrent justement de cette "généricité" qui les rend incapables de répondre à des problèmes plus spécifiques pour lesquelles elles n'auraient pas été créées (comme par exemple différencier des chiens suivant leurs couleurs de poils avec la méthode HOG). Autre inconvénient, ces méthodes manuelles doivent être reproduites et adaptées à chaque nouvelle famille de données, et demandent un travail initial d'ingénierie potentiellement coûteux. En cela, il faut bien comprendre que les approches de type SVM *n'apprennent pas les bonnes représentations mais se contentent des les exploiter.*

5.2.1. Projections aléatoires. Lorsqu'aucune représentation fiable n'est disponible, ou lorsque l'utilisateur n'a pas le temps ou l'énergie à consacrer à la mise en place d'un système de représentation spécifique à la tâche de classification ou régression à effectuer, une autre possibilité est de construire un large échantillon de représentations aléatoires des données, en espérant que parmi ces représentations, certaines (ou une combinaison linéaire appropriée de certaines) de ces représentations seront suffisamment discriminantes. Pour ce faire, on pourra notamment utiliser des *projections aléatoires non linéaires* des données (ou *random features* en anglais). Ces projections aléatoires peuvent être caractérisées sous la forme :

$$\phi(x) \equiv \phi_\omega(x) = \left[\phi_{\omega_1}(x), \dots, \phi_{\omega_p}(x)\right]^\mathsf{T} \in \mathbb{R}^p$$

où $\omega = (\omega_1, \dots, \omega_p)$ sont p variables (ou vecteurs) aléatoires et où $\phi_{\omega_i}(x) = \phi_i(x; \omega_i)$ est une représentation de x via une fonction $\phi_i(\cdot; \omega_i)$ *paramétrée* par la variable aléatoire ω_i. L'avantage fondamental de cette approche est que l'on peut générer un nombre arbitrairement grand de représentations $\phi_{\omega_i}(x)$ en se basant sur une fonction de référence élémentaire $\phi_0(x; \omega_i)$ pour des tirages successifs de ω_i, ainsi formant la représentation $\phi_{\omega_i}(x)$.

L'exemple le plus connu, nous verrons très vite pourquoi, est celui des *projections de Fourier aléatoires* (*random Fourier features* en anglais). Pour $x \in \mathbb{R}^q$, celles-ci sont définies via :

$$\phi_{\omega_i}(x) = \phi_i(x; \omega_i) = \exp\left(\imath x^\mathsf{T} \omega_i\right)$$

où $\omega_i \sim \mathcal{N}(0, I_q)$ sont tirés indépendamment pour chaque i. Notons ici que nous abusons légèrement de la contrainte $\phi(x) \in \mathbb{R}^p$ étant donné que $\phi_i(x; \omega_i)$ est à valeur complexe. Dans beaucoup d'algorithme, cela sera sans conséquence mais pour d'autres (dans les réseaux de neurones aléatoires notamment), il faudra considérer des valeurs réelles, auquel cas on travaillera plutôt avec le vecteur étendu de dimension $2p$

$$\phi_\omega(x)^\mathcal{R} = \left[\cos(\imath x^\mathsf{T} \omega_1), \sin(\imath x^\mathsf{T} \omega_1), \dots, \cos(\imath x^\mathsf{T} \omega_p), \sin(\imath x^\mathsf{T} \omega_p)\right]^\mathsf{T} \in \mathbb{R}^{2p}$$

qui contient la même information "fonctionnelle" que $\phi_\omega(x) \in \mathbb{C}^p$.

Comme précédemment évoqué, l'idée qui se cache derrière les projections aléatoires consiste à se dire qu'en choisissant une famille de fonctions de base $\omega_i \mapsto \phi(x; \omega_i)$ bien paramétrée, on "couvre" bien l'espace des fonctions f qu'on cherche à estimer si ces questions s'approximent bien par des combinaisons linéaires de ces fonctions aléatoires. Évidemment, comme on n'échantillonnera qu'un nombre fini de ω_i, la richesse de ces représentations dépend tout à la fois la fonction $\omega \mapsto \phi_i(x; \omega)$ *et* de la loi de tirage des variables aléatoires ω. Ainsi, dans l'exemple précédent, comme nous le verrons, si au lieu de prendre $\omega_i \sim \mathcal{N}(0, I_q)$, on choisissait une autre loi de tirage, on obtiendra "typiquement" une autre base de fonctions élémentaires.

Ces représentations par projections aléatoires sont assez populaires car elles sont théoriquement compréhensibles et analysent, mais aussi et surtout parce qu'elles sont un moyen simple et peu coûteux de produire un grand nombre de représentations non-linéaires de données "génériques". À savoir, si l'expérimentateur n'a qu'une idée très limitée de la structure interne qui organise ses données, utiliser des méthodes de classification (supervisées ou non) basées sur un grand nombre de représentations aléatoires peut consister en une première tentative d'extraction d'informations non linéaires.

Les projections aléatoires (parfois linéaires dans le contexte que nous allons évoquer) peuvent également servir à extraire une représentation de plus petite dimension des données, et ce dans un objectif de stockage. Par exemple, au lieu de stocker un vecteur de dimension 10 000, on peut le projeter aléatoirement dans un espace de dimension 100 en espérant conserver un maximum d'information utile sur la donnée ; ici toujours bien sûr lorsque l'on n'a que très peu d'a priori sur la structure des données. Il existe même des approches dites de *sketching* des données qui permettent de ne jamais avoir à stocker aucune donnée d'entrainement, mais simplement des représentants qui enrichissent la méthode de classification ou régression en construction "dynamique" (au fil de l'arrivée séquentielle des données). Mais nous sortons du cadre strict du cours ici.

En extrapolant quelque peu sur le dernier chapitre du cours ayant attrait aux réseaux de neurones aléatoires, remarquons que le *perceptron* qui, à une entrée $x \in \mathbb{R}^q$ associe la couche neuronale $\sigma(x^\mathsf{T}\omega_1), \dots, \sigma(x^\mathsf{T}\omega_p)$ de dimension p peut être vue comme une projection non-linéaire qui applique successivement les transformations :

$$x \mapsto Wx = [\omega_1^\mathsf{T} x, \dots, \omega_p^\mathsf{T} x]^\mathsf{T} \mapsto \sigma(Wx) = [\sigma(\omega_1^\mathsf{T} x), \dots, \sigma(\omega_p^\mathsf{T} x)]^\mathsf{T}.$$

En choisissant de prendre les $\omega_i \in \mathbb{R}^q$ aléatoirement, on est en train ici de construire un simple *réseau de neurones aléatoire* à une couche. Effectuer une régression linéaire régularisée (disons par un paramètre $\gamma > 0$) vers une sortie $y \in \mathcal{Y}$ de ce réseau de neurones revient ni plus ni moins qu'à opérer un LSSVM sur les données, mais avec une vision "réseaux de neurones". Ces réseaux de neurones ont connu un regain d'intérêt au début des années 2000 sous le nom (assez pompeux !) de *extreme learning machines*. Nous y reviendrons.

Mais la valeur la plus intéressante des méthodes de projections aléatoires réside dans leur très élégante manière de générer (ou du moins d'approximer asymptotiquement bien, quand leur taille p augmente) des *noyaux* communément utilisés dans la littérature et précisément motivés par l'*astuce du noyau* qui relie représentations et noyaux.

5.3. L'astuce du noyau

Nous l'avons vu à plusieurs reprises dans les sections précédentes, travailler directement sur les données "brutes" ne permet pas en général de séparer ces données linéairement par des hyperplans dans l'espace ambiant (c'est-à-dire l'espace où vivent ces données). Passer par un système de "bons" représentants est l'assurance au contraire d'augmenter nos chances de rendre les données en question séparables. Nous avons vu notamment les exemples des anneaux concentriques et des lunes imbriquées, mais ces exemples se réduisent à des "exemples-jouets" (qu'on connait plus sous le terme anglais de *toy models*) et, en général, trouver des représentants parfaitement adaptés aux données revient presque à résoudre le problème de classification ou régression manuellement à la façon d'un ingénieur ; ce qui nous fait sortir réellement du cadre idéalisé de l'apprentissage "automatique", avec un minimum d'intervention humaine.

On a donc compris qu'il est pertinent de plutôt choisir un système de représentants assez grand en se reposant sur deux idées principales : (i) en créant une grande diversités de représentations fonctionnelles, on "pave" bien l'espace des fonctions qu'on devient capable d'approximer et (ii) il est plus simple de faire passer un hyperplan pour séparer des vecteurs "projetés" dans un espace de grande dimension que dans leur espace ambiant potentiellement de petite dimension (mais attention à la limite triviale du surapprentissage dans ce cas !). Le problème est que ces grands systèmes de représentants (p très grand) vont naturellement induire une explosion des temps de calculs : si on pense aux SVMs, résoudre le problème primal en dimension p peut devenir impossible et on devra passer par le dual ; si on pense aux LS-SVM, la solution finale est une régression linéaire qui impose de calculer la matrice $\Phi^\mathsf{T}\Phi \in \mathbb{R}^{n\times n}$ avec $\Phi = [\phi(x_1), \dots, \phi(x_n)] \in \mathbb{R}^{p\times n}$, ce qui a un coût de l'ordre de $O(n^2 p)$ et, pour chaque nouvelle donnée x, on devra évaluer $\phi(x)^\mathsf{T}\Phi$, ce qui coutera $O(np)$ opérations à nouveau. Si p vient à grandir, ces coûts peuvent devenir intolérables.

5.3.1. Idée générale. C'est à ce niveau que l'astuce du noyau intervient. En effet, si on reprend spécifiquement les expressions des phases d'apprentissage et de test des SVMs et LSSVMs, on s'aperçoit que tous les calculs impliquant les données d'entrainement $\phi(x)_1, \dots, \phi(x)_n$ ou les nouvelles données $\phi(x)$ peuvent, à quelque manipulation parfois nécessairement, peuvent se résumer au calcul des produits scalaires $\{\phi(x_i)^\mathsf{T}\phi(x_j)\}_{i,j=1}^n$ et $\{\phi(x)^\mathsf{T}\phi(x_i)\}_{i=1}^n$. En effet, prenons l'exemple de l'algorithme SVM : pour $\phi(x) \in \mathbb{R}^p$ de classe y inconnue, il s'agit d'évaluer $g(x) = (w^\star)^\mathsf{T}\phi(x) = \sum_i \alpha_i^\star y_i \phi(x_i)^\mathsf{T}\phi(x) + b$ qui dépend donc des produits scalaires $\phi(x)^\mathsf{T}\phi(x_i)$ et des $\alpha_i^\star$; quant aux $\alpha_i^\star$, ils sont l'unique solution du problème d'optimisation $\sup_{\alpha\geq 0} -\frac{1}{2}\alpha^\mathsf{T} D_y \Phi^\mathsf{T}\Phi D_y \alpha + \alpha^\mathsf{T} 1_n$ où on rappelle que $\Phi = [\phi(x)_1, \dots, \phi(x)_n]$ et $D_y = \mathrm{diag}(y_1, \dots, y_n)$: il s'agit donc, pour ce qui est des données d'entrainement, d'évaluer $\Phi^\mathsf{T}\Phi$ dont l'entrée (i,j) vaut $\phi(x_i)^\mathsf{T}\phi(x_j)$; s'agissant du biais b, il s'obtient à partir des cas d'égalité des vecteurs support impliquant les produits scalaires $w^\mathsf{T}\phi(x_i)$ et donc à nouveau seulement les produits scalaires $\phi(x_i)^\mathsf{T}\phi(x_j)$. De la même manière, s'agissant de la méthode LSSVM, les multiplicateurs de Lagrange α s'expriment maintenant sous la forme explicite $\alpha = (\Phi^\mathsf{T}\Phi + \gamma^{-1} I_n)^{-1}(y - b1_n)$ où b dépend aussi des données uniquement à travers la matrice $(\Phi^\mathsf{T}\Phi + \gamma^{-1} I_n)^{-1}$.

Dès lors, pour résoudre ces problèmes, il ne s'agit que de calculer les termes $\{\phi(x_i)^\mathsf{T}\phi(x_j)\}_{i,j=1}^n$ et $\{\phi(x)^\mathsf{T}\phi(x_i)\}_{i=1}^n$. Chaque produit scalaire a en théorie un coût calculatoire de p opérations produit, avec p potentiellement très grand, voire même infini, en plus des calculs explicites (s'ils ne sont pas stockés) des $\phi(x_i)$. Cependant, il est possible que la fonction

$$\begin{aligned} \kappa : \mathcal{X} \times \mathcal{X} &\to \mathbb{R} \\ x, y &\mapsto \kappa(x,y) = \phi(x)^\mathsf{T}\phi(y) \end{aligned}$$

puisse prendre une forme simple, permettant notamment de ne pas avoir à calculer le produit scalaire explicitement.

Commençons par un exemple très simple (et volontairement construit à la main) : pour $x = [x[1], x[2]]^\mathsf{T} \in \mathbb{R}^2$, soit

$$\phi(x) = [x[1]^2, x[2]^2, \sqrt{2}x[1]x[2]]^\mathsf{T}$$

alors

$$\phi(x)^\mathsf{T}\phi(y) = x[1]^2 y[1]^2 + x[2]^2 y[2]^2 + 2x[1]y[1]x[2]y[2] = (x^\mathsf{T} y)^2.$$

En définissant $k(x,y) = (x^\mathsf{T}y)^2$, on parvient donc à évaluer $\phi(x)^\mathsf{T}\phi(y)$ à l'aide du produit scalaire $x^\mathsf{T}y$, soit deux opérations multiplicatives et un carré, au lieu des douze produits impliqués dans le calcul complet de $\phi(x)^\mathsf{T}\phi(y)$. On gagne donc ici

déjà un facteur 6 sur cet exemple somme-toute trivial. Cette réduction calculatoire issue d'un calcul simplifié du produit scalaire de l'espace $\phi(\mathcal{X})$ en une opération élémentaire dans l'espace $\mathcal{X}$ est ce qu'on appelle *l'astuce du noyau*.

Avant d'aller plus loin et de démontrer la profondeur théorique qui se cache derrière cet exemple simple, prenons un autre exemple, bien plus marquant, et qui vraisemblablement a lancé la gloire de l'astuce du noyau à l'époque (à savoir en 2008 avec un article légendaire dû à Rahimi et Recht). Supposons que $\mathcal{X} = \mathbb{R}^q$ et considérons la représentation aléatoire, déjà vue plus haut

$$\phi_\omega(x) = \frac{1}{\sqrt{p}} \left[\exp(\imath x^\mathsf{T}\omega_1), \ldots, \exp(\imath x^\mathsf{T}\omega_p)\right]^\mathsf{T} \in \mathbb{R}^p$$

où $\omega_1, \ldots, \omega_p \in \mathbb{R}^q$ est une suite de vecteurs aléatoires indépendants tirés de la mesure $\mathcal{N}(0, I_q)$. À savoir : on considère une projection de Fourier aléatoire (*random Fourier feature*) de longueur p. Alors, le produit scalaire $\phi_\omega(x)^\mathsf{T}\phi_\omega(y)$ s'écrit

$$\phi_\omega(x)^\mathsf{T}\phi_\omega(y) = \frac{1}{p}\sum_{i=1}^p \exp(\imath x^\mathsf{T}\omega_i)\exp(\imath y^\mathsf{T}\omega_i) = \frac{1}{p}\sum_{i=1}^p \exp(\imath (x+y)^\mathsf{T}\omega_i).$$

Prenons maintenant p arbitrairement grand : alors, du fait de la loi forte des grands nombres, on obtient, avec probabilité 1 que :

$$\phi_\omega(x)^\mathsf{T}\phi_\omega(y) \underset{p\to\infty}{\longrightarrow} \mathbb{E}_{\omega_1}\left[\exp(\imath (x+y)^\mathsf{T}\omega_1)\right].$$

Or, si on se rappelle de la fonction caractéristique $\varphi(u) \equiv \mathbb{E}_z[\exp(\imath z^\mathsf{T}u)] = \exp(-\frac{1}{2}\|z\|^2)$ d'un vecteur aléatoire gaussien $z \sim \mathcal{N}(0, I_q)$, on obtient finalement

$$\phi_\omega(x)^\mathsf{T}\phi_\omega(y) \underset{p\to\infty}{\longrightarrow} \kappa(x,y) = \exp\left(-\frac{1}{2}\|x+y\|^2\right)$$

à savoir précisément (à un scalaire $(2\pi)^{-q/2}$ près) la fonction de densité d'un vecteur gaussien.

Cet exemple est extrêmement précieux, car il faut bien comprendre ce qu'il se passe ici : la base "de Fourier" aléatoire des fonctions $\phi_{\omega_i} : x \mapsto \exp(\imath x^\mathsf{T}\omega_i)$ forme une base de fonctions génératrices de fonctions continues bornées sur des compacts de $\mathbb{R}^q$ (si $\mathcal{X} = a\mathbb{R}^q$) et, plus le nombre p de tirages de ω_i est grand, plus "riches" sont les fonctions qu'il devient possible d'approximer. Ainsi, en prenant $p \to \infty$, on gagne en résolution mais on perd en capacité de calcul des produits scalaires $\phi_\omega(x)^\mathsf{T}\phi_\omega(y)$ avec $\omega = (\omega_1, \ldots, \omega_p)$. Ainsi, grâce au fait que $\phi_\omega(x)^\mathsf{T}\phi_\omega(y) \to_{p\to\infty} \exp(-\frac{1}{2}\|x+y\|^2)$, le calcul coûteux du produit scalaire avec p très grand se réduit à un calcul de la norme $\|x+y\|^2 = (x+y)^\mathsf{T}(x+y)$ de taille q potentiellement petit.

Retranscrite en terme d'algorithmes d'apprentissage (classification ou régression) au sein desquels elle peut être appliquée, tels que SVM ou LSSVM, l'astuce du noyau permet d'effectuer à faible coût une régression linéaire sur un spectre potentiellement large, voire même infini !, de représentations non linéaires des données. Notamment, l'hyperplan séparateur $w \in \mathbb{R}^p$ des SVMs, utilisé dans la décision $g(x) = w^\mathsf{T}\phi(x) + b \gtrless 0$, est donc un hyperplan dans un espace de dimension potentiellement infinie mais qui, du fait de l'astuce du noyau, n'apparait plus explicitement dans la décision, qui prend désormais la forme

$$g(x) = \sum_{i=1}^n \alpha_i y_i \phi(x_i)^\mathsf{T}\phi(x) = \sum_{i=1}^n \alpha_i y_i k(x_i, x) = \alpha^\mathsf{T} D_y k(\cdot, x)$$

où on a défini $k(\cdot, x) = [k(x_1, x), \ldots, k(x_n, x)]^\mathsf{T} \in \mathbb{R}^n$.

En termes de coût de calcul, on a donc successivement opéré les transformations suivantes :

(1) en regardant les SVMs (ou LSSVMs) sous leur forme *linéaire*, à savoir en travaillant directement sur l'espace $\mathcal{X} = \mathbb{R}^q$ des données brutes (*raw data* en anglais), il est apparu assez naturel, lorsque q est assez petit, de les considérer sous leur forme *primale* géométrique en calculant les paramètres $(w, v) \in \mathbb{R}^q$ de l'hyperplan séparateur, de sorte à évaluer chaque nouveau point $x \in \mathcal{X}$ via la décision $g(x) = w^\mathsf{T} x + b$; l'opération d'évaluation coûte ici $O(q)$ produits, limitant ainsi le coût de calcul à la plus petite des dimensions ($n \gg q$ en général) du problème ;

(2) en comprenant que les données brutes sont rarement séparables dans leur espace ambiant, on souhaite alors passer par une représentation plus riche $\phi(x) \in \mathbb{R}^p$ avec p idéalement assez grand ; mais ceci fait passer le coût d'évaluation $g(x)$ sous forme primale de $O(q)$ à $O(p)$ produits, ce qui peut devenir coûteux, et même très coûteux si $p \to \infty$, de sorte qu'on se trouve maintenant dans une situation où le coût est associé à la taille la plus grande du problème ($p \gg n \gg q$ en général) ;

(3) en passant au dual, et en exploitant surtout l'astuce du noyau, on parvient alors à évaluer $g(x)$ (ainsi que tous les calculs de la phase d'apprentissage) permettant de générer la fonction g) pour un coût $O(n) \ll O(p)$.

Ces promesses de gain en temps de calcul sont très encourageantes. Cependant, l'astuce du noyau, comme nous l'avons présentée ici, n'a été présentée qu'à travers deux exemples finement choisis, et rien ne dit que cette idée se généralise à des choix arbitraires de fonctions de représentations $\phi(x)$. À savoir, pour une fonction $\phi : \mathcal{X} \to \mathbb{R}^p$ donnée, existe-il un noyau $k(\cdot, \cdot) : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ "pertinent" tel que $\phi(x)^\mathsf{T}\phi(y) = k(x, y)$?

La section suivante apporte un élément important de réponse.

5.3.2. Résultats théoriques. La question précédente est difficile à formuler mathématiquement en ce sens qu'il faudrait définir ce qu'est un noyau pertinent. De fait, en définissant formellement $k(x, y) = \phi(x)^\mathsf{T}\phi(y)$, on forme une fonction à noyau bien définie. Mais cette fonction n'admet pas forcément une représentation peu coûteuse en temps de calcul.

Par contre, en retournant la question initiale, une question mieux formée et pratiquement plus utile est la suivante : étant donné un noyau $k(\cdot, \cdot)$ arbitraire, existe-il une représentation associée $\phi(\cdot)$ telle que $\phi(x)^\mathsf{T}\phi(y) = k(x, y)$? Cette question est utile en ce sens que, dans l'écriture duale des SVMs ou LSSVMs, on peut dans tous les cas "choisir gratuitement" la fonction $k(\cdot, \cdot)$ et donc induire une nouvelle forme de classification avec fonction de décision $g(x) = \alpha^\mathsf{T} k(\cdot, x)$ où α est calculé à partir de la matrice $K = \{k(x_i, x_j)\}_{i,j=1}^n$. Cependant, lorsque l'on opère ainsi, comment alors garantir l'existence d'une fonction de représentation ϕ associée telle que $g(x) = w^\mathsf{T}\phi(x) + b$ où (w, b) est obtenu à partir des $\phi(x_i)$? À savoir : comment garantir que l'on résout vraiment un SVM (ou LSSVM) et pas un "problème artificiel" dénué de sens ?

La réponse à cette question est très optimiste et due à Mercer, à travers le très populaire Théorème 2 que nous avons déjà présenté mais qu'il est bon de replacer ici dans son contexte :

THÉORÈME 10 (Noyaux de Mercer). *Toute fonction $k(z, z') : \mathbb{R}^q \times \mathbb{R}^q \to R$ continue, symétrique (à savoir telle que $k(z, z') = k(z', z)$) et semi-définie positive, à savoir telle que pour tout n et pour toute famille $z_1, \ldots, z_n$, la matrice*

$$K = \{k(z_i, z_j)\}_{1 \leq i,j \leq n}$$

est définie positive, alors il existe une fonction $\phi : \mathbb{R}^q \to H$ pour un certain espace pré-hilbertien H (un espace vectoriel muni d'un produit scalaire, possiblement de

dimension infinie), telle que[1]

$$k(z, z') = \phi(z)^\mathsf{T}\phi(z').$$

En d'autres termes, le théorème mentionne qu'étant donnée une fonction $k(\cdot,\cdot)$, si toute matrice $K = \{\kappa(x_i, x_j)\}_{i,j=1}^n$ construite à partir de n'importe quel n et n'importe quel jeu de données $x_1, \ldots, x_n \in \mathbb{R}^q$ est nécessairement (semi-)définie positive, alors il existe une fonction de représentation $\phi(\cdot)$ associée à $k(\cdot,\cdot)$.

Le mécanisme d'utilisation de l'astuce du noyau fonctionne donc effectivement "à l'envers" : on se donne un *noyau défini positif* (c'est-à-dire satisfaisant la condition précédente) et on sait alors qu'il existe un ϕ associé. Cependant, l'approche n'est pas constructive et il n'est pas forcément aisé d'identifier le représentant ϕ associé au noyau k. De plus, il n'est pas forcément aisé de trouver, ou du moins de vérifier, le caractère défini positif des noyaux. Le résultat de Rahimi et Recht concernant les *projections de Fourier aléatoires* permet en fait de répondre aux deux questions pour une sous-classe de noyaux k définis positifs. Précisément, Rahimi et Recht nous rappellent en premier lieu le résultat suivant, dû à Bochner : þ[Noyau invariants par translations] Tout noyau $k(\cdot,\cdot)$ vérifiant la propriété $k(x, y) = f(x - y)$ pour une certaine fonction continue f est défini positif si et seulement si f est la transformée de Fourier d'une mesure à valeur dans $\mathbb{R}^+$. Pour exploiter ce résultat de Bochner, il suffit de prendre une mesure μ supportée par $\mathcal{X}$ et telle que, pour tout ensemble mesurable A, $\mu(A) \geq 0$ (par exemple une mesure de probabilité) ; on construit alors $f(u) = \int e^{-2\pi\imath u^\mathsf{T} z}\mu(dz)$, ce qui donne lieu à un noyau $k(\cdot,\cdot)$ défini par $k(x, y) = f(x - y)$ et qu'on sait alors être défini positif. Mais, à nouveau, cela n'identifie pas la fonction de représentation ϕ sous-jacente. Pour cela, Rahimi et Recht remarquent alors qu'on peut exploiter le théorème de Bochner comme suit : donnons nous une mesure positive μ et observons que

$$\begin{aligned} k(x, y) = f(x - y) &= \int e^{-2\pi\imath z^\mathsf{T}(x-y)}\mu(dz) = \mathbb{E}_\mu\left[e^{-2\pi\imath z^\mathsf{T}(x-y)}\right] \\ &= \mathbb{E}_\mu\left[e^{-2\pi\imath z^\mathsf{T} x}\left(e^{-2\pi\imath z^\mathsf{T} y}\right)^*\right] \end{aligned}$$

avec $(\cdot)^*$ le complexe conjugué. Alors, en définissant

$$\phi_{\omega_i}(x) = e^{-2\pi\imath\omega_i^\mathsf{T} x}$$

pour $\omega_i \sim \mu$, on obtient $k(x, y) = \mathbb{E}[\phi_{\omega_i}(x)\phi_{\omega_i}(y)]$ de sorte que, en exploitant la loi des grands nombres pour $\phi_1, \phi_2, \ldots$ i.i.d. de loi μ, on a finalement

$$k(x, y) = \sum_{i=1}^{\infty} \phi_{\omega_i}(x)\phi_{\omega_i}(y) = \phi_\omega(x)^\mathsf{T}\phi_\omega(y)$$

où $\phi_\omega(x) = [\phi_{\omega_1}(x), \phi_{\omega_2}(x), \ldots]^\mathsf{T}$. Nous parvenons donc ainsi à définir toute une famille de pairs représentations $\phi(\cdot)$ et noyaux $k(\cdot)$ associés, paramétrées par le choix de la mesure μ.

Des noyaux définis par cette relation, le plus populaire a déjà été défini. Il s'agit du noyau dit *gaussien*, aussi appelé *noyau de la chaleur* pour ses liens avec la physique, ou aussi parfois de manière exagérée *noyau radial* (*radial basis function* en anglais) du fait de son écriture sous la forme $k(x, y) = f(\|x - y\|)$.[2] Ce noyau est défini sur $\mathcal{X} = \mathbb{R}^q$ par

$$\mu(dz) = \frac{1}{(2\pi)^{q/2}} e^{-\frac{1}{2}\|z\|^2} dz \quad \text{et} \quad k(z, z') = e^{-\frac{1}{2}\|z-z'\|^2}.$$

1. On abuse ici de la notation $(\cdot)^\mathsf{T}(\cdot)$ pour définir le produit scalaire, même en dimension infinie.

2. Nous indiquons ici que c'est exagéré car ce n'est pas le seul noyau ayant cette propriété.

D'autres pairs sont rapportées par Rahimi et Recht. Par exemple

$$\text{(Noyau laplacien) : } \mu(dz) = \prod_{i=1}^{q} \frac{1}{\pi(1+z_i^2)} dz \quad \text{et} \quad k(z,z') = e^{-\sum_{i=1}^{q} |z_i - z_i'|}$$

$$\text{(Noyau de Cauchy) : } \mu(dz) = e^{-\sum_{i=1}^{q} |z_i|} \quad \text{et} \quad k(z,z') = \prod_{i=1}^{q} \frac{2}{1+|z_i - z_i'|^2} dz.$$

Comme leur nom l'indique les noyaux invariants par translation ont la propriété de vérifier $k(x+z, y+z) = k(x,y)$, ce qui impose des contraintes spécifiques sur la fonction de discrimination des données. D'autres résultats similaires, pour d'autres classes de noyaux (par exemple des noyaux de type $k(z,z') = f(z^{\mathsf{T}} z)$ ou invariants par rotation), ont aussi été étudiés mais ont généralement des formes moins élégantes que les noyaux invariants par translation.

Nous complétons cette section par une considération un peu plus abstraite des méthodes à noyaux et leur extension au-delà des noyaux définis positifs.

Remarque 11 (Au delà des noyaux de Mercer). Nous l'avons vu, lorsqu'il s'agit de prendre une décision par la fonction

$$g(x) = w^{\mathrm{T}} k(x, \cdot) + b$$

où (w,b) sont fonctions de la matrice à noyau $K = \{k(x_i,x_j)\}_{i,j=1}^n$, $g(x)$ n'a de sens vis-à-vis du problème originel de classification que lorsque $k(\cdot,\cdot)$ est semi-défini positif. Si $k(\cdot,\cdot)$ est un noyau arbitraire qui n'est pas semi-défini positif, il est en fait assez évident qu'aucune fonction ϕ n'existe qui serait telle que $k(x,y) = \phi(x)^{\mathsf{T}}\phi(y)$, et alors le problème d'optimisation primal n'est plus défini : on perd d'emblée la vision géométrique des SVMs ainsi que le problème d'optimisation.

Néanmoins, force est de constater que la fonction $g(x)$ reste elle bien définie pour n'importe quel noyau $k(\cdot,\cdot)$! On peut donc imaginer une classe de classifieurs $g(x)$ formellement définis par $g(x) = w^{\mathrm{T}} k(x,\cdot) + b$ avec la même définition de (w,b) mais pour $k(\cdot,\cdot)$ totalement arbitraire. Par exemple $k(x,y) = A(x^{\mathsf{T}}y)^2 + B(x^{\mathsf{T}}y) + C$ pour des constantes A, B, C arbitraires. Mais dans ce cas, on ne sait plus bien ce qu'on est en train de résoudre et rien n'indique (bien au contraire !) que ces méthodes aient une quelconque raison de fonctionner.

En fait, de manière très surprenante, il s'avère, comme le démontrent certains résultats récents, que certains de ces noyaux peuvent s'avérer extrêmement puissants, et que même certains d'entre eux (qui ne sont donc pas définis positifs) produisent des performances plus élevées que les performances atteintes par le "meilleur" des noyaux définis positifs. La vision moderne, notamment impulsée par la théorie des matrices aléatoires, lorsqu'il s'agit d'étudier les performances de ces algorithmes pour des *données de grandes dimensions*, semble en effet aller dans le sens d'une amélioration notable des performances accessibles par cette classe bien plus libre de noyaux $k(\cdot,\cdot)$ même si, à nouveau, on perd de fait la vision géométrique tout à la fois que la vision optimisation.

Pour comprendre ces autres noyaux discutés dans la remarque, une vision nouvelle des problèmes doit être mise en place. C'est ce que propose la théorie des matrices aléatoires que nous allons brièvement évoquer dans la dernière section de ce chapitre.

5.4. Analyse moderne des algorithmes d'apprentissage par noyaux

Pour conclure cette section, nous effectuons une petite digression sur l'état actuel de la recherche en termes d'analyse et d'amélioration des méthodes d'apprentissage basées sur les approches par noyaux.

Un problème central avec les approches à noyaux est que, aussi élégantes soient-elles, l'analyse théorique de leurs performances demeure délicate. En effet, même dans le cadre du LSSVM dont l'expression est complètement explicite, analyser les performances d'entrainement par exemple, revient à pouvoir évaluer les statistiques de la quantité

$$\begin{aligned} E_{\text{train}}(g) &\equiv \frac{1}{n_{\text{train}}}(y - b1_{n_{\text{train}}})^{\mathsf{T}}(I_{n_{\text{train}}} + \gamma\Phi^{\mathsf{T}}\Phi)^{-2}(y - b1_{n_{\text{train}}}) \\ &= \frac{1}{n_{\text{train}}}(y - b1_{n_{\text{train}}})^{\mathsf{T}}(I_{n_{\text{train}}} + \gamma K)^{-2}(y - b1_{n_{\text{train}}}) \end{aligned}$$

où on rappelle que $K = \{k(x_i, x_j)\}_{i,j=1}^{n} \in \mathbb{R}^{n\times n}$. Il faut bien voir ici que, même si on suppose que les x_i sont tirés de la mesure la plus élémentaire, par exemple $x_i \sim \mathcal{N}(\pm\mu, I_q)$, plus problèmes rendent l'estimation de $E_{\text{train}}(g)$ délicate : (i) le caractère non-linéaire de $k(\cdot,\cdot)$ et en particulier les dépendances non triviales entre les entrées de la matrice K, ainsi que (ii) la forme $(I+\gamma K)^{-2}$ qui implique l'analyse de l'inverse d'une grande matrice.

Des travaux récents permettent pourtant de passer outre ces deux difficultés, et notamment le puissant outil de la *théorie des matrices aléatoires*. Cet outil, qui a la base a pour objectif d'analyser les propriétés spectrales (distributions des valeurs propres notamment) de grandes matrices à entrées indépendantes, s'est récemment enrichi et permet d'étudier des modèles matriciels dont les entrées ont des structures de dépendances assez complexes. La clé de voûte de la théorie des matrices aléatoires est l'exploitation de phénomènes de *concentration* ou de *convergence* lorsque les dimensions de la matrice grandissent ($n, p \to \infty$ pour nous ici) : on peut voir ces phénomènes comme des lois des grands nombres mais pour des mesures de fonctionnelles de matrices au lieu de moyennes de variables aléatoires (par exemple la trace d'une matrice aléatoire, la trace de son carré, le logarithme de son déterminant, etc.). Assez étonnamment, un deuxième outil propre à la théorie des matrices aléatoires (en fait plus formellement issu de la théorie des opérateurs dans les espaces de Hilbert) nous rend la vie très facile ici : la *résolvante* et la *tranformée de Stieltjes* ; précisément, pour étudier le spectre de grandes matrices Z quand leurs dimensions grandissent, la théorie des matrices aléatoires passe par leur résolvante $Q(z) = (Z - zI)^{-1}$ ou par sa transformée de Stieltjes $m(z) = \text{tr}(Z - zI)^{-1}$. Ces formes d'inverses régularisées de matrices $(Z - zI)^{-1}$ sont exactement celles qui apparaissent ici dans l'expression de $E_{\text{train}}(g)$; plus précisément, il s'agit du carré $(Z - zI)^{-2}$ mais qu'on peut aussi écrire comme $(d/dz)(Z - zI)^{-1}$. Il est donc suffisant d'étudier ces résolvantes, ce que la théorie des matrices aléatoires sait très bien faire, pour évaluer $E_{\text{train}}(g)$ pour différents modèles statistiques de données X et de noyaux K construits à partir de X.

Et ces analyses sont en fait extrêmement surprenantes ! Un des résultats centraux, en plus de la capacité des matrices aléatoires à proprement rendre compte des performances asymptotiques exactes attendues, est un résultat dit d'*universalité*. Précisément, la théorie prédit que, pour une large gamme de statistiques des données $X = [x_1, \ldots, x_n]$, les performances asymptotiques $E_{\text{train}}(g)$ et $E_{\text{test}}(g)$ *ne dépendent que des moments d'ordre un et deux des* x_i. Que faut-il y comprendre ? Il faut comprendre les choses ainsi : que les x_i soient des données statistiquement complexes *ou bien* de simples gaussiennes ayant les mêmes moyennes et covariances que le modèle complexe, les performances des SVM, LSSVM, régression logistiques, etc., seront *exactement les mêmes* dès lors que les dimensions des données sont assez grandes. Ce résultat très surprenant, qu'on dit *universel* (du fait de son indépendance de la distribution des données), n'est clairement pas vrai en petite dimension et notre

intuition ne s'y trompe pas ; mais en grandes dimensions, souvent, tout se passe *comme si* les données étaient de simples vecteurs gaussiens.

Plus spécifiquement le résultat d'universalité a lieu dès que le modèle statistique des données est assez "lisse" et plus précisément, lorsque les données peuvent être modélisées comme des vecteurs dits *concentrés* : à savoir des vecteurs satisfaisant un phénomène de *concentration de la mesure*. Formellement, un vecteur $z \in \mathcal{R}^p$ est concentré si, pour toute fonctionnelle 1-Lipschitz $h : \mathbb{R}^p \to \mathbb{R}$:

$$\mathbb{P}\left(|h(z) - \mathbb{E}[h(z)]| > \varepsilon\right) < Ce^{-c\varepsilon^2}$$

pour certaines constantes universelles $C, c > 0$. À savoir : toute *observation scalaire lisse* (1-Lipschitz) du grand vecteur z est proche de son espérance. Quand on y pense, c'est une requête très forte car elle doit être vraie pour toute fonction h ! Et, effectivement, il est difficile de trouver des distributions de vecteurs z qui satisfassent cette forte requête. Mais ces distributions existent et sont en fait très riches !

Premièrement, il peut être démontré par des arguments géométriques que $z \sim \mathcal{N}(0, I_p)$ est concentré. A priori, cela ne nous mène pas bien loin. Mais ce qui est fort, c'est que, pour toute transformation 1-Lipschitz $F : \mathbb{R}^p \to \mathbb{R}^q$ pour p, q arbitraire, il est aisé de vérifier (par la définition ci-dessus) que, si z est concentré alors $F(z)$ est aussi concentré avec les mêmes C, c. Ainsi, on trouve immédiatement que toute transformation Lipschitz d'un vecteur gaussien $\mathcal{N}(0, I_p)$ est concentré par définition. Et cela constitue une famille bien plus riche qu'on ne l'imagine !

En effet, récemment, une petite révolution a eu lieu au sein de l'apprentissage profond avec l'émergence des *réseaux génératifs antagonistes* (plus connus sous le nom anglais de GANs : *generative adervsarial networks*). Ces réseaux de neurones profonds fonctionnent de la manière suivante, bien représentée en Figure 5.4.1 : (i) un premier réseau, appelé générateur (à gauche sur le dessin), prend en entrée des vecteurs gaussiens $\mathcal{N}(0, I_p)$ aléatoires et les transforme de manière non-linéaire en une sortie "image générée" qui, avant l'apprentissage lorsque le réseau n'est pas entrainé, ne correspond pas à grand chose ; (ii) pendant ce temps, un second réseau, appelé discriminateur (à droite sur le dessin), a pour but de discriminer un ensemble d'"images réelles" des "images générées". La fonction objectif du réseau générateur est de "tromper" le réseau discriminateur, tandis que le réseau discriminateur a pour objectif d'améliorer sa capacité à résoudre images réelles versus générées : les réseaux sont donc "antagonistes", le premier améliorant ses performances lorsque les performances du second diminuent et vice-versa. Au final, et par une magie vraiment propre aux réseaux profonds (personne ne sait honnêtement dire théoriquement pourquoi ces réseaux fonctionnent !), le *réseau générateur produit des images quasi-réelles à partir de bruit blanc* $\mathcal{N}(0, I_p)$ capables de leurrer le réseau discrimateur. Mais prenons un peu de recul : une fois le réseau générateur entraîné, qu'a-t-on obtenu ? Tout simplement une fonction F qui n'est autre que l'enchainement de couches linéaires et de fonctions d'activations non-linéaires généralement 1-Lipschitz (sigmoïdes, rampes, etc.), et donc par définition *une fonction Lipschitz* de $\mathbb{R}^p$ dans $\mathbb{R}^q$, avec q la taille des images générées. Les images générées sont donc par définition des vecteurs concentrés de loi $F(\mathcal{N}(0, I_p))$! Le résultat d'universalité des matrices aléatoires s'applique donc et on garantit ainsi que les performances d'un SVM, LSSVM, régression logistique, etc., appliqués à des images GANs, sont théoriquement les mêmes que celles obtenues par les mêmes algorithmes appliqués à des vecteurs gaussiens de même moyenne et covariance ! De là, il vient assez naturellement que les vraies images, qui ressemblent tant à des images générées par des GANs qu'un réseau de neurones profond très bien entraîné s'y trompe, les performances des algorithmes de classification ou régression se comporteront de même, comme si les données étaient de simples vecteurs gaussiens !

Ce constat a des répercussions théoriques et surtout pratiques sont cruciales pour la compréhension, l'analyse, l'amélioration et le développement futur des algorithmes d'apprentissage, qu'on peut dorénavant étudier sans scrupules de ne travailler qu'avec des "modèles jouets gaussiens". Mais, au delà du constat théorique se cache un message très contre-intuitif : alors qu'on pensait jusque très récemment que les modèles d'apprentissage automatique traitant de vraies données à la structure complexe devaient, pour bien fonctionner, nécessairement être elles-mêmes très complexes, on s'aperçoit finalement qu'en grandes dimensions, seules les statistiques de premier ordre des données contiennent l'information nécessaire au bon fonctionnement des algorithmes. Il ne faudrait cependant pas surinterpréter le résultat d'universalité : on ne dit jamais ici que les données réelles *ressemblent* à des vecteurs gaussiens, ce que la Figure 3.2.1 confirme bien d'ailleurs, mais seulement que les algorithmes d'apprentissage en grandes dimensions se comportent théoriquement sur des vraies données et sur des vecteurs gaussiens de même statistiques d'ordre un et deux (moyennes et covariances) de la même manière.

La Figure 5.4.2 confirme cette observation et ce résultat théorique en représentant (partie basse) l'histogramme des valeurs propres ainsi que la distribution des deux premiers vecteurs propres d'une matrice de noyau gaussien $K = \{\exp(-\frac{1}{2}\|F(x_i) - F(x_j)\|^2)\}_{i,j=1}^n$, pour x_i des images GANs ou des images réelles issues de la base ImageNet, et F une application d'extraction de représentants par un réseau de neurone profond populaire (soit resnet50 auquel cas $F(x) \in \mathbb{R}^p$ avec $p = 2\,048$, soit vgg16 avec $p = 4\,096$, soit densenet201 avec $p = 1\,920$). L'intérêt de ne pas travailler directement avec un noyau gaussien sur les données brutes (à savoir avec un noyau $\exp(-\frac{1}{2}\|x_i - x_j\|^2)$ mais avec une première transformation des images est que la base de données ImageNet contient des images très complexes qu'un simple noyau gaussien ne parviendrait pas à discriminer efficacement : on utilise donc un premier système intelligent de représentants, dédié aux images, en l'occurrence ici des représentations issues de réseaux de neurones profonds ; comme on l'a vu auparavant, pour x une image GAN, $F(x)$, étant l'image par un réseau de neurones, demeure un vecteur concentré et donc la théorie reste valable. La figure démontre bien les prédictions théoriques : qu'il s'agisse des données GANs ou réelles, les spectres (valeurs et vecteurs propres) de la matrice à noyau K dans toutes les modalités précédentes sont quasi identiques pour $F(x_i)$ issus d'images x_i (en gris) ou pour des $F(x_i)$ modélisés par des vecteurs gaussiens (en vert). Au passage, la partie supérieure de la figure donne des exemples (assez choquant il faut en convenir !) du type d'images complètement artificielles mais bluffantes générées par le GAN à partir des images réelles précisées aussi dans la partie supérieure.

En définitive, la porte ouverte par les matrices aléatoires donne sur une myriade de nouvelles explorations théoriques qui vont permettre, enfin !, de mieux comprendre le comportement des algorithmes que nous avons étudiés jusqu'ici dans le cours mais qui, lorsqu'ils sont appliqués sur des vraies données, restent aujourd'hui très compliqués à comprendre. L'avenir de la recherche en apprentissage automatique et en intelligence artificielle qui, de plus en plus, réclament plus de garanties et de compréhensions théoriques, passe très vraisemblablement par cette nouvelle porte...

5.5. Exercices

Exercice 12 (SVM, LSSVM, régression logiqtique... à noyaux). Le Listing 4.2 a fourni un code complet pour implémenter les différents classifieurs *linéaires* étudiés dans le chapitre précédent. Le but de l'exercice est de reprendre ces classifieurs et de le adapter à un modèle de représentations de données à noyaux. À savoir, il s'agit

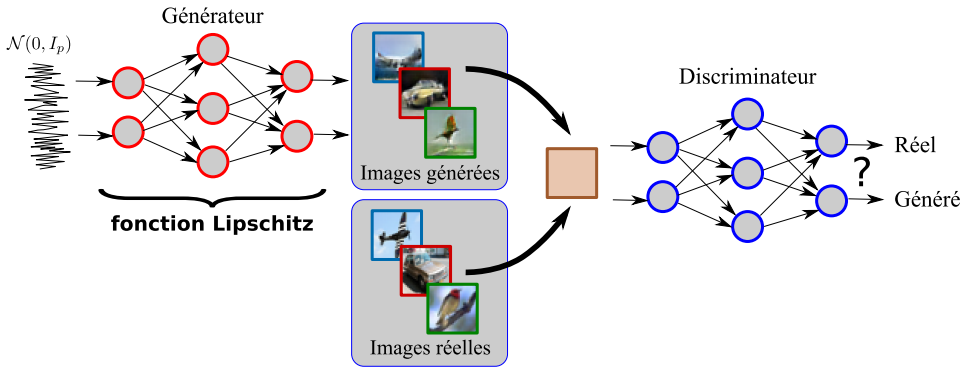

FIGURE 5.4.1. Représentation schématique du fonctionnement d'un GAN : le réseau générateur de gauche tente de tromper le réseau discriminateur de droite en proposant des images de plus en plus "ressemblant" aux images réelles de la base d'entrainement.

— d'une part de ne plus travailler avec un modèle linéaire pour les données (`X=randn(p,n)+mu*y'`) mais avec des données non-linéairement séparables : on pourra pour cela générer des gaussiennes de covariances distinctes mais de même moyenne (ou de moyenne proche), ou idéalement travailler avec un vrai jeu de données (proprement centré et mis à l'échelle) telle que la base MNIST ;
— d'autre part, de reprendre le code du Listing 4.2 en remplaçant tous les opérateurs linéaires du type `X'X` ou `X'x` par des noyaux : on prendra notamment des noyaux $k(x,y) = \exp(-\frac{1}{2\sigma^2}\|x-y\|^2)$ en jouant sur le paramètre σ ou $k(x,y) = P(x^\top y)$ pour un certain polynôme $P(t)$.

Implémenter ces différentes alternatives et comparer les résultats en termes d'erreurs d'entrainement et de test. Tirer notamment des conclusions en termes de surapprentissage et de capacité de généralisation des algorithmes.

EXERCICE 13 (Projections aléatoires et leurs limites). Reprendre l'exercice précédent mais, au lieu de remplacer `X'X` par un noyau, remplacer `X` par `Phi`, une matrice de représentation par projection aléatoire de chaque vecteur de données. Choisir en particulier des *projections aléatoires de Fourier* : en jouant sur le nombre de représentations aléatoires, montrer que les performances de l'algorithme ainsi obtenu convergent bien vers celles du noyau limite (dans le cas des projections de Fourier, vers le noyau gaussien). Jouer alors avec les différents paramètres du problème et établir un compromis entre "complexité" et "performance" lié au choix du projecteur aléatoire ou du noyau limite. Est-il toujours mieux d'utiliser le noyau limite ? Établir plus précisément le compromis optimal à effectuer en fonction des paramètres mis en jeu.

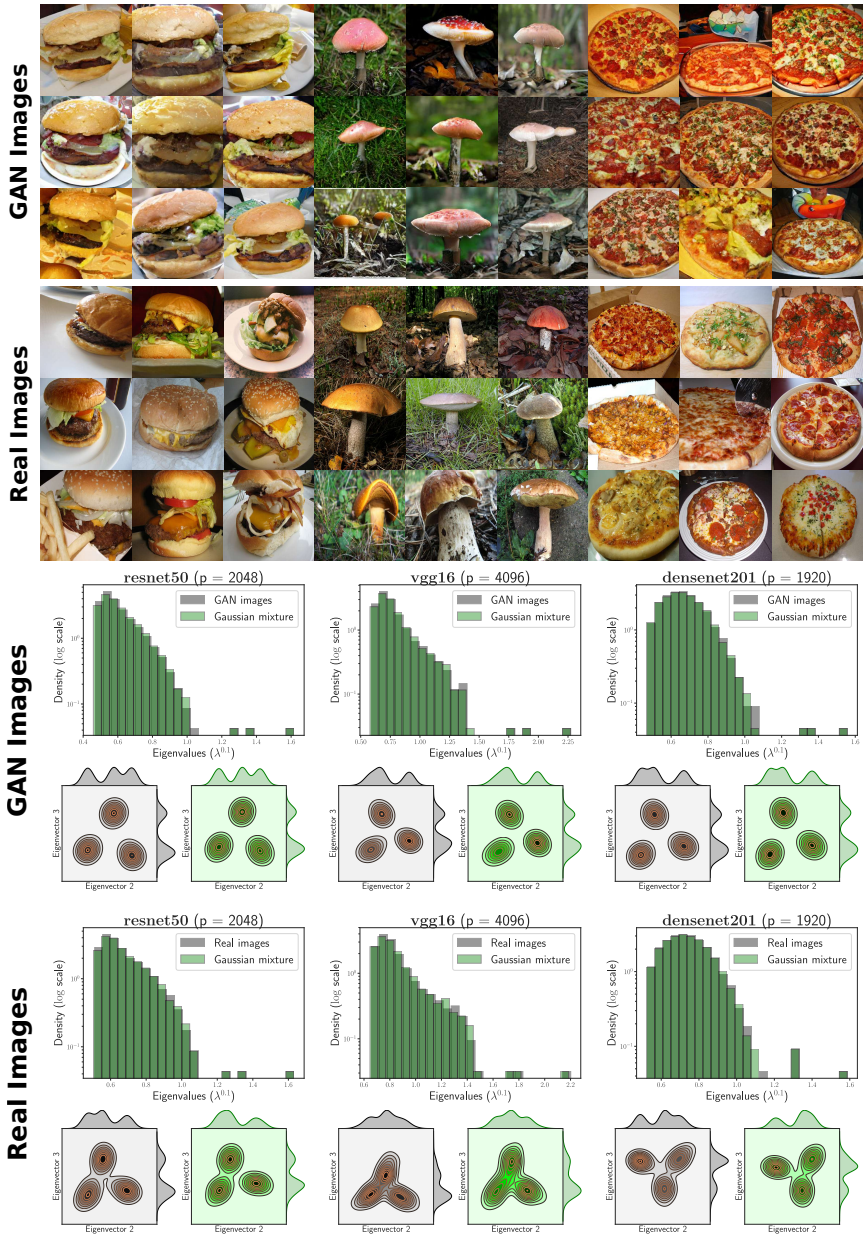

FIGURE 5.4.2. En haut, les vecteurs de données x_i : images réelles de la populaire base de donnée ImageNet ainsi que des (fausses) images produites par des réseaux de neurones antagonistes (GANs) à partir de vecteurs gaussiens isotropes. En bas, l'histogramme des valeurs propres, ainsi qu'une représentation en deux dimensions des deux vecteurs propres dominants d'une matrice à noyau $K = \{\exp(-\frac{1}{2}\|F(x_i) - F(x_j)\|^2\}_{i,j=1}^n$, pour x_i ces images et $F(x)$ une représentation moderne par réseau de neurones (VGG, resnet, etc.) dont la taille p est indiquée entre parenthèses (en gris) ou $F(x_i)$ des vecteurs gaussiens aléatoires ayant les mêmes statistiques (en vert). La coïncidence entre le comportement des matrices K sous hypothèse gaussienne ou réelle est frappante : elle est en fait démontrée rigoureusement pour les images GANs (car elles proviennent formellement de transformations Lipschitz de vecteurs gaussiens) mais ne peut évidemment pas être prouvée pour des vraies images (seulement fortement pressentie étant donné le résultat théorique sur les images GAN !).

Chapitre 6

LSSVM et régression : des ELM aux ESN

Dans les chapitres précédents, nous avons successivement présenté des méthodes d'apprentissage basées sur des idées plutôt heuristiques et assez difficilement défendables ou du moins analysables théoriquement (notamment l'approche des plus proches voisins et les arbres de décision), avant de nous tourner vers des formulations de problèmes d'optimisation qu'on peut tantôt voir sous un angle géométrique, tant tôt sur l'angle de la minimisation d'un risque empirique régularisé bien formulé. Pour ces derniers modèles d'algorithmes d'apprentissage, nous avons observé qu'ils se résument souvent à une forme plus ou moins complexe d'un problème de régression linéaire où il s'agit d'identifier un vecteur w : ce vecteur w permet alors d'évaluer la classe d'un vecteur x inconnu par une fonction de décision $g(x) = w^\mathsf{T}\phi(x)$ (avec potentiellement l'addition d'un biais scalaire b) où ϕ "projette" x dans un espace où les données sont plus facilement linéairement séparables. Seule la méthode d'apprentissage du vecteur w change selon qu'on utilise un SVM, un LSSVM, une régression logistique, ou toute autre forme de minimisation régularisée du risque empirique.

Sous cette synthèse très simplificatrice des algorithmes étudiés jusqu'alors, on peut en fait créer un lien assez évident entre minimisation régularisée du risque empirique et une classe simplifiée de réseaux de neurones : les *réseaux de neurones aléatoires*, et en particulier les dites *extreme learning machines*, que nous allons brièvement présenter dans ce chapitre.

Mais ces extreme learning machines sont élégantes sur le principe, elles se réduisent à un simple régresseur linéaire régulariser sur une base de projection non-linéaire (souvent aléatoire) des données, et n'apporte au final pas grand chose de nouveau par rapport aux LSSVM. Mais le cœur du chapitre consistera à présenter une autre classe, plus riche, de ces réseaux de neurones aléatoires et qui ont un intérêt assez central pour traiter de l'apprentissage de *séries temporelles* : prédiction, classification, etc. Ces réseaux sont en effet moins complexes, mais plus facilement analysables et de surcroît donnent de très bons résultats sur des séries temporelles quasi-chaotiques, que leur compétiteur naturel que sont les réseaux profonds de type LSTM (pour *long short term memory*) très compliqués à stabiliser et faire fonctionner.

6.1. "Extreme learning machines"

6.1.1. Introduction aux ELM. Une *extreme learning machine* (il n'existe pas de traduction française à notre connaissance) n'est pas un objet bien compliqué, et nous l'avons en réalité déjà étudié sous le nom de LSSVM, mais il est intéressant de le présenter car il est motivé par une approche "réseaux de neurones" plutôt que minimisation du risque empirique.

L'idée suit l'idée générale du perceptron de Rosenblatt à une exception majeure prête : chaque donnée (ou représentant de donnée) $x_i \in \mathcal{X} = \mathbb{R}^p$ traverse une première couche linéaire $W_{\text{in}} \in \mathbb{R}^{N\times p}$ *qui est fixée et jamais "apprise"* avant de se voir appliquer une fonction d'activation non-linéaire $\sigma(\cdot)$ appliquée entrée par

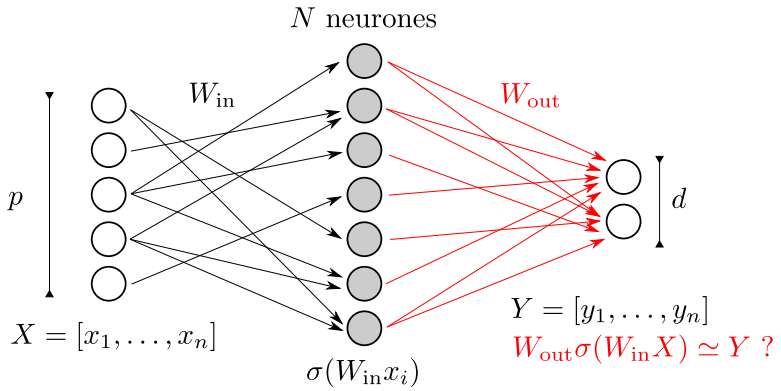

FIGURE 6.1.1. Représentation schématique d'un ELM : simple réseau de neurones à une couche cachée dont la matrice de connexion W est *fixe* et seule la matrice de *classification* (ou régression) W_{out} est apprise.

entrée. Une seconde couche linéaire $W_{\text{out}} \in \mathbb{R}^{d\times N}$ permet alors d'obtenir la sortie $y \in \mathbb{R}^d$ souhaitée, où d sera comme souvent dans ce cours pris de petite taille ($d = 1$ si on souhaite $y \in \{\pm 1\}$ par exemple). Au final, l'équation du réseau s'écrit simplement sous la forme

$$y = W_{\text{out}}\sigma(W_{\text{in}}x)$$

comme illustré schématiquement dans la Figure 6.1.1.

Dans le réseau de neurones de Rosenblatt (le perceptron), étant données des observations $(x_1, y_1), \ldots, (x_n, y_n)$, il s'agirait d'apprendre simultanément les matrices W_{in} et W_{out} afin de minimiser une fonction de coût, tel qu'un risque empirique $\sum_i \ell(W_{\text{out}}\sigma(W_{\text{in}}x_i), y_i)$. Cependant, pour des fonctions de coût naturelles $\ell(\cdot,\cdot)$, cette minimisation, qui est généralement convexe en W_{in} et convexe en W_{out}, n'est bien souvent *pas conjointement convexe en* $(W_{\text{in}}, W_{\text{out}})$. On ne peut donc résoudre le problème qu'approximativement en utilisant par exemple des méthodes de descente de gradients mais qui, inévitablement (d'autant plus lorsque le réseau et les données sont de grandes tailles) atteindront un minimum local, jusqu'à aujourd'hui très difficile à analyser et à stabiliser.

Le fait de *fixer* W_{in} change la donne : il s'agit maintenant seulement d'optimiser la matrice (ou le vecteur) de sortie $W_{\text{out}} \in \mathbb{R}^{d\times N}$ pour une certaine fonction de coût $\ell(\cdot,\cdot)$. Et, évidemment, le choix naturel de l'ELM va se porter sur un coût quadratique régularisé, à savoir, pour un jeu de données d'entrainement $(x_1, y_1), \ldots, (x_{n_{\text{train}}}, y_{\text{train}})$,

$$E_{\text{train}} = \sum_{i=1}^{n} \left(W_{\text{out}}\sigma(W_{\text{in}}x_i) - y_i\right)^2 + \gamma\|W_{\text{out}}\|_F^2$$

où $\|\cdot\|_F$ est la norme de Frobenius, extension naturelle de la norme euclidienne des vecteurs pour les matrices, définie par $\|A\|_F^2 = \sum_{ij} A_{ij}^2$.

Il s'agit ici, comme dans le cas des LSSVM, d'un problème d'optimisation quadratique non contraint dont la solution est donnée par annulation du gradient

$$\gamma W_{\text{out}} + \sum_{i=1}^{n} \left(W_{\text{out}}\sigma(W_{\text{in}}x_i) - y_i\right)\sigma(W_{\text{in}}x_i)^{\mathsf{T}} = 0$$

ou de manière équivalente

$$W_{\text{out}}\left(\gamma I_N + \sigma(W_{\text{in}}X)\sigma(W_{\text{in}}X)^{\mathsf{T}}\right) = Y\sigma(W_{\text{in}}X)^{\mathsf{T}}$$

avec, on le rappelle, $Y = [y_1, \ldots, y_n] \in \mathbb{R}^{N\times n}$, $X = [x_1, \ldots, x_n] \in \mathbb{R}^{p\times n}$ et $\sigma(W_{\text{in}}X)$ est une application entrée-par-entrée de l'opérateur σ sur la matrice $W_{\text{in}}X$. Au final, $\sigma(W_{\text{in}}X)\sigma(W_{\text{in}}X)^{\mathsf{T}}$ étant semi-définie positive, on peut inverser la matrice entre parenthèse à gauche du signal égal, et on obtient donc comme solution

$$W_{\text{out}} = Y\sigma(W_{\text{in}}X)^{\mathsf{T}}\left(\gamma I_N + \sigma(W_{\text{in}}X)\sigma(W_{\text{in}}X)^{\mathsf{T}}\right)^{-1}.$$

Et donc on reconnait à nouveau la forme d'un régresseur linéaire régularisé, qu'il est pratique d'exprimer sous la forme

$$\begin{aligned} W_{\text{out}}^{\mathsf{T}} &= \left(\gamma I_N + \sigma(W_{\text{in}}X)\sigma(W_{\text{in}}X)^{\mathsf{T}}\right)^{-1}\sigma(W_{\text{in}}X)Y^{\mathsf{T}} \\ &= \sigma(W_{\text{in}}X)\left(\gamma I_n + \sigma(W_{\text{in}}X)^{\mathsf{T}}\sigma(W_{\text{in}}X)\right)^{-1}Y^{\mathsf{T}}. \end{aligned}$$

La phase d'entraînement du réseau de neurones est donc ici élémentaire et ne requiert aucune espèce d'apprentissage numérique, étant donné que W_{in} demeure fixe et que W_{out} prend une forme explicite. Une fois le réseau ainsi "entraîné", toute nouvelle donnée $x \in \mathbb{R}^p$ traverse le réseau via la formule

$$\begin{aligned} W_{\text{out}}\sigma(W_{\text{in}}x) &= \left(\sigma(W_{\text{in}}x)^{\mathsf{T}}W_{\text{out}}^{\mathsf{T}}\right)^{\mathsf{T}} \\ &= \left(\sigma(W_{\text{in}}x)^{\mathsf{T}}\sigma(W_{\text{in}}X)\left(\gamma I_n + \sigma(W_{\text{in}}X)^{\mathsf{T}}\sigma(W_{\text{in}}X)\right)^{-1}Y^{\mathsf{T}}\right)^{\mathsf{T}}. \end{aligned}$$

En particulier, en définissant la fonction de représentation $\phi(x) = \sigma(W_{\text{in}}x) \in \mathbb{R}^N$ et le noyau associé $k(x, x') = \phi(x)^{\mathsf{T}}\phi(x') = \sigma(W_{\text{in}}x)^{\mathsf{T}}\sigma(W_{\text{in}}x')$, on trouve

$$W_{\text{out}}\sigma(W_{\text{in}}x) = \left(k(x,\cdot)^{\mathrm{T}}\left(\gamma I_n + K\right)^{-1}Y^{\mathsf{T}}\right)^{\mathsf{T}} \in \mathbb{R}^d$$

où $K = \{k(x_i, x_j)\}_{i,j=1}^n \in \mathbb{R}^{n\times n}$ et $k(x,\cdot) = \{k(x, x_i)\}_{i=1}^n \in \mathbb{R}^n$.

Si la sortie du réseau est scalaire, on peut écrire $W_{\text{out}}^{\mathsf{T}} \equiv w \in \mathbb{R}^N$ et $Y^{\mathsf{T}} = y \in \mathbb{R}^n$ sous formes vectorielles, et finalement le réseau extreme learning machine se réduit à une fonction de décision

$$g(x) \equiv k(x,\cdot)^{\mathrm{T}}\left(\gamma I_n + K\right)^{-1}y.$$

Comparée à l'Equation 4.3.1, prise ici pour $b = 0$ (ce qui est équivalent, nous l'avons vu, à travailler sur des données recentrées et de même cardinal par classe), nous nous rendons compte que les extreme learning machines ne sont rien d'autre qu'un LSSVM à noyau *pour une forme très spécifique de fonction de représentation*, à savoir la famille des $\phi(x) = \sigma(W_{\text{in}}x) \in \mathbb{R}^N$ où les paramètres libres sont : (i) la taille N du réseau de neurones, (ii) la (ou les) fonction d'activation σ et (iii) $W_{\text{in}} \in \mathbb{R}^{N\times p}$ arbitraire. Le noyau associé est donné par $k(x, x') = \phi(x)^{\mathsf{T}}\phi(x')$.

6.1.2. Le choix de W_{in} et le lien avec les projections aléatoires. Cette première analogie entre extreme learning machines et LSSVM cache une autre analogie, avec les projecteurs aléatoires. En effet, en supposant que $W_{\text{in}} = [w_1^{\mathsf{T}}, \ldots, w_N^{\mathsf{T}}]^{\mathsf{T}} \in \mathbb{R}^{N\times p}$ est formé par N vecteurs indépendants tirés d'une certaine mesure de probabilité μ sur $\mathbb{R}^p$, on peut alors voir la fonction $x \mapsto \sigma(w_i^{\mathsf{T}}x)$ comme un projecteur aléatoire non linéaire, et donc $x \mapsto \sigma(W_{\text{in}}x)$ comme un vecteur de taille N de projecteurs aléatoires.

D'après nos discussions dans les sections précédentes, en choisissant bien le projecteur aléatoire $x \mapsto \sigma(w_i^{\mathsf{T}}x)$, le noyau $k_N(x, y) = \sigma(w_i^{\mathsf{T}}x)^{\mathsf{T}}\sigma(w_i^{\mathsf{T}}y)$ peut approximer un noyau limite

$$k(x, y) = \lim_{N\to\infty} k_N(x, y) = \lim_{N\to\infty} \sigma(w_i^{\mathsf{T}}x)^{\mathsf{T}}\sigma(w_i^{\mathsf{T}}y)$$

où la limite existe en général presque sûrement. En particulier, nous l'avons vu précédemment, en prenant $\sigma(t) = \exp(-2\pi\imath t)$ (ou alternativement pour maintenir des sorties réelles, $\sigma(t) = \cos(2\pi\imath t)$ et $\sigma(t) = \sin(2\pi\imath t)$ pour un neurone sur deux), on trouve

$$k(x,y) = \lim_{N\to\infty} k_N(x,y) = \exp\left(-\frac{1}{2}\|x-y\|^2\right).$$

C'est une façon de visualiser les extreme learning machines : ces réseaux de neurones, construits en prenant W_{in} aléatoirement, engendrent un nombre arbitrairement grand de projections aléatoires non linéaires des données et qui, lorsque N est assez grand, approximent un noyau bien maîtrisé. L'avantage de cette méthode apparaît lorsque le nombre de données n est très grand, donc dans une configuration où $n \gg N$ typiquement et où, en refusant volontairement d'exploiter l'astuce du noyau, il est plus pratique de travailler avec le représentant $\phi(x) = \sigma(W_{\text{in}}x) \in \mathbb{R}^N$ qu'avec le noyau limite $K \in \mathbb{R}^{n\times n}$; en ajustant le nombre N de projections aléatoires, on contrôle alors assez facilement la complexité de l'algorithme.

En somme, et malgré leur étonnant succès dans le début des années 2010, les extreme learning machines ne sont rien d'autres que les régresseurs linéaires d'une famille limitée de projecteurs aléatoires (de la forme $\phi(x) = \sigma(W_{\text{in}}x)$) masqués derrière le mot-clé probablement un peu abusif de "réseaux de neurones". Le terme "extreme" étant choisi ici pour insister sur le fait que N est en général très grand.

Néanmoins, étudier les ELMs a pour nous un autre intérêt : celui de faire le pont entre les méthodes d'apprentissage discutées jusqu'ici et les *echo-state networks* qui forment la dernière section de notre cours et qui, eux, portent un intérêt tout particulier dans l'*apprentissage (régression, classification, prédiction) de séries temporelles*.

6.2. "Echo-state networks"

Les *echo-state networks* peuvent être vus comme l'extension naturelle des ELM en des réseaux de neurones *récurrents*. Nous avons vu en effet dans l'introduction du cours aux réseaux de neurones que, si ces derniers doivent "mimer" le comportement d'un réseau de neurones biologiques, l'interconnexion entre les neurones se doit d'être bien plus riche qu'une succession de connexions "par couches". Le perceptron, ou le cas particulier des ELMs étudiés dans la section précédente, opèrent en deux couches successives, les neurones centraux n'étant aucunement connectés entre eux. On parle dans ce cas de réseaux *feedforward* en ce sens que l'information traverse le réseau de bout en bout sans jamais être mémorisée et sans jamais permettre aux données suivantes d'exploiter le traitement effectué sur les données précédentes. Cette modélisation limitée, nous l'avons vu, semble assez pertinente pour des tâches de vision, où chaque nouvelle image x introduite dans le réseau est supposée distincte de l'image suivante x' qui entrera par la suite, mais elle ne l'est plus lorsque les données entrant successivement dans le réseau deviennent corrélées temporellement, comme c'est le cas des séries temporelles.

6.2.1. Présentation des ESNs. Les réseaux *echo-state* proposent une alternative simple en adaptant légèrement (mais avec des conséquences cruciales) les réseaux ELM. Pour simplifier les considérations à venir, nous allons nous placer dans un contexte mono-varié : à savoir, l'espace $\mathcal{X}$ des données d'entrée et l'espace $\mathcal{Y}$ des sorties sont restreints à $\mathbb{R}$. La généralisation à $\mathbb{R}^p$ et $\mathbb{R}^d$, respectivement, est assez simple mais ne fait qu'alourdir les notations.

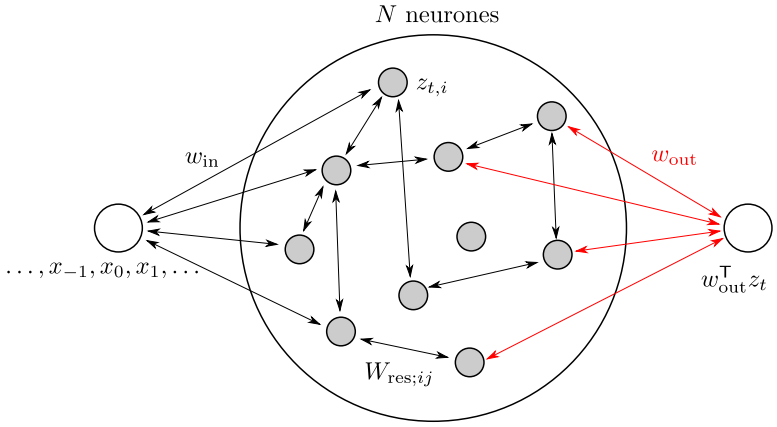

FIGURE 6.2.1. Représentation schématique d'un ESN : comme le réseau ELM de la Figure 6.1.1, c'est un simple réseau de neurones à une couche cachée de matrice (ici vecteur) d'entrée $w_{\text{in}} \in \mathbb{R}^N$, qu'on gardera fixe, de matrice (ici aussi vecteur) de sortie $w_{\text{out}} \in \mathbb{R}^N$ qu'on entrainera ; mais surtout ayant une matrice d'interconnexion neuronale W_{res} qui induit un *echo* des états $\ldots, z_{t-2}, z_{t-1}$ du réseau jusqu'au temps t, et qu'à nouveau on gardera fixe. L'ensemble des neurones de ce réseau "accumule la mémoire" des données passées et est ainsi appelé "réservoir". Ce réseau est adapté à des opérations de classifications, régressions et prédictions de séries temporelles.

Dans le contexte des ELMs, le modèle associé serait donc celui d'un réseau dont la sortie associée à une entrée $x \in \mathbb{R}$ serait donnée par

$$r = w_{\text{out}}^{\mathsf{T}} \sigma(w_{\text{in}} x) \in \mathbb{R}$$

que nous allons volontairement réécrire

$$r = w_{\text{out}}^{\mathsf{T}} z \in \mathbb{R}, \quad \text{où} \quad z = \sigma(w_{\text{in}} x) \in \mathbb{R}^N$$

avec $w_{\text{in}}, w_{\text{out}} \in \mathbb{R}^N$, et la sortie, que l'on souhaite proche de y, est dénotée ici r en référence à l'appellation *readout* en vigueur dans les réseaux ESN.

Supposons maintenant que les données x sont issues d'une série temporelle $\ldots, x_{t-1}, x_t, x_{t+1}, \ldots \in \mathbb{R}$, paramétrée par le temps t, ayant pour sorties associées la série $\ldots, r_{t-1}, r_t, r_{t+1}, \ldots$. Afin d'induire de la mémoire au sein du réseau et notamment de le rendre capable de prédiction (ce qu'un ELM qui traite les données une-à-une et indépendamment ne peut faire !), les echo-state networks introduisent une nouvelle matrice de connexion carrée $W_{\text{res}} \in \mathbb{R}^{N \times N}$ entre les neurones ('res' signifiant ici *réservoir*, nous verrons pourquoi), de sorte que r_t devient maintenant :

$$r_t = w_{\text{out}}^{\mathsf{T}} z_t \in \mathbb{R}, \quad \text{où} \quad z_t = \sigma\left(w_{\text{in}} x_t + W_{\text{res}} z_{t-1}\right).$$

Dans cette écriture, le terme $z_t \in \mathbb{R}^N$ est appelé *état du réservoir* au temps t. Cet état ne dépend non plus seulement de la nouvelle entrée $w_{\text{in}} x_t$ introduite dans le réseau, mais également de l'état z_{t-1} précédent du réseau dont l'*echo* est ressenti à travers la matrice W_{res}. La Figure 6.2.1 illustre cette nouvelle configuration.

L'expression de r_t peut paraître bien plus compliquée que celle des ELMs en ce sens que z_t est définie à partir de z_{t-1}, elle-même obtenue à partir de z_{t-2} et ainsi de suite. D'ailleurs, notons au passage que pour être complètement défini, il faut qu'une référence z_0 existe : en général on utilisera plutôt une référence lointaine,

de type $z_{-100} = 0_N$, qui réalise une "vidange" (en anglais *flushout*) du réseau ; le fait de prendre cette référence loin dans le passé assure (du moins on l'espère) qu'au temps $t = 0, 1, 2, \ldots$, le réseau se comporte de manière stationnaire et n'est plus affecté par le choix particulier de z_{-100} (la mémoire de cet événement lointain s'étend "effacée"). Néanmoins, en dépit de cette dépendance temporelle, les réseaux ESNs sont extrêmement simples à opérer. Comme pour les ELMs, la matrice (ou vecteur) de connexion w_{in} entre les entrées et les neurones est maintenue fixe. De même, la matrice de réservoir $W_{\text{res}} \in \mathbb{R}^{N\times N}$ est également choisie dès le début et reste fixée. Seule à nouveau la matrice (ou vecteur) de sortie w_{out} sera entrainée. Et pour ce faire, rien de plus compliqué qu'avec un ELM ! En effet, en supposant l'existence d'un ensemble $(x_1, y_1), \ldots, (x_n, y_n)$ d'apprentissage et en minimisant le même risque empirique quadratique régularisé que pour les ELMs

$$\inf_{w_{\text{out}}\in\mathbb{R}^N} \sum_{i=1}^{n} (r_i - y_i)^2 + \gamma \|w_{\text{out}}\|^2 = \inf_{w_{\text{out}}\in\mathbb{R}^N} \sum_{i=1}^{n} (w_{\text{out}}^{\mathsf{T}} z_i - y_i)^2 + \gamma \|w_{\text{out}}\|^2$$

on a pour solution explicite très simple

$$w_{\text{out}} = Z \left(\gamma I_n + Z^{\mathrm{T}} Z\right)^{-1} y$$

avec ici $Z = [z_1, \ldots, z_n] \in \mathbb{R}^{N\times n}$, où nous rappelons que z_t est évalué de manière récurrente via $z_t = \sigma(w_{\text{in}} x_t + W_{\text{res}} z_{t-1})$ à partir d'un certain z_0 (ou un indice temporel plus lointain dans le passé) de référence.

Le lien étroit entre le régresseur w_{out} et les régresseurs de projecteurs aléatoires non récurrents permet de donner une interprétation assez claire de ce que l'ESN cherche à réaliser : grâce à son grand nombre de neurones potentiel N activés aléatoirement par les vecteurs/matrices w_{in} et W_{res}, ainsi que par la fonction d'activation non linéaire σ, l'ESN produit N *représentations temporelles non linéaires* de la série temporelle d'entrée $\ldots, x_{t-1}, x_t, x_{t+1}, \ldots$. Ainsi, au lieu d'une représentation seulement "spatiale" des données x_t en un certain $\phi(x_t) \in \mathbb{R}^N$, on obtient une forme plus riche de représentation *spatio-temporelle* $\phi(x_t, x_{t-1}, \ldots) \in \mathbb{R}^{N\times\infty}$. Ces représentations, souvent tirées aléatoirement, fournissent une base (qu'on espère riche !) permettant d'effectuer des opérations de régression ou de classification de la série temporelle toute entière.

Dès lors, contrairement aux ELMs, une fois entraîné, le réseau fonctionne dans un mode "série temporelle" et attend donc naturellement l'arrivée de "la suite" de la série d'entrainement. Il ne s'agit donc plus de tester la sortie associée à une nouvelle entrée arbitraire, mais plutôt associée à la suite de la série. Ainsi, la nouvelle donnée est x_{n+1} et le score de sortie du réseau pour cette nouvelle valeur vaut

$$g(x_{n+1}) = \sigma(w_{\text{in}} x_{n+1} + W_{\text{res}} z_n)^{\mathsf{T}} w_{\text{out}} = z_{n+1}^{\mathsf{T}} \left(\gamma I_n + Z^{\mathrm{T}} Z\right)^{-1} y$$

en définissant

$$z_{n+1} = \sigma(w_{\text{in}} x_{n+1} + W_{\text{res}} z_n)^{\mathsf{T}} Z^{\mathsf{T}}.$$

Parmi les tâches d'apprentissage en vogue dans les ESNs, celle de prédiction de séries temporelles "pseudo-chaotiques" connait un fort succès. Par pseudo-chaotique ou quasi-chaotique, on doit comprendre une série temporelle dont l'évolution est déterminée par des termes à mémoire longue : en particulier, si cette mémoire disparait, on ne peut plus prédire la série. La série n'est cependant pas chaotique en ce sens qu'une petite erreur dans la mémoire longue n'est pas suffisante pour perdre les capacités de prédictions. Quoiqu'il en soit, l'un des objectifs et des succès des ESNs concernent la prédiction de ces séries temporelles difficiles à anticiper.

Si la tâche d'apprentissage est de prédire la série temporelle en avance de T pas de temps dans le futur, on a alors affaire à un problème de régression où $y_t = x_{t+T}$.

Il suffit alors pendant l'apprentissage de remplacer y de l'expressio de w_{out} par la version "décalée de T" pas de temps du vecteur $z = [z_1, \ldots, z_n]^\mathsf{T}$.

Pour bien prendre conscience de la capacité de mémorisation du réseau d'événement potentiellement lointains dans le passé, une autre tâche, qui sera plutôt un "exemple-jouet" consiste à envoyer dans le réseau la série temporelle $\ldots, 0, 0, 1, 0, 0, \ldots$ et d'y associer la sortie $\ldots, 0, 0, 1, 0, 0, \ldots$ mais avec un décalage de T pas de temps dans le *passé*. L'erreur d'entrainement $\sum_{i=1}^{n}(r_i - y_i)^2$ traduira ici la "dégradation" du signal (ici un Dirac à un temps donné) lorsque l'on cherche à le récupérer fidèlement par le réseau à un temps plus lointain dans l'avenir. Nous allons voir dans ce qui suit que le choix de la matrice W_{res} a une énorme influence sur ces performances.

Avant de conclure cette section, un dernier point est à noter : on ne peut ici plus effectuer d'analogie simple avec les méthodes à noyaux. En effet, les colonnes z_i de Z ne prennent plus la forme d'une représentation $\phi(x_i)$ de la donnée x_i mais dépendent également de $x_{i-1}, x_{i-2}, \ldots$. D'ailleurs, derrière cette expression très simple se cache en fait une grande richesse en terme de capacité de mémorisation du réseau, si riche et en vérité si complexe que même les recherches les plus récentes dans le domaine peinent à maîtriser pleinement le comportement, comme nous allons le voir dans ce qui suit.

6.2.2. Fonctionnement interne et paramétrisation des ESNs. Le point central de fonctionnement des ESNs concerne le réglage de la matrice W_{res}, en corrélation avec le choix de la fonction d'activation σ, qui ensemble jouent le rôle de fournisseur de mémoire. Il faut en effet bien comprendre ce qu'il se passe au sein de l'ESN et, pour cela, il est intéressant (même si en pratique on ne fera pas cela !) de supposer en premier lieu que $\sigma(t) = t$ est une simple activation linéaire et que l'on ne fournit aucun échantillon x_i au réseau (à savoir $x_i = 0$).

Alors, dans ce cas, on obtient au temps t :

$$z_t = \sigma(w_{\text{in}} x_t + W_{\text{res}} z_{t-1}) = W_{\text{res}} z_{t-1} = W_{\text{res}}^2 z_{t-2} = \ldots = W_{\text{res}}^k z_{t-k}$$

et ce pour tout $k \in \mathbb{N}$. Ainsi, les phénomènes suivants peuvent avoir lieu, en fonction du rayon spectral $\rho(W_{\text{res}})$, défini comme la valeur propre de plus haute amplitude :

- soit $\rho(W_{\text{res}}) < 1$, et alors $\|z_t\| = \|W_{\text{res}}^k z_{t-k}\|$. Comme $\rho(W_{\text{res}}) \to_{k\to\infty} 0$, il vient alors que $z_t \to_{k\to\infty} 0$ (il suffit pour cela de décomposer z_{t-k} sous la forme d'une combinaison linéaire de vecteurs propres de W_{res}) : la mémoire dans ce cas s'évanouit, et ce, à une vitesse exponentiellement rapide !
- soit $\rho(W_{\text{res}}) > 1$, et dans ce cas, si z_{t-k} a un alignement non nul avec l'un des vecteurs propres de W_{res} associé à l'une de ces valeurs propres d'amplitude supérieure à 1 (ce qui, dans le cas d'un tirage aléatoire de W_{res} risque de se produire avec probabilité 1), on trouve que $\|z_t\| \to_{k\to\infty} \infty$ et le réseau diverge. On dit qu'il entre en régime *chaotique* ;
- soit $\rho(W_{\text{res}}) = 1$ et alors le réseau reste vraisemblablement stable mais se trouve dans une condition "limite" instable où tout apport supplémentaire d'énergie au réservoir le déstabilise. Ce qui arrivera lorsque l'on prendra en compte l'apport des entrées x_t que nous avons jusqu'alors négligées !

Toute la difficulté de la maîtrise du comportement des ESNs vient précisément de la réintroduction des données $\ldots, x_{t-1}, x_t, x_{t+1}, \ldots$ dans le réseau et surtout du passage du cas linéaire $\sigma(t) = t$ au bien plus intéressant modèle non-linéaire. L'ajout des données d'entrée $\ldots, x_{t-1}, x_t, x_{t+1}, \ldots$ a en fait en général une influence restreinte, comme cela a pu être établi dans certaines études. En particulier, le choix $\rho(W_{\text{res}}) = 1$ devient dans ce cas en effet instable si $\sigma(t) = t$, mais le réseau demeure stable pour n'importe quel choix de $\rho(W_{\text{res}}) < 1$.

Les choses deviennent cependant réellement intéressantes lorsqu'il s'agit de passer du cas linéaire $\sigma(t) = t$ au cas non-linéaire. Dans ce cas, pour ne pas créer des

effets d'échelle, on travaillera naturellement avec des fonctions $\sigma(t)$ 1-Lipschitz (à savoir telles que $|\sigma(t) - \sigma(t')| \leq |t - t'|$).[1] C'est par exemple le cas des populaires fonctions $\sigma(t) = t$, $\sigma(t) = \max(t, 0)$ (aussi communément appelée fonction ReLU pour *rectified linear unit*, un nom vraiment pompeux pour une simple rampe !), ou encore les sigmoïdes $\sigma(t) = \tanh(t)$ et $\sigma(t) = -1 + 2/(1 + \exp(-t))$. La Figure 6.2.2 propose quelques exemples de fonctions d'activation courantes en réseaux de neurones.

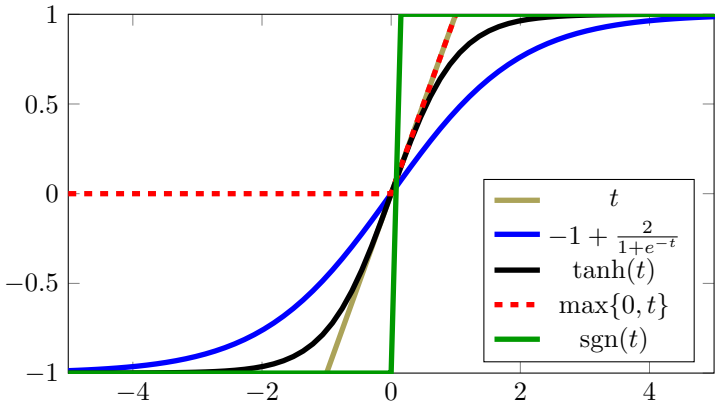

FIGURE 6.2.2. Fonctions d'activation communément employées dans les réseaux de neurones.

Dans le cas de ces activations non linéaires, un phénomène très intéressant se produit : il est possible d'autoriser $\rho(W_{\text{res}}) > 1$ mais en général seulement "légèrement supérieur à 1". Le phénomène qui dans ce cas semble avoir lieu est lié à l'"amortissement" ou l'atténuation de l'énergie du vecteur d'état z_t du réseau du fait du caractère 1-Lipschitz de σ. Prenons par exemple la fonction $\sigma(t) = \max\{0, t\}$: dans ce cas, en moyenne une fois sur deux les valeurs observées avant la traversée des neurones sont négatives et sont donc éliminées du réseau, réduisant effectivement de moitié l'énergie du réservoir et permettant ainsi à $\rho(W_{\text{res}})$ de s'approcher presque de 2. L'intérêt de cette stratégie est considérable car elle permet alors d'accentuer plutôt que de systématiquement diminuer certains "modes du réservoir" qui vont demeurer longtemps à l'intérieur du réservoir au détriment d'autres modes qui vont être éliminés. Si ces modes sont bien sélectionnés (et on imagine que les données d'entrées x_t pousseront les bons modes à rester), une mémoire effectivement à long terme peut être conservée. Cette dernière phrase est volontairement assez floue et peu rigoureuse (nous n'avons pas défini ce que nous avons gratuitement appelé "un mode") du fait que le mécanisme précis de fonctionnement des ESN dans le cas non linéaire est encore très mal compris. En pratique cependant, on sait que les phénomènes suivants ont lieu :

— si $\rho(W_{\text{res}}) < 1$, l'évanouissement exponentiellement rapide de la mémoire est accentué par le caractère 1-Lipschitz de σ qui "comprime" encore la norme des vecteurs d'état du réseau ; dans le cas où on retire les entrées du réseau, le réservoir se vide très vide et le seul attracteur de l'équation $z = \sigma(W_{\text{res}} z)$ est l'état $z = 0_N$ qui n'a évidemment aucun intérêt ;

— si $\rho(W_{\text{res}}) \gg 1$, l'amortissement de la fonction d'activation ne sera pas suffisant et le réseau sera instable, aura un comportement chaotique ; dans

1. Évidemment, si on prend une fonction 2-Lipschitz telle que $\sigma(t) = 2t$, il faudrait nécessairement alors demander que $\rho(W_{\text{res}}) < 1/2$, et on évite cet effet d'échelle pour se simplifier la vie.

ce cas, sans entrée dans le réseau, $z = \sigma(W_{\text{res}} z)$ a un nombre de solutions grandissant exponentiellement avec N, ce qui n'est en fait pas non plus intéressant : cela signifie que le réseau ne se "fixe pas" à sous-classe de représentants mais peut être attiré par beaucoup trop d'états dans l'espace des représentants ;

— si $\rho(W_{\text{res}}) = 1 + \varepsilon$ pour un ε assez petit (en général il faut même que $\varepsilon \sim O(1/N)$ donc diminue avec la taille du réseau), le réseau devient à nouveau stable et, surtout, sans entrée dans le réseau, l'équation $z = \sigma(W_{\text{res}} z)$ admet un nombre polynomial en N de solutions stables : le réseau "s'accroche" donc à des représentants spécifiques qui peuvent rendre compte pour certains de mémoires à très long terme. Le contrôle du choix de ε est cependant très difficile et, en théorie, rien n'empêche que, si le réseau est stable pour un grand nombre d'états z, il peut soudain devenir instable (avec probabilité faible mais non nulle).

La Figure 6.2.3 illustre le comportement d'un ESN pour différents paramètres (nombre de neurones et facteur de régularisation). Observons tout d'abord la très bonne adéquation aux données d'entrainement, ce qui est évidemment attendu, particulièrement pour γ petit. Notons aussi ici l'intérêt crucial de la phase de flushout qui permet au réseau d'avoir le temps (ce temps est d'ailleurs presque trop court ici !) de "s'adapter" aux données : il faut bien comprendre que, jusque là, les données précédentes sont vues comme une séquence de zéros, ce qui crée au temps $t = 1$ une rupture nette de stationnarité. On remarque notamment que pour N petit, le prédicteur est limité en flexibilité mais permet de réduire le facteur de régularisation γ (cependant en créant de fortes instabilités pendant la période initiale de flushout), alors que N plsu large impose des valeurs plus larges de γ pour éviter le surapprentissage. Notons qu'ici, $\rho(W_{\text{res}})$ est pris légèrement supérieur à 1, pour un choix de fonction d'activation $\sigma(t) = \tanh(t)$.

Listing 6.1. Code Matlab de la Figure 6.2.3

```
% Paramètres
n=1000;
N=1000;
gamma=1;            % régularisation
r=1.01;             % rayon spectral de W_res
sigma=@(t)tanh(t);  % activation

% Fonction à prédire et shift
f=@(t) cos(16*t)'.*sin(10*t+pi/2)';
shift=.05;

% Construction aléatoire des matrices de connexion
w_in=randn(N,1);
w_in=w_in/norm(w_in);

W_res=randn(N,N);
W_res=r*W_res/abs(eigs(W_res,1));

% Entrée 'x' et sortie 'y' associée
x=f(linspace(0,1,n));
y=f(linspace(0,1,n)-shift);

% Création de l'ESN: (i) flushout entre t=1 et t=200, (ii)
    apprentissage entre t=201 et t=400, (iii) régression pour t>400
```

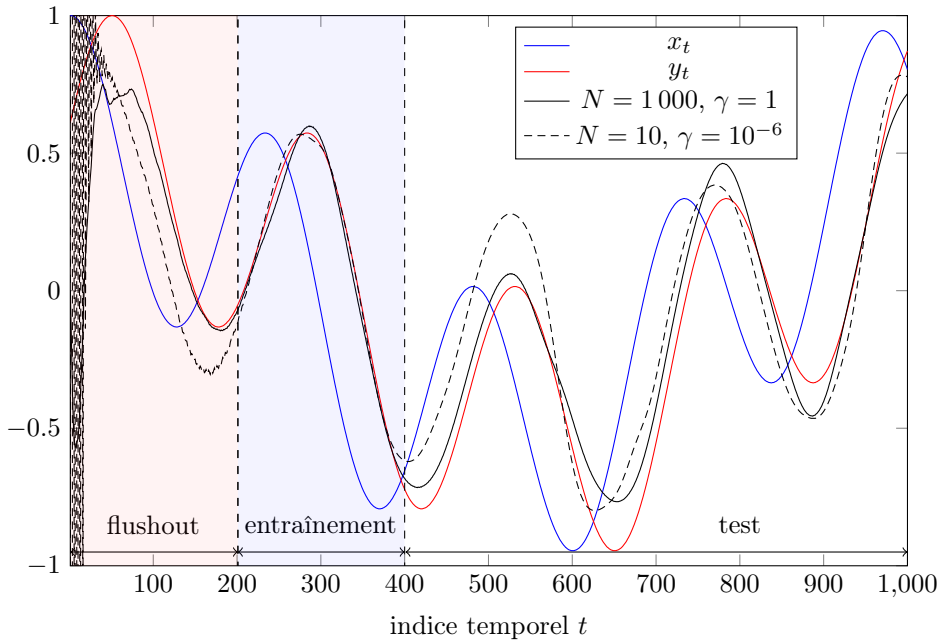

FIGURE 6.2.3. Exemple de prédiction de série temporelle par un ESN : l'objectif est de prédire la fonction $t \mapsto \cos(16t)\sin(10t + \pi/2)$. Période de flushout, apprentissage, puis prédiction. Paramètres : $n = 1\,000$, $\gamma = 1$, $\rho(W_{\text{res}}) = 1.01$, et $\sigma(t) = \tanh(t)$.

```
Z=zeros(N,n);
for i=2:n
        Z(:,i)=sigma(w_in*x(i)+W_res*Z(:,i-1));
end
w_out=Z(:,201:400)*inv(gamma*eye(200)+Z(:,201:400)'*Z(:,201:400))*y
    (201:400);
y_est=w_out'*Z;

% Figure du résultat
figure
hold on;
plot(1:n,x);
plot(1:n,y,'r');
plot(1:n,y_est,'k')
```

6.3. Exercices

EXERCICE 14 (Réseau "echo-state"). Le but de l'exercice est de reprendre le code du Listing 6.1, en Matlab ou en Python, et de tester :

- d'autres fonctions d'activation `sigma`, dans le but notamment de mettre en évidence l'apport de la non-linéarité à la fois en terme de performance mais aussi de stabilité ;
- d'autres valeurs du rayon spectral `r`, afin de découvrir les limitations de stabilité lorsque $r > 1$, à mettre en phase avec le choix de $\sigma(\cdot)$;

— d'autres problèmes de prédiction, notamment avec un pas `shift` de prédiction plus ou moins élevé, ainsi que pour des périodes d'apprentissage plus ou moins longues ;
— des problèmes non plus de prédiction, mais de mémoire, en changeant notamment "`-shift`" en "`+shift`" dans la valeur de la sortie `y` : en déduire notamment ici les capacités de mémoire du réseau en fonction des paramètres choisis ;
— d'autres structures de matrices aléatoires `w_in` et surtout `W_res`. Il pourra notamment être pertinent de prendre W_{res} unitaire de sorte que toutes ses valeurs propres soient de même module. On pourra également tester le modèle intéressant d'une structure "de mémoire à plusieurs profondeurs" où $W_{\text{res}} = \text{BlockDiag}(W_1, \dots, W_m)$ où chaque W_i admet un rayon spectral différent (ceci engendre plusieurs blocs mémoriels dédiés chacun à des tâches de mémorisation plus ou moins longues).

Rendre compte des résultats obtenus afin de comprendre les apports et limitations des réseaux "echo-state" pour la prédiction et la mémorisation.

Exercice 15 (Réseau "echo-state" multivarié). Généraliser l'étude théorique ainsi que le code du Listing 6.1 à des données d'entrées et de sorties multivariées. On pourra tester les capacités de prédictions de ces réseaux sur des séries temporelles multivariées telles que des cours boursiers sur un portefeuille de taille réglable.

Exercice 16 (Jouer avec les réseaux "echo-state"). Êtes-vous capable de générer une séquence binaire (une suite de 0 et de 1) de manière totalement aléatoire ? Pour le vérifier, on peut "s'amuser" à implémenter un réseau echo-state qu'on apprendra à anticiper le prochain bit. On effectue pour cela une première étape de flush-out avec n données binaires Bernoulli indépendantes en entrée. Ensuite, ajouter successivement à la séquence un nouveau bit qu'on cherchera à être le plus "aléatoire" possible et regénérer le vecteur de read-out w_{out} calculé à partir de cette donnée et des $n-1$ précédentes. Continuer ainsi en faisant "glisser la fenêtre d'apprentissage" afin qu'elle reste de taille n. À chaque nouveau bit émis, évaluer les performances du réseau à anticiper le prochain bit.

Ce jeu peut se jouer à deux : l'objectif de chaque joueur est double (i) tromper le réseau de l'adversaire en lui faisant croire que votre séquence binaire est vraiment aléatoire, (ii) générer le réseau le plus efficace possible pour anéantir votre adversaire ! Dans l'esprit, on est ici en train de construire un réseau de neurones antagoniste (de type GAN), mais l'un des deux réseaux de neurones est humain !

Noter au passage qu'un ESN bien construit gagnera très vite la partie : en effet, rappelons par exemple que sur des séries de cinq bits consécutifs (du type 01001), une séquence sur 32 doit être la séquence 00000, une séquence sur 32 la séquence 11111, etc., que nous avons du mal à considérer comme réellement aléatoire...

Chapitre 7

Lectures complémentaires

Parmi les lectures standard en apprentissage automatique, les références suivantes pourront fournir un bon complément du cours :

— Un ouvrage très visuel et très élémentaire qui traite d'apprentissage avec un regard plutôt tourné vers les petites dimensions. Mathématiquement très léger, il cherche surtout à introduire les notions et méthodes de base de la théorie, avec un éclairage intuitif plus que théorique :

Bishop, Christopher M. Pattern recognition and machine learning. springer, 2006.

— Beaucoup plus théorique, cet ouvrage s'adresse à ceux qui cherchent à comprendre les fondamentaux théorique de l'apprentissage statistique (notons que le titre fait référence à l'apprentissage *statistique* plutôt que *machine* pour bien mettre l'accent sur son approche théorique). Cet ouvrage fait référence dans le domaine :

Friedman, Jerome, Trevor Hastie, and Robert Tibshirani. The elements of statistical learning. Vol. 1. No. 10. New York : Springer series in statistics, 2001.

— Avec une approche orientée "noyaux", cet ouvrage décrit également en profondeur la théorie de l'apprentissage, avec ici aussi une vision statistique et mathématique plus prononcée :

Schlkopf, Bernhard, Alexander J. Smola, and Francis Bach. Learning with kernels : support vector machines, regularization, optimization, and beyond. the MIT Press, 2018.

Chapitre 8

TP1 : Classification supervisée par SVM

Le TP a été construit pour laisser un maximum de liberté à l'étudiant. Il est ainsi attendu que le compte-rendu (i) réponde non seulement aux questions posées, mais surtout (ii) démontre une autonomie et une liberté exploratoire prise par l'étudiant. Le rapport devra être élaboré à la manière d'un article scientifique, avec une introduction claire de l'objectif, des outils, et des résultats ; ces derniers résultats seront impérativement et intelligemment commentés en regard aux notions vues en cours.

L'objectif de ce premier TP est double : (i) compléter l'étude des machines à vecteur support par une implémentation concrète et moderne (qui ne soit pas de type "boîte noire") et (ii) l'exploitation concrète et l'utilisation pratique de machines à vecteurs support.

La première partie (item (i)) prend une forme très théorique : ainsi, **il est conseillé d'anticiper la préparation de ce TP à la maison**. La seconde partie (item (ii)) est plus expérimentale et réalisable en majeure partie pendant le cours. Il est possible de commencer par la partie (ii) si on se sent plus à l'aise avec l'implémentation. Cependant, une moitié (la plus importante) de la partie (ii) utilise les notions développées en (i).

8.1. Préliminaires sur le problème d'optimisation SVM

Rappelons tout d'abord que, pour des couples d'observations $(x_1, y_1), \dots, (x_n, y_n) \in \mathbb{R}^p \times \{-1, 1\}$, en écrivant l'équation des hyperplans séparateurs solutions du problème SVM sous la forme $\mathcal{H}_q : z \mapsto w^\mathsf{T} z + b = q$ (pour $q \in \{0, 1, -1\}$), le problème d'optimisation à résoudre est le suivant :

$$\text{argmin}_{(w,b)} \frac{1}{2}\|w\|^2 + \frac{1}{n}\sum_{i=1}^{n} h\left(1 - y_i(w^\mathsf{T} x_i + b)\right)$$

où, selon que l'on résout un SVM "dur" ou "souple", $h(t)$ est la fonction

$$h(t) = \begin{cases} \iota_{\{t \leq 0\}}(t) \\ \frac{1}{\lambda}\max\{0, t\} \end{cases}$$

pour un certain $\lambda > 0$.

Nous allons commencer l'exercice par une simplification du problème d'optimisation.

Question 1.1. En définissant les vecteurs étendus

$$\tilde{w} = \begin{bmatrix} w \\ b \end{bmatrix} \in \mathbb{R}^{p+1}$$

$$\tilde{x}_i = \begin{bmatrix} x_i \\ 1 \end{bmatrix} \in \mathbb{R}^{p+1}$$

ainsi que la matrice diagonale de sélection

$$P_w = \text{diag}(1, \dots, 1, 0) \in \mathbb{R}^{(p+1)\times(p+1)}$$

montrer tout d'abord que le problème d'optimisation est équivalent au problème à un seul vecteur :

$$\operatorname{argmin}_{\tilde{w}\in\mathbb{R}^{p+1}} \frac{1}{2}\|P_w\tilde{w}\|^2 + \frac{1}{n}\sum_{i=1}^{n} h\left(1 - y_i\tilde{w}^\mathsf{T}\tilde{x}_i\right).$$

On se trouve ainsi avec la somme de deux fonctions de $\tilde{w}$, toutes deux convexes, mais l'une *lisse* (en fait quadratique) et l'autre *non différentiable* : ceci rend l'optimisation par descente de gradient impossible. On va donc utiliser des outils plus modernes, directement dans le problème primal, et basés sur les méthodes dites "proximales". On implémentera ici l'approche dite du *forward-backward splitting* issue du théorème suivant.

THÉORÈME 12 ("Forward-backward splitting"). *Soient $f_1, f_2 : \mathbb{R}^q \to \mathbb{R}$ deux fonctions convexes, f_2 étant différentiable et de gradient Lipschitz (à savoir $\|\nabla f_2(z) - \nabla f_2(z')\| \leq L\|z - z'\|$ pour un certain L indépendant de z, z'). Alors, si on définit, pour $x_1 \in \mathbb{R}^q$ arbitraire et pour tout $k \geq 1$*

$$x_{k+1} = \operatorname{prox}_{\gamma f_1}\left(x_k - \gamma\nabla f_2(x_k)\right)$$

où $\gamma > 0$ est arbitraire, il vient que $x_k \overset{k\to\infty}{\longrightarrow} x^\star$ pour un certain

$$x^\star \in \operatorname{argmin}_{x\in\mathbb{R}^q}\left\{f_1(x) + f_2(x)\right\}.$$

Ici, l'opérateur prox_h est l'opérateur "proximal" défini par

$$\operatorname{prox}_f(x) = \operatorname{argmin}_{y\in\mathbb{R}^q}\left\{f(y) + \frac{1}{2}\|x - y\|^2\right\}.$$

On connait en particulier un certain nombre d'opérateurs proximinaux donnés par la table suivante

f	$\operatorname{prox}_f(x)$
0	x
$\lambda\|x\|_1$	$\{\operatorname{sgn}([x]_i)\max(\|[x]_i\| - \lambda, 0))\}_{i=1}^q$
$\imath_{\{Ax=y\}}(x)$	$x + A^\mathsf{T}(AA^\mathsf{T})^{-1}(y - Ax)$
$\imath_{\{x\geq 0\}}(x)$	$\max(0, x)$
$\frac{1}{2}\|Ax - y\|^2$	$(I_q + A^\mathsf{T}A)^{-1}(x + A^\mathsf{T}y)$
$x^\mathsf{T}A^\mathsf{T}y$	$x - A^\mathsf{T}y$
$\frac{1}{2}x^\mathsf{T}Ax$	$(I_q + A)^{-1}x$
$\sum_{i=1}^q \max(0, -\alpha[x]_i)$	$\left\{[x]_i 1_{[x]_i\geq 0} + ([x]_i - \alpha)1_{[x]_i\leq\alpha}\right\}_{i=1}^q$.

Le théorème annonce donc qu'il est possible de résoudre un problème d'optimisation convexe de la forme $\min_x f_1(x) + f_2(x)$ *même si f_1 n'est pas différentiable*, ce qui est le cas pour nous avec la fonction h. Pour cela, il suffit d'itérer successivement les opérations (i) $\tilde{x}_k = x_k - \gamma\nabla f_2(x_k)$ (opérateur "forward" de descente de gradient avec un pas $\gamma > 0$) et (ii) $x_{k+1} = \operatorname{prox}_{\gamma f_1}(\tilde{x}_k)$ (opérateur dite "backward" équivalente à une descente de gradient pour la fonction non différentiable f_1), et ce jusqu'à convergence (garantie pour tout $\gamma > 0$).

Pour pouvoir appliquer le théorème et utiliser un opérateur proximal connu (ils ne sont pas forcément tous faciles à calculer selon la complexité de la fonction f_1 !), on opère maintenant le changement de variable

$$\zeta = \begin{bmatrix} y_1\tilde{x}_1^\mathsf{T}\tilde{w} - 1 \\ \vdots \\ y_n\tilde{x}_n^\mathsf{T}\tilde{w} - 1 \end{bmatrix} = D_y\tilde{X}^\mathsf{T}\tilde{w} - 1_n$$

où $D_y = \operatorname{diag}(y_1, \ldots, y_n)$ et $\tilde{X} = [\tilde{x}_1, \ldots, \tilde{x}_n] \in \mathbb{R}^{(p+1)\times n}$.

Question 1.2. Montrer que le problème d'optimisation peut se réécrire dès lors sous la forme

$$\min_{\zeta\in\mathbb{R}^n}(\zeta+1_n)^\mathsf{T} D_y\tilde{X}^\mathsf{T}\left(\tilde{X}\tilde{X}^\mathsf{T}\right)^{-1}P_w\left(\tilde{X}\tilde{X}^\mathsf{T}\right)^{-1}\tilde{X}D_y(\zeta+1_n)+\frac{1}{n}\sum_{i=1}^n h(-[\zeta]_i).$$

En définissant alors

$$f_1(\zeta)=\frac{1}{n}\sum_{i=1}^n h(-[\zeta]_i)$$

$$f_2(\zeta)=(\zeta+1_n)^\mathsf{T} D_y\tilde{X}^\mathsf{T}\left(\tilde{X}\tilde{X}^\mathsf{T}\right)^{-1}P_w\left(\tilde{X}\tilde{X}^\mathsf{T}\right)^{-1}\tilde{X}D_y(\zeta+1_n)$$

évaluer :

— le gradient $\nabla f_2(\zeta)$

— la fonction $\mathrm{prox}_{\gamma f_1}(\zeta)$ pour un $\gamma>0$ arbitraire (se référer pour cela à la table précédente et la définition de l'opérateur proximal).

Question 1.3 (bonus). Redémontrer les expressions des opérateurs proximaux pour les fonctions $\sum_i \max(0,-\alpha[x]_i)$ et $\imath_{\{Ax=b\}}(x)$ données dans le tableau du Théorème 12.

8.2. Implémentation

8.2.1. Les données. Pour rendre l'exercice concret, nous allons classer des données réelles $Z=[z_1,\dots,z_n]$ vivant dans un espace ambiant $\mathbb{R}^q$. Cependant, pour des raisons de visualisation, nous allons commencer par projeter ces données dans $\mathbb{R}^2$ et nous travaillerons donc ici avec les données projetées $X=[x_1,\dots,x_n]$ de taille $p=2$. Pour cela, on va tout d'abord procéder à une réduction de dimension par analyse en composantes principales. Si $Z\in\mathbb{R}^{q\times n}$ est l'ensemble des données réelles, on posera alors

$$X=U^\mathsf{T} Z$$

où $U=[u_1,u_2]\in\mathbb{R}^{q\times p}$ est la résultante de la décomposition en valeurs singulières $Z=\sum_{i=1}^q \omega_i u_i v_i^\mathsf{T}$, avec $\omega_1\geq\ldots\geq\omega_q$.

Question 2.1. Nous considèrerons dans un premier temps pour base de données la populaire base MNIST (base de chiffres manuscrits sur $28\times 28=784$ pixels), jointe au TP mais que des modules Python peuvent automatiquement charger, en utilisant par exemple :

```
1 from sklearn.datasets import fetch_mldata
2 mnist = fetch_mldata('MNIST original')
```

Extraire de cette base de données $n_1=n_2=100$ images appartenant à deux classes distinctes (par exemple des 1 et des 7, plutôt difficiles à classer) que l'on vectorisera en dimension $q=784$. Procéder à la réduction de dimension et construire ainsi X. Pour assurer le bon fonctionnement de la réduction de dimension, observer le rendu de la réduction de dimension en comparant les z_i aux vecteurs réduits

$$\tilde{z}_i=Ux_i$$

que l'on remettra en forme de 28×28 pixels (ici, comme $\tilde{z}_i=Ux_i=UU^\mathsf{T} z_i$ où UU^T est un *projecteur* sur l'espace réduit, on retrouve bien une version "dégradée" de z_i).

Diviser alors les données en données d'entrainement et données de test (par exemple $n_1/2=n_2/2=50$ données d'entrainement et $n_1/2=n_2/2=50$ données de test).

8.2.2. Classification. Cette deuxième partie concerne l'implémentation à proprement parler de la méthode SVM, d'une part grâce aux bibliothèques Matlab ou Python qu'il s'agit ici de découvrir, et d'autre part en utilisant l'étude menée précédemment.

Question 2.2. Implémenter de deux manières les classifieurs SVM dur et doux sur le jeu de données ci-dessus :

(1) dans un premier temps, en utilisant les routines "black box" des modules Matlab ou Python ;

(2) dans un second temps, en utilisant l'implémentation par la méthode proximale étudiée dans la partie précédente. Pour cette seconde méthode, on prendra comme critère d'arrêt de l'algorithme un seuil sur la distance entre deux itérées successives du vecteur ζ ; à partir de ζ, on reconstruira alors la solution (w, b).

Représenter ensuite dans un plan en $p = 2$ dimensions l'ensemble des données x_i ainsi que l'hyperplan séparateur obtenu par chacun des algorithmes. Comparer les résultats des deux algorithmes. Estimer empiriquement le taux de classification correcte.

Question 2.3. Procéder de même avec d'autres paires de données et d'autres tailles pour les ensembles d'entrainement et de test. Généraliser ensuite à des projections des données x_i de dimension $p > 2$ et observer les gains en performance : commenter.

Chapitre 9

TP2 : Prédiction de séries temporelles financières

Le TP a été construit pour laisser un maximum de liberté à l'étudiant. Il est ainsi attendu que le compte-rendu (i) réponde non seulement aux questions posées, mais surtout (ii) démontre une autonomie et une liberté exploratoire prise par l'étudiant. Le rapport devra être élaboré à la manière d'un article scientifique, avec une introduction claire de l'objectif, des outils, et des résultats ; ces derniers résultats seront impérativement et intelligemment commentés en regard aux notions vues en cours.

L'objectif de cet exercice consiste à prédire l'évolution de séries temporelles de cours de la bourse à l'aide de réseaux de neurones de type ESN.

9.1. Les données

Pour cela, vous avez accès à plusieurs bases de données de cours boursiers, sous la forme de rentabilités journalières (*log-returns*) :

— l'indice HSI-50 (Heng Seng, HongKong) de 2011 à 2013
— l'indice NYSE-148 (New York Stock Exchange, USA) de 2010 à 2013
— l'indice S&P-500 (Standard and Poor, USA) de 2010 à 2013
— l'indice S&P-100 (Standard and Poor, USA) de 2011 à 2013

Ces données sont stockées au sein de fichiers Matlab (`.mat`) qu'il est possible de charger sous Matlab via la fonction `load`, ou en Python via la commande

```
1 import scipy.io
2 mat = scipy.io.loadmat('file.mat')
```

Une fois chargées, ces données prennent la forme d'une matrice `Y` de dimension $p \times n$ où p est le nombre d'actifs (par exemple $p = 50$ pour HSI-50) et où n est le nombre de jours d'enregistrement de la série temporelle.

On travaillera dans un premier temps sur *un seul actif* (à choisir parmi les p actifs du portefeuille), avant de généraliser l'étude à un nombre arbitraire d'actif. Il est donc bienvenu de concevoir le code en anticipation de cette généralisation.

Vous avez bien sûr la possibilité de récupérer vous-mêmes des cours de votre choix, plus récents, par exemple du CAC-40 sur ces dernières années (voire même incluant les derniers jours). On peut également choisir de travailler sur des séries temporelles à différentes échelles de temps (évolution minute-par-minute, ou au contraire mois-par-mois).

Étant données les variations rapides très difficiles à anticiper de certains cours, il sera pertinent, pour commencer, de travailler avec des données simplifiées, à variations lentes. Une façon d'y parvenir et d'effectuer un lissage des données comme proposé dans la remarque suivante.

Remarque 13 (Lissage des données). Il est recommandé, au moins dans un premier temps, de débuter l'exercice sur des données "lissées" à variations plus lentes, et donc plus simples à gérer par un ESN. Pour ce faire, on peut appliquer

une fenêtre glissante de moyennage des données d'une durée L. Si $x \in \mathbb{R}^n$ est une série temporelle, on pourra utiliser la formule

$$x_L = Tx, \quad T = \frac{1}{L}\left\{1_{|i-j|\leq L}\right\}_{i,j=1}^{n}$$

pour obtenir une version lissée $x' \in \mathbb{R}^n$ de $x \in \mathbb{R}^n$. La matrice T est une matrice de Toeplitz qui peut être construite facilement via des routines Matlab (`toeplitz`) ou Python (`scipy.linalg.toeplitz`) appropriées. Pour simplifier les choses, on ignorera ici les effets de bord.

On pourra commencer avec un moyennage sur deux mois ($L = 60$).

9.2. L'objectif

9.2.1. Régression. L'objectif du TP est d'anticiper les variations à venir du marché à partir des données x (ou x_L) observées à l'aide d'un réseau de neurone echo-state qu'il s'agit donc de construire et configurer. Une entière liberté vous est laissée quant au choix de vos paramètres, mais il est fortement conseillé de débuter avec un réseau stable (reprendre par exemple l'exemple du cours) que l'on testera au préalable sur un exemple synthétique simple (comme par exemple la prédiction d'une fonction lisse élémentaire ; voir ici aussi l'exemple du cours).

Si $x \in \mathbb{R}^n$ est l'entrée du réseau, on souhaitera donc que la sortie $y \in \mathbb{R}^n$ corresponde à une version de x décalée dans le temps :

$$y_i = x_{i+\text{shift}}$$

où shift est un paramètre que l'on souhaitera le plus grand possible. En prenant en compte le lissage proposé dans la Remarque 13, shift devra être de l'ordre de grandeur de L pour que le problème reste pertinent.

Il s'agit donc ici bien d'un exercice de régression et non de classification.

9.2.2. Apprentissage sous hypothèse stationnaire. Dans un premier temps, on fera l'hypothèse, comme dans le code du Listing 6.1 d'un comportement stationnaire de la série temporelle. Dans ce cas, il est suffisant d'effectuer un apprentissage en 3 étapes, comme suit :

(1) dans un premier temps, on considère un temps T_{flush} (on pourra prendre $T_{\text{flush}} \sim 100$) de remise à zéro (*flushout*) du réseau où l'état du réservoir, appelons le $z_t \in \mathbb{R}^N$ pour un réseau de N neurones au temps t, est mis à jour récursivement sans opérer de régression linéaire à la sortie ;

(2) dans un second temps, l'entraînement, d'une durée T_{train} (on pourra prendre $T_{\text{train}} \sim 200$ ou plus), on effectuera l'apprentissage par régression linéaire régularisée, où le paramètre γ de régularisation sera au départ pris assez petit (et ce afin de bien contrôler le comportement de l'ESN) : cette opération produira le vecteur de régression $w_{\text{out}} \in \mathbb{R}^N$ (généralisé à une matrice de régression dans la fin de l'exercice) ;

(3) enfin, dans un troisième temps, la généralisation, qui durera jusqu'à la fin de la série temporelle (imputée bien sûr pour y des shift derniers jours), estimera la valeur de y en opérant le filtre w_{out} précédemment construit sur les évolutions du réservoir.

En somme, cette partie du TP est fortement liée au code du Listing 6.1. Les seules difficultés de cette partie consistent en une attention toute particulière à prêter aux indices des vecteurs et matrices : les différents paramètres shift, T_{flush}, T_{learn}, ainsi que les effets de bords (par exemple, y_i n'existe que pour $i \leq n - \text{shift}$) sont autant de paramètres de confusions possible. Il est fortement recommandé de poser sur papier la chronologie temporelle des différentes actions effectuées pour

éviter : (i) des erreurs d'indexation dans le code, ou pire (ii) des codes donnant des résultats triviaux (soit complètement inopérants, soit utilisant illégalement des informations a priori non connues).

Question 1. Implémenter un ESN suivant la méthode de prédiction proposée ci-dessus et régler finement ses paramètres afin d'obtenir des résultats corrects. Reporter et commenter dans votre rapport :

- le choix des paramètres et du réseau implémenté ;
- les performances obtenues en termes d'erreurs *relatives* moyennes : (i) pour différents choix de L et shift (on tentera de s'approcher au plus près d'une prédiction au jour le jour !), (ii) pour différentes configurations du réseau ;
- les courbes comparatives des données d'entrées, de sorties et des données estimées.

Qu'observe-t-on lorsque l'on s'éloigne trop de la zone d'entraînement ? Que conclure de la stationnarité présumée de la série temporelle ? Comment améliorer les performances ? (en dehors l'idée proposée dans la suite)

9.2.3. Apprentissage par fenêtre glissante. Dans cette section, pour combattre les problèmes de stationnarité observés précédents, on souhaite modifier l'implémentation proposée en Listing 6.1 en rendant l'apprentissage "dynamique". Au lieu de fixer les données d'entraînement à une zone temporelle fixe (de T_{flush} à $T_{\text{flush}} + T_{\text{train}}$), nous allons plutôt "faire glisser" la fenêtre d'apprentissage jusqu'au dernier jour connu, et ce jour après jour.

Pour cela, nous réduisons maintenant l'apprentissage à 2 étapes, comme suit :

(1) comme précédemment, dans un premier temps, on effectue un nettoyage du réseau de $t = 1$ à $t = T_{\text{flush}}$, en conservant les mêmes paramètres que précédemment ; ici rien ne change ;

(2) ensuite, et c'est ici que les choses changent, pour chaque pas de temps $t \geq T_{\text{flush}}$, on utilise les T_{train} derniers indices de temps pour construire le vecteur de régression $w_{\text{out}}(t) \in \mathbb{R}^N$, qui dépend donc désormais de t ; ce vecteur est alors utilisé pour effectuer la prédiction de y_t. Attention ici à être particulièrement vigilant à l'indexation des vecteurs (notamment x et y) ! Un décalage d'une unité de temps pourrait donner lieu à des résultats triviaux.

Question 2. Implémenter l'ESN proposé en l'intégrant dans le même code que précédemment (afin de garder les mêmes paramètres) via un choix d'implémentation (`switch` en Matlab ou en utilisant une variable dictionnaire en Python). Comme précédemment, reporter les performances et détailler/justifier le choix des paramètres. Qu'observe-t-on désormais des performances ?

9.2.4. Généralisation à plusieurs actifs. Généraliser l'implémentation précédente à un nombre D d'actifs. Les vecteurs $x \in \mathbb{R}^n$, $y \in \mathbb{R}^n$, $w_{\text{in}} \in \mathbb{R}^N$ et $w_{\text{out}} \in \mathbb{R}^N$ deviennent désormais des matrices de tailles $D \times n$ et $D \times N$, respectivement. On pourra notamment s'inspirer de l'Exercice 15 pour appliquer la généralisation (qui n'est pas bien difficile).

Question 3. Implémenter l'ESN proposé en généralisation le code développé précédemment et en permettant en particulier à l'utilisateur de choisir le sous-ensemble des D actifs de la base de données. Tenter de montrer en particulier que, en ajoutant séquentiellement des actifs au jeu de données (en augmentant D donc), la performance de l'ESN sur les actifs communs tend à s'améliorer. À quoi imputer ce comportement ? Comme précédemment, le "jeu" consiste à tenter de s'approcher dans la mesure du possible d'une prédiction au jour le jour (ce qui est évidemment difficile en pratique !).

Conclure le TP de toutes les remarques d'implémentation, de considérations pratiques, en lien avec l'apprentissage mais aussi en lien avec les séries temporelles traitées ici qui vous paraissent pertinentes. On s'attardera notamment à discuter (et si possible à implémenter) des idées d'améliorations de l'algorithme.