Université de Montréal

Modèles à noyaux à structure locale

par

Pascal Vincent

Département d'informatique et de recherche opérationnelle

Faculté des arts et des sciences

Thèse présentée à la Faculté des études supérieures

en vue de l'obtention du grade de

Philosophiæ Doctor (Ph.D.)

en informatique

Octobre 2003

Université de Montréal

Faculté des études supérieures

Cette thèse intitulée:

Modèles à noyaux à structure locale

présentée par:

Pascal Vincent

a été évaluée par un jury composé des personnes suivantes:

Balázs Kégl

président-rapporteur

Yoshua Bengio

directeur de recherche

Jean-Jules Brault

membre du jury

Sam Roweis

examinateur externe

Lael Parrott

représentant du doyen de la FES

Résumé

La plupart des problèmes concrets auxquels on souhaite appliquer les algorithmes d'apprentissage apparaissent en dimension élevée. Or le "fléau de la dimensionalité" pose un défi pour obtenir de bonnes performances. Aussi le succès des Machines à Vecteurs de Support (SVMs) à noyaux, particulièrement sur des problèmes en haute dimension, a engendré un regain d'intérêt pour les méthodes à noyaux. Cette thèse propose trois algorithmes à noyaux, pour l'essentiel des extensions d'algorithmes classiques permettant de grandement améliorer leur performance en haute dimension, et de surpasser les SVMs. Ce faisant, nous améliorons également notre compréhension des caractéristiques des problèmes en haute dimension. La première partie de l'ouvrage est une introduction au domaine de l'apprentissage statistique. La seconde partie présente les algorithmes à noyaux les plus connus. Dans la troisième partie nous présentons notre recherche, au travers de trois articles. Enfin la quatrième partie effectue une synthèse et suggère des pistes pour aller plus loin. Le premier article, *Kernel Matching Pursuit*, définit un algorithme constructif donnant lieu à une solution ayant la même forme que les SVMs mais permettant un strict contrôle du nombre de points de support et donnant lieu à des solutions davantage clairsemées que les SVMs. Le second

article, *K-Local Hyperplane and Convex Distance Nearest Neighbor Algorithms*, propose une variante de l'algorithme des plus proches voisins, en redéfinissant la distance d'un point à une classe comme la distance à la variété linéaire supportée par les voisins de ce point. Cette extension permet de surpasser les SVMs sur des problèmes concrets en haute dimension. Le troisième article, *Manifold Parzen Windows*, étudie une variante de l'estimateur de densité classique de Parzen. En utilisant des Gaussiennes aplaties orientées selon les directions principales apparaissant dans les données du voisinage, on peut mieux représenter une densité concentrée le long d'une variété non linéaire de plus faible dimension, ce qui s'avère profitable en haute dimension. La principale conclusion de ces travaux est double. D'une part ils montrent que des algorithmes d'inspiration non paramétrique classique, qui ne font aucunement appel à "l'astuce du noyau", sont capables de performance aussi bonne, voire supérieure, à celle des SVMs. D'autre part ils montrent qu'en haute dimension il y a beaucoup à gagner à développer des algorithmes sachant tirer partie de l'hypothèse selon laquelle les données seraient davantage concentrées le long de variétés non linéaires de faible dimension. Ceci constitue un espoir pour battre le fléau de la dimensionalité.

Mots-clés : méthodes à noyaux, statistiques non paramétriques, fléau de la dimensionalité, Machines à Vecteurs de Support, solutions clairsemées, k plus proches voisins, fenêtres de Parzen.

Abstract

Most real world problems, for which one wishes to apply machine learning techniques, appear in high dimensional spaces where the "curse of dimensionality" poses a serious challenge. Thus the success of kernelized Support Vector Machines (SVMs) on a number of high dimensional tasks has prompted a renewed interest in kernel methods. In this thesis, we propose three alternative kernel methods, mostly extensions of classical algorithms, with greatly improved performance in high dimension, that are able to outperform SVMs. In the process, we also increase our understanding of important characteristics of high dimensional problems. Part one of the document is a general introduction to machine learning. Part two describes the most common kernel methods. In part three, we present our research through three published articles. Finally, part four provides a synthesis of our contribution, and hints to possible future developments. The first article, *Kernel Matching Pursuit*, defines a constructive algorithm that leads to solutions of the same form as SVMs, but allows a strict control over the number of support vectors, leading to much sparser solutions. The second article, *K-Local Hyperplane and Convex Distance Nearest Neighbor Algorithms*, is a variation of the nearest neighbors algorithm in which we define the distance between a point and

a class as the distance to the linear sub-manifold supported by the neighbors of that point. This extension allows to outperform SVMs on a number of high dimensional problems. The third article, *Manifold Parzen Windows*, studies an extension of the classical Parzen density estimator, in which we use flattened Gaussian pancakes oriented along the principal directions appearing in the neighborhood of a point. This allows to better capture a density when it is concentrated along a lower dimensional non linear manifold, which proves useful in high dimension. The important contribution of our research is twofold. First it shows that algorithms of classical non-parametric inspiration, that do not call upon the "kernel trick" in any way, are able to perform as well or better than SVMs. Second, our results indicate that in high dimension, much is to be gained by designing algorithms that take into account the "manifold hypothesis", i.e. that data might be more concentrated along lower dimensional non linear sub-manifolds. This is so far the best hope we have to beat the curse of dimensionality.

Keywords : kernel methods, non-parametric statistics, curse of dimensionality, Support Vector Machines, sparsity, k nearest neighbors, Parzen windows.

Table des matières

Liste des tableaux

Liste des figures

Liste des abréviations

I.A.	Intelligence Artificielle
c.a.d.	c'est à dire
p.d.f.	Fonction de densité de probabilité (*probability density function*).
c.d.f.	Fonction cumulative (*cumulative density function*)
i.i.d.	Indépendantes et identiquement distribuées
KNN	L'algorithme des k plus proches voisins (*K Nearest Neighbors*)
SVM	L'algorithme des machines à vecteurs de support (*Support Vector Machines*)
KMP	L'algorithme *Kernel Matching Pursuit* présenté dans le premier article
HKNN	L'algorithme *K-Local Hyperplane Distance Nearest Neighbor Algorithms* présenté dans le deuxième article
CKNN	L'algorithme *K-Local Convex Distance Nearest Neighbor Algorithms* présenté dans le deuxième article
EM	L'algorithme *Expectation Maximisation*
PCA ou ACP	Analyse en Composantes Principales
SVD	Décomposition en valeurs singulières

Notations mathématiques

$\mathbb{R}$	Ensemble des nombres réels.
$\mathbb{R}^+$	Ensemble des nombres réels positifs ou nuls.
$\mathcal{N}(\mu, \sigma)$	Avec $\sigma \in \mathbb{R}$ distribution gaussienne normale centrée en μ et de variance σ^2.
$\mathcal{N}(\mu, C)$	Avec C matrice, distribution gaussienne normale centrée en μ et de matrice de covariance C.
$\|x\|$	Valeur absolue de x.
$\|\|x\|\|$	Norme L_2 de x.
$< x, y >$	Produit scalaire usuel entre x et y.
$X \sim P(X)$	Variable aléatoire X de distribution ou densité $P(X)$.
$E[f(X)]_{X \sim P(X)}$	Espérence de $f(X)$ pour X tiré selon $P(X)$.
$V[f(X)]_{X \sim P(X)}$	Variance (si $f(X)$ scalaire) ou matrice de covariance (si $f(X)$ vectoriel) pour X tiré selon $P(X)$.

A mes parents, pour leur soutien indéfectible…

Remerciements

Je tiens à remercier mon directeur de recherches Yoshua Bengio, pour m'avoir accueilli et guidé toute ces années au travers des joies, des difficultés et des doutes de la carrière de chercheur, pour sa confiance, et surtout pour avoir su partager sa passion et communiquer son enthousiasme pour ce merveilleux domaine de recherches.

Je remercie également tous les étudiants et chercheurs qui sont passés par le laboratoire Lisa, tout particulièrement ceux d'entre eux qui ont été contraint de vivre avec mes imparfaites créations logicielles, pour leur patience et leur amitié.

Introduction générale

Notre faculté d'apprendre est ce qui nous permet de constamment nous adapter à notre environnement changeant, de grandir et de progresser. C'est en grande partie à elle que l'humanité doit sa survie et ses plus grands succès.

L'apprentissage a toujours été le fer de lance de la branche Connexioniste de l'Intelligence Artificielle, et les algorithmes et techniques qu'elle a su développer au fil des années, trouvent de nos jours des applications pratiques dans un nombre grandissant de domaines : de la reconnaissance de la parole ou de l'écriture, à la finance et au projet de génome humain. Mais même si ses succès récents sont impressionnants, en comparaison avec les facultés humaines, nous n'en sommes qu'aux balbutiements. De nombreux mystères demeurent, et le secteur des algorithmes d'apprentissage demeure un domaine de recherche très actif.

Dans cette thèse nous nous attardons sur les algorithmes d'apprentissage à noyaux, et présentons, au travers de trois articles, notre contribution à ce domaine qui connaît depuis quelques années un regain d'intérêt, avec l'espoir de parvenir à des algorithmes performants en haute dimension.

L’ouvrage est divisé en quatre parties. La première partie est essentiellement une présentation du paradigme de l’apprentissage et un survol du domaine. Son but premier est de familiariser le lecteur, qui ne le serait pas déjà, avec les concepts essentiels du domaine et avec sa terminologie. La seconde partie dresse un tableau des algorithmes à noyaux les plus connus, et passe brièvement en revue un certain nombre de travaux antérieurs. La troisième partie présente et inclue trois articles que nous avons contribués à la recherche sur les modèles à noyaux. Enfin la quatrième partie tente une synthèse, discute des mérites et des limites de ce que nous avons proposé, et suggère des pistes pour aller plus loin.

Première partie

Introduction

Chapitre 1

Présentation du domaine de l'apprentissage automatique

1.1 Qu'est-ce que l'apprentissage automatique

La faculté d'apprendre de ses expériences passées et de s'adapter est une caractéristique essentielle des formes de vies supérieures. Elle est essentielle à l'être humain dans les premières étapes de la vie pour apprendre des choses aussi fondamentales que reconnaître une voix, un visage familier, apprendre à comprendre ce qui est dit, à marcher et à parler.

L'apprentissage automatique[1] est une tentative de comprendre et reproduire cette faculté d'apprentissage dans des systèmes artificiels. Il s'agit, très schématiquement, de concevoir des algorithmes capables, à partir d'un nombre important

[1] Anglais : *Machine Learning*

d'exemples (les *données* correspondant à "l'expérience passée"), d'en assimiler la nature afin de pouvoir appliquer ce qu'ils ont ainsi appris aux cas futurs.

1.2 Situation historique multi-disciplinaire

Traditionnellement, on considère que le domaine de l'apprentissage automatique sub-symbolique est né vers la fin des années 50, comme une branche dissidente de l'Intelligence Artificielle classique, avec la publication des travaux de Rosenblatt sur le Perceptron [77].

Historiquement, c'est là le fruit de la rencontre de l'Intelligence Artificielle et des neuro-sciences. Ce qu'on a alors appelé la branche "connexioniste" de l'I.A. ambitionnait de parvenir à créer des machines capables d'intelligence en tentant de mimiquer le fonctionnement des systèmes nerveux biologiques, ou tout du moins en s'inspirant fortement des connaissances sur les réseaux de neurones biologiques, et présentait un départ radical de l'approche symbolique de logique "Aristotélicienne" adoptée par l'I.A. classique. Ainsi ont été développés les réseaux de neurones artificiels.

Dès sa naissance, le domaine était donc résolument inter-disciplinaire. Au cours des 45 années qui ont suivi, ce caractère n'a fait que s'accentuer, et si l'attrait pour la stricte inspiration biologique s'est beaucoup estompé (certains diront malheureusement), c'est avant tout parce que des connexions profondes ont été développées avec d'autres disciplines. En effet la formalisation du domaine, son mûrissement, la compréhension théorique accrue des problèmes impliqués, se sont accompagnés d'un rapprochement avec des disciplines ayant de solides fondations

mathématiques et théoriques telles que la théorie de l'information et le traitement du signal, l'optimisation non-linéaire, mais surtout et de façon prépondérante ces dernières années avec le point de vue statistique.

1.2.1 L'apprentissage automatique par rapport aux statistiques classiques

Du point de vue du problème de l'apprentissage, on peut diviser les statistiques classiques en deux branches :

– Les statistiques paramétriques, dont le cadre suppose que l'on connaît la forme du *vrai* modèle qui a généré les données, ignorant seulement ses paramètres, et où il s'agit d'estimer au mieux les paramètres du dit modèle à partir d'un échantillon de données fini.
– Les statistiques non paramétriques (k plus proches voisins, fenêtres de Parzen, . . . Voir la section 5.2). Là, la plupart des études statistiques s'intéressent aux propriétés de convergence et consistance de l'estimateur quand le nombre d'exemples tend vers l'infini.

Les recherches en apprentissage automatique se sont quant à elles concentrées davantage sur des problèmes réels complexes, où il serait absurde de croire que l'on puisse disposer du *vrai* modèle, et où l'on est également loin d'avoir une quantité illimitée de données. Bien que les statistiques classiques se soient un peu intéressées à ces questions, depuis l'avènement de l'informatique ce champ d'investigation a surtout été exploré par la communauté de l'apprentissage automatique. Par ses origines dans des domaines moins frappés de rigueur et de formalisme mathématique (la neuro-biologie et l'électronique/informatique), les recherches en

intelligence artificielle sub-symbolique ont pris un chemin davantage empirique, se satisfaisant très bien de produire des “monstres” mathématiques comme les réseaux de neurones, du moment qu’ils fonctionnaient et donnaient de bons résultats ! Dans la mesure où les modèles utilisés étaient plus complexes, les questions de sélection de modèle et du contrôle de leur capacité se sont imposées naturellement avec force.

Mais on voit que, bien plus qu’une différence de fond entre les deux domaines, ce qui les sépare est une différence de culture et d’emphase : les études statistiques classiques se sont souvent auto-limitées à des modèles se prêtant bien à une analyse mathématique (modèles assez simples, en faible dimension). En comparaison, la recherche en intelligence artificielle était résolument engagée sur la voie de la complexité, avec pour seule limite la capacité du matériel informatique, et poussée par le besoin de mettre au point des systèmes répondant aux problèmes concrets du moment.

Néanmoins, avec le temps, le domaine de l’apprentissage automatique a mûri, s’est formalisé, théorisé, et s’est ainsi inéluctablement rapproché des statistiques, au point d’être rebaptisé *apprentissage statistique*. Pour autant, bien que s’étant considérablement réduit, le fossé culturel n’est pas totalement comblé, notamment en ce qui concerne les conventions quant aux façons de procéder et à la terminologie. Nous espérons donc que le lecteur davantage familier avec le formalisme statistique saura pardonner l’approche certainement moins rigoureuse de l’auteur.

1.3 Les tâches de l'apprentissage

On peut séparer les *tâches* de l'apprentissage automatique en trois grandes familles :

– apprentissage supervisé,
– apprentissage non-supervisé,
– apprentissage par renforcement.

1.3.1 L'apprentissage supervisé

La formulation du problème de l'apprentissage supervisé est simple : on dispose d'un nombre fini d'exemples d'une tâche à réaliser, sous forme de paires *(entrée, sortie désirée)*, et on souhaite obtenir, d'une manière automatique, un système capable de trouver de façon relativement fiable la sortie correspondant à toute nouvelle entrée qui pourrait lui être présentée.

On distingue en général trois types de problèmes auxquels l'apprentissage supervisé est appliqué. Ces tâches diffèrent essentiellement par la nature des paires *(entrée, sortie)* qui y sont associées :

Classification

Dans les problèmes de classification, l'entrée correspond à une instance d'une classe, et la sortie qui y est associée indique la classe. Par exemple pour un problème de reconnaissance de visage, l'entrée serait l'image bitmap d'une personne

telle que fournie par une caméra, et la sortie indiquerait de quelle personne il s'agit (parmi l'ensemble de personnes que l'on souhaite voir le système reconnaître).

Régression

Dans les problèmes de régression, l'entrée n'est pas associée à une classe, mais dans le cas général, à une ou plusieurs valeurs réelles (un vecteur). Par exemple, pour une expérience de biochimie, on pourrait vouloir prédire le taux de réaction d'un organisme en fonction des taux de différentes substances qui lui sont administrées.

Séries temporelles

Dans les problèmes de séries temporelles, il s'agit typiquement de prédire les valeurs futures d'une certaine quantité connaissant ses valeurs passées ainsi que d'autres informations. Par exemple le rendement d'une action en bourse... Une différence importante avec les problèmes de régression ou de classification est que les données suivent typiquement une distribution non stationnaire.

1.3.2 L'apprentissage non-supervisé

Dans l'apprentissage *non supervisé* il n'y a pas de notion de sortie désirée, on dispose seulement d'un nombre fini de données d'apprentissage, constituées "d'entrées", sans qu'aucun label n'y soit rattaché.

Estimation de densité

Dans un problème d'estimation de densité, on cherche à modéliser convenablement la distribution des données. L'estimateur obtenu $\hat{f}(x)$ doit pouvoir donner un bon estimé de la densité de probabilité à un point de test x issu de la même distribution (inconnue) que les données d'apprentissage.

Partitionnement

Le problème du partitionnement[2] est le pendant non-supervisé de la classification. Un algorithme de partitionnement tente de partitionner l'espace d'entrée en un certain nombre de "classes" en se basant sur un ensemble d'apprentissage fini, ne contenant aucune information de classe explicite. Les critères utilisés pour décider si deux points devraient appartenir à la même classe ou à des classes différents sont spécifiques à chaque algorithme, mais sont très souvent liés à une mesure de distance entre points. L'exemple le plus classique d'algorithme de partitionnement est l'algorithme K-Means.

Réduction de dimensionalité

Le but d'un algorithme de réduction de dimensionalité est de parvenir à "résumer" l'information présente dans les coordonnées d'un point en haute dimension ($x \in \mathbb{R}^n$, n grand) par un nombre plus réduit de caractéristiques ($y = f(x), y \in \mathbb{R}^m, m < n$). Le but espéré est de préserver l'information "importante", de la mettre en évidence en la dissociant du bruit, et possiblement de révéler une struc-

[2] Anglais : *clustering*

ture sous-jacente qui ne serait pas immédiatement apparente dans les données d'origine en haute dimension. L'exemple le plus classique d'algorithme de réduction de dimensionalité est l'Analyse en Composantes Principales (ACP) [52].

1.3.3 L'apprentissage par renforcement

Nous ne faisons ici que mentionner très succinctement le cadre général de l'aprentissage par renforcement, ce domaine étant hors du champ de notre sujet. Nous invitons le lecteur désireux d'en savoir plus à se référer à [93]. La particularité et la difficulté du cadre de l'apprentissage par renforcement est que les décisions prises par l'algorithme influent sur l'environnement et les observations futures. L'exemple typique est celui d'un robot autonome qui évolue et effectue des actions dans un environnement totalement inconnu initialement. Il doit constamment apprendre de ses erreurs et succès passés, et décider de la meilleure politique à appliquer pour choisir sa prochaine action.

1.3.4 Inter-relations entre les techniques

Bien entendu, les frontières entre les tâches que nous venons de présenter sont souples. Ainsi on applique couramment, et avec succès, des algorithmes conçus pour faire de la régression à des problèmes de classification, ou bien on estime des densités dans le but de faire de la classification (voir la section 3.1).

Notez qu'une bonne estimation de densité permet en théorie de prendre la décision optimale concernant un problème de classification ou de régression. Mais

d'un autre côté l'estimation de densité est souvent un problème plus difficile, en pratique avec un nombre fini de données d'entraînement.

Précisons que, dans la suite de l'exposé, nous limiterons notre attention aux problèmes de classification, régression, et estimation de densité dans $\mathbb{R}^n$ (la représentation de donnée la plus souvent utilisée). Nous porterons un intérêt tout particulier aux cas où n est grand, de l'ordre de 100 à 1000 : la *haute dimension*.

Chapitre 2

La généralisation : le grand défi de l'apprentissage

2.1 Mémoriser n'est pas généraliser

Le terme *apprentissage* dans la langue courante est ambigu. Il désigne aussi bien l'apprentissage "par coeur" d'une poésie, que l'apprentissage d'une tâche complexe telle que la lecture.

Clarifions la distinction :

– Le premier type d'apprentissage correspond à une simple mémorisation. Or les ordinateurs contemporains, avec leurs mémoires de masse colossales, n'ont aucune difficulté à mémoriser une encyclopédie entière[1], sons et images inclus.

[1]bien que la façon dont ils accèdent à cette mémoire ne soit pas qualitativement très différente de le façon dont un être humain accède à l'encyclopédie de sa bibliothèque - sa mémoire étendue -, juste un peu plus rapide…

– Le second type d'apprentissage se distingue fondamentalement du premier en cela qu'il fait largement appel à notre faculté de généraliser. Ainsi pour apprendre à lire, on doit être capable d'identifier un mot écrit d'une manière que l'on n'a encore jamais vue auparavant.

Bien qu'il soit trivial de mémoriser une grande quantité d'exemples, les généraliser à la résolution de nouveaux cas, même s'ils ne diffèrent que légèrement, est loin d'être un problème évident. Ce que l'on entend par *apprentissage* dans les algorithmes d'apprentissage, et qui en fait tout l'intérêt et toute la difficulté est bel et bien la capacité de généraliser, et non pas simplement celle d'un apprentissage par coeur.

2.2 Notations et formalisation du problème

En général nous utilisons des P majuscules pour identifier des probabilités, et des p minuscules pour les densités de probabilité, f étant réservé pour les fonctions de décision. Mais dans ce qui suit, par un léger abus de notation, nous utiliserons un P majuscule indifféremment pour désigner une probabilité *ou* une densité, en fonction du contexte et de la nature, discrète, catégorique, ou continue des variables. Ainsi avec des variables aléatoires X continue et Y discrète, $P(X = x|Y = y)$ sera une densité conditionnelle, alors que $P(Y = y|X = x)$ sera une probabilité conditionnelle.

Formalisons à présent le problème de l'apprentissage supervisé :[2]

[2]nous rappelons que nous ne considérons pas ici le cas des données temporelles, dont la nature non stationnaire occasionne des complications supplémentaires

On dispose d'un ensemble $\mathcal{D}$ de l données d'apprentissage, sous forme de paires *(entrée, sortie)* : $\mathcal{D} = \{(x_1, y_1), (x_2, y_2), ..., (x_l, y_l)\}$ que l'on suppose tirées de manière $i.i.d.$ d'une distribution inconnue $P(X, Y)$.

Typiquement, entrée et sortie sont représentées sous forme d'un vecteur réel : $X \in \mathbb{R}^n$, $Y \in \mathbb{R}^{n'}$

Le problème de l'apprentissage supervisé est alors, à partir de l'ensemble d'apprentissage (et possiblement de connaissances à priori que l'on possède du domaine), de trouver, pour de nouveaux (x, y) tirés de la même distribution $P(X, Y)$, un moyen de calculer le y associé au x en commettant le moins d'erreurs possible.

L'approche la plus couramment utilisée consiste à tout d'abord utiliser $\mathcal{D}$ pour trouver la "meilleure" fonction $f : x \mapsto y$, $f \in \mathcal{F}$ (où $\mathcal{F}$ est choisi à l'avance). Puis à utiliser la fonction ainsi modélisée pour associer un y à tout nouveau x de test.

On dispose en général d'une fonction de coût $C(y_{\textit{calculé}}, y_{\textit{réel}})$. Par exemple, pour la classification, si y représente le numéro de la classe, on pourrait compter le nombre d'erreurs de classification à l'aide de la fonction de coût définie comme :

$C(y_1, y_2) = 0$ si $y_1 = y_2$

$C(y_1, y_2) = 1$ si $y_1 \neq y_2$

Ou bien pour un problème de régression, on pourra s'intéresser à l'erreur quadratique : $C(y_1, y_2) = \|y_1 - y_2\|^2 = \sum_{j=1}^{n'} (y_{1j} - y_{2j})^2$

Idéalement, on voudrait trouver $f \in \mathcal{F}$ qui minimise l'erreur de généralisation, ou risque espéré : $R(f) = E[C(f(X), Y)] = \int C(f(x), y) P(x, y) dx dy$

Mais comme on ne connaît pas la vraie distribution $P(X, Y)$, on doit se contenter de trouver le f qui minimise un *estimé* de cette erreur de généralisation. Typiquement cet estimé $\tilde{R}(f)$ est construit à partir de deux termes :

– le risque (ou erreur) empirique calculé sur les données d'apprentissage : $R(f, \mathcal{D}) = \frac{1}{l} \sum_{i=1}^{l} C(f(x_i), y_i)$

– et une pénalité $H(f)$ qui induit une préférence sur les solutions f

$\mathcal{F}$ est souvent un ensemble de fonctions paramétré par un vecteur de paramètres θ et rechercher $f_\theta \in \mathcal{F}$ revient dans ce cas à rechercher un $\theta \in \mathbb{R}^p$ qui minimise $\tilde{R}(f_\theta)$

Cette approche qui consiste d'abord à trouver une fonction à partir des données d'entraînement, pour ensuite appliquer cette fonction sur les nouvelles données de test est l'approche *inductive*. Une approche légèrement différente, qui consiste à trouver une fonction dépendant également du ou des *points de test* considérés est dite *transductive.*

La fonction ainsi obtenue constitue un *modèle,* dans le sens qu'elle permet de modéliser la relation entre entrée et sortie. L'optimisation des paramètres se nomme la phase *d'entraînement* ou *d'apprentissage* du modèle, et permet d'obtenir un *modèle entraîné.*

2.3 Mesure de la performance de généralisation

Lorsque l'on construit un modèle afin qu'il minimise le risque empirique calculé sur les données d'apprentissage, l'erreur d'apprentissage ainsi obtenue ne peut être considérée comme une bonne mesure de l'erreur de généralisation : elle est

évidemment biaisée. Pour obtenir un estimé non biaisé de l'erreur de généralisation, il est crucial de mesurer l'erreur sur des exemples qui n'ont pas servi à entraîner le modèle. Pour cela, on divise l'ensemble des données disponibles en deux parties :

– un sous ensemble d'entraînement, dont les données serviront à l'apprentissage (ou entraînement) du modèle ;
– un sous ensemble de test, dont les données seront utilisées uniquement pour évaluer la performance du modèle entraîné. Ce sont les données "hors-échantillon d'entraînement". On obtient ainsi l'*erreur de test* qui est un estimé bruité, mais non biaisé de l'erreur de généralisation.

Par ailleurs l'ensemble d'entraînement est souvent lui-même partagé entre un sous-ensemble qui sert à apprendre les paramètres d'un modèle à proprement dit, et un autre sous-ensemble dit "de validation", qui sert à la sélection de modèle. La *sélection de modèle* est ici comprise au sens large : il peut s'agir de choisir le meilleur entre plusieurs familles de modèles très différents, ou entre des variantes très semblables d'un même modèle, dues uniquement à des variations des valeurs d'un hyper-paramètre contrôlant la définition du modèle.

Dans les cas où l'on dispose de trop peu de données, on peut utiliser une technique de *validation croisée*[92], pour générer plusieurs paires de sous-ensembles entraînement/test. Cette technique est coûteuse en temps de calcul, mais permet d'obtenir un bon estimé de l'erreur de test, tout en conservant suffisamment d'exemples pour l'entraînement.

Lorsque, dans le présent document, nous parlons de *performance* d'un modèle, nous entendons la performance "en test", telle que mesurée sur un ensemble de test ou par validation croisée.

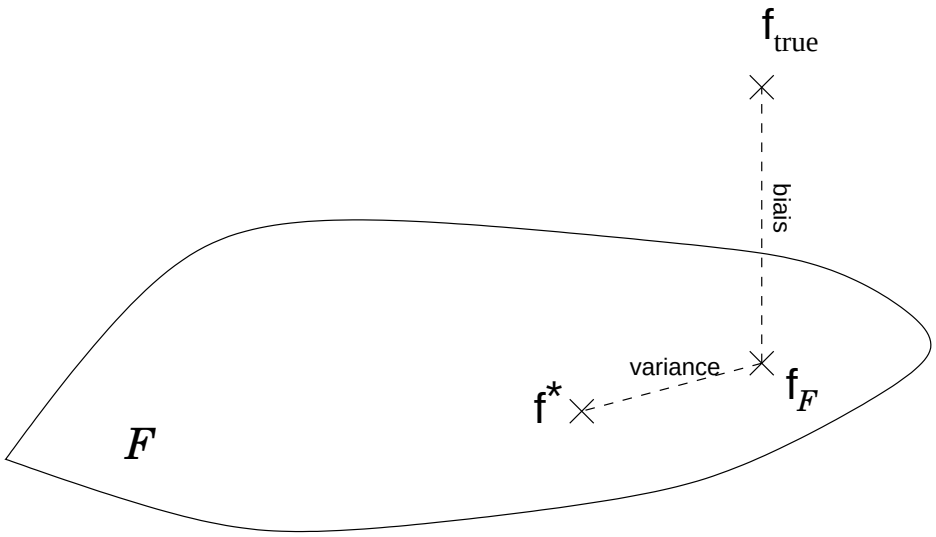

FIG. 2.1 – Biais et Variance

2.4 Quelques notions de théorie d'apprentissage

Le lecteur intéressé par un développement formel plus complet de la théorie de l'apprentissage statistique est invité à se référer à [102]. Nous nous contentons de présenter ici quelques notions.

D'une manière générale, l'approche inductive permet de combiner deux types d'informations pour résoudre le problème particulier (de classification ou de régression) qui nous occupe :

– Des connaissances ou intuitions à priori sur la forme que la solution devrait avoir. Elles se traduisent dans le choix de l'ensemble de fonctions $\mathcal{F}$ dans lequel on va chercher la solution, dans le choix de la fonction de coût C, et dans le choix de la fonction de pénalité $H(f)$ qui permet de spécifier une préférence parmi les fonctions de $\mathcal{F}$.

– Un nombre fini l d'exemples de paires (x, y), possiblement bruités, de valeurs de la "vraie" fonction en un certain nombre de points : notre ensemble d'apprentissage.

Un algorithme d'apprentissage nous permet en principe de trouver dans l'ensemble $\mathcal{F}$ la fonction f^* qui satisfait le mieux ces contraintes. Mais on se heurte typiquement à deux problèmes inconciliables :

– La "vraie fonction" idéale f_{true} ne se trouve peut-être pas dans l'ensemble $\mathcal{F}$ que nous avons choisi. En conséquence de quoi on aurait tendance à choisir un ensemble de fonction *plus vaste* pour limiter ce problème.
– Le nombre limité d'exemples d'entraînement dont nous disposons n'est pas suffisant pour localiser de façon précise la fonction $f^*_{\mathcal{F}}$ qui est la plus proche de f_{true} parmi notre ensemble de fonctions. Ce problème peut logiquement être réduit en choisissant un ensemble de fonctions $\mathcal{F}$ *plus petit*.

Le terme d'erreur dû au premier problème se nomme le *biais*, et celui dû au second se nomme la *variance* (car il est dû à la variabilité de l'échantillon fini qu'est l'ensemble d'apprentissage que l'on nous donne). Et le dilemme que cela occasionne est appelé *dilemme biais-variance*. Voir la figure 2.1.

On voit que la "taille" ou "complexité" de l'ensemble de fonctions $\mathcal{F}$ joue un rôle fondamental. Ce que l'on nomme la *capacité* $h(\mathcal{F})$ est une mesure de cette complexité.

2.5 Pratiques courantes de contrôle de capacité

Un nombre de pratiques couramment employées permet d'exercer un certain contrôle sur la *capacité*. Le principe fondamental est celui de la *régularisation*[97], qui permet d'introduire une préférence sur les fonctions de $\mathcal{F}$. La fonction de pénalité $H(f)$ que nous avons déjà mentionnée, est ajoutée au coût optimisé, et ce terme est appelée *terme de régularisation* : on favorise ainsi les fonctions plus simples (faible capacité) et pénalise davantage les plus complexes (capacité élevée).

Voici quelques méthodes couramment utilisées pour traduire ce principe en pratique. Le lecteur intéressé à la mise en pratique concrète de ces techniques particulièrement pour l'entraînement des réseaux de neurones est invité à se référer à [70].

– "*weight-decay*" : introduit une pénalité $H(f_\theta) = \sum \theta_i^2$ quadratique sur les valeurs des *paramètres* θ_i. On introduit ainsi une préférence pour des valeurs de θ proches de 0.
– design d'architecture : permet de choisir $\mathcal{F}$
– design de fonction de coût : permet de choisir $H(f)$ et ainsi d'établir une préférence sur les $f \in \mathcal{F}$
– arrêt prématuré : technique permettant de décider d'arrêter un algorithme itératif avant qu'il n'atteigne la fonction f optimale pour le coût optimisé, et qui limite ainsi la taille ou capacité "effective" de l'espace de fonctions exploré.

Pour résumer, un algorithme d'apprentissage performant pour une tâche donnée s'obtient avant tout en combinant un nombre *suffisamment élevé* de données d'en-

traînement et de bonnes connaissances à priori sur le problème, à condition qu'on puisse en disposer.

Chapitre 3

Une tentative de classifi cation des algorithmes d’apprentissage

La quantité et la diversité des algorithmes d’apprentissage rend toute entreprise de taxonomie hasardeuse. Celle, peut-être peu conventionnelle, que nous présentons ici, tente une classification des algorithmes d’après ce que nous considérons comme leur *philosophie*. Dans cette présentation nous nous limiterons la plupart du temps, par souci de simplicité, à une perspective de classification, mais certains des concepts présentés peuvent s’adapter aisément à des problèmes de régression ou d’estimation de densité.

3.1 Les modèles génératifs

Ces algorithmes partent de l'hypothèse que les données que l'on observe ont été générées par un processus aléatoire que l'on va modéliser. On part alors d'un modèle paramétrisé de ce que l'on pense pouvoir être ce processus, et on tente d'estimer les paramètres qui ont le plus vraisemblablement donné naissance aux données observées (principe du maximum de vraisemblance). Ceci correspond au cadre des statistiques paramétriques tel que développé par Fisher [30, 31, 32, 33, 2], même si en pratique on ne suppose pas forcément que le modèle envisagé est réellement le "vrai" modèle ayant généré les données, mais simplement qu'il permettra de bien généraliser une fois appliqué à de nouveaux points de test.

Un algorithme basé sur un modèle génératif aboutit généralement à un estimateur de densité $p(x)$. Mais on peut aussi utiliser ces techniques en appliquant la règle de Bayes pour construire un classifieur. Pour les problèmes de classification, on construit un modèle de densité p_c différent pour chaque classe c : $P(X = x|Y = c) = p_c(x)$, dont les paramètres sont estimés de manière à maximiser la vraisemblance des x_i qui correspondent à cette classe dans l'ensemble d'apprentissage. Puis, au moment de prendre une décision quant à la classe c d'un nouveau x qui nous est présenté, on utilise la règle de Bayes pour obtenir la probabilité à posteriori de la classe :

$$P(Y = c|X = x) = \frac{P(X=x|Y=c)P(Y=c)}{P(X=x)}$$

où $P(Y = c)$ est la probabilité à priori de la classe c (et $P(X = x)$ est un simple facteur de normalisation).

Une alternative à l'adoption de la solution de maximum de vraisemblance, est la pure approche Bayesienne qui considère l'intégrale sur toutes les valeurs possible des paramètres du modèle, en tenant compte d'une probabilité à priori sur ces valeurs (un à priori sur le modèle). Bien que très attrayante d'un point de vue théorique, l'approche Bayesienne présente souvent des difficultés contraignantes de mise en oeuvre en pratique, impliquant le recours à des approximations, et limitant leur applicabilité.

Parmi les approches inspirées de modèles génératifs qui ont emporté un grand succès, on peut citer les Chaînes de Markov Cachées (utilisées notamment dans les systèmes de reconnaissance de la parole), et les modèles de Mixtures de Gaussiennes.

3.2 Modélisation directe de la surface de décision

Le concept de surface de décision est commun à tous les algorithmes de classification dans $\mathbb{R}^n$. En effet, la capacité de décider d'une classe pour chaque point x de l'espace d'entrée induit automatiquement une partition de cet espace. Si on se limite, pour la simplicité de l'exposé, au cas de deux classes ($y \in \{-1, +1\}$), pour $x \in \mathbb{R}^n$, il en résulte une frontière (pas nécessairement continue) qui délimite les zones des deux classes, et que l'on nomme surface de décision. Il s'agit d'une "surface" de dimension $n - 1$ dans l'espace d'entrée de dimension n.

Tous les algorithmes de classification dans $\mathbb{R}^n$ induisent de telles surfaces de décision, mais seuls quelques uns en font leur point de départ : ils partent d'une

hypothèse quant à la forme de cette surface de décision[1] : linéaire, polynomiale, etc... puis cherchent, pour cette classe de fonctions, les paramètres qui vont minimiser le critère de coût désiré (idéalement l'espérance des erreurs de classification). Ainsi, l'ancêtre des réseaux de neurones, l'algorithme du Perceptron [77] recherche une surface de décision linéaire, représentée par une *fonction de décision* $f(x) = (< w, x > +b), w \in \mathbb{R}^n, b \in \mathbb{R}$.

La surface de décision correspond à $f(x) = 0$, et la classe d'un point x est donnée par le signe de $f(x)$.

Il en va de même pour le plus récent et très populaire algorithme des SVM [11, 102], sur lequel nous reviendrons en détails au chapitre 5. Bien que le critère optimisé, qui repose sur des fondations théoriques plus solides, soit quelque peu différent de celui du Perceptron, et qu'une "astuce"[2] permette d'étendre aisément l'algorithme à la modélisation de surfaces de décision non linéaires (Polynomiales...), le point de départ des SVM n'en reste pas moins une façon de trouver une simple surface de décision linéaire convenable.

3.3 Extraction progressive de caractéristiques

Les modèles génératifs, lorsqu'ils sont utilisés comme indiqué précédemment pour la classification, suggèrent un processus qui part des classes et génère les entrées observées (la variable aléatoire X). Mais on peut également imaginer un

[1] plus spécifi quement ils partent d'une hypothèse quant à la la forme de la *fonction de décision* $f(x)$ dont le signe indique la décision, et qui induit la surface $\{x \in \mathbb{R} | f(x) = 0\}$

[2] *l'astuce du noyau* (Anglais : *kernel trick*), qui peut d'ailleurs également s'appliquer au Perceptron [37].

processus inverse, qui part des entrées, et par transformations et calculs successifs, produit en sortie la décision quant à la classe correspondante, et ceci sans supposer que ces transformations aient quoi que ce soit à voir avec un processus qui aurait généré les données. Dans cette catégorie on trouve les réseaux de neurones multi-couches, qui se sont largement inspirés de nos connaissances sur le système nerveux (en particulier les architectures en couches des premiers étages du processus visuel). Dans cette optique, la couche d'entrée représente les données sensorielles brutes, et chaque couche subséquente calcule des caractéristiques de plus haut niveau, et ce progressivement jusqu'à la dernière couche, qui calcule un score pour chaque classe. Le plus bel exemple de succès de ce type d'approche est sans doute l'architecture LeNet pour la reconnaissance de caractères [57, 12, 58].

3.4 Modèles basés sur des distances à des prototypes

Un grand nombre d'algorithmes d'apprentissage se basent, pour prendre leur décision (concernant la classe d'un point de test par exemple), sur la distance calculée entre le point de test, et un certain nombre de *prototypes* (des points appartenant au même espace d'entrée que le point de test). Il faut comprendre ici la notion de *distance* au sens large, comme une mesure de similarité-dissemblance entre deux points. Les *noyaux*, sur lesquels nous reviendrons au chapitre 5, entrent généralement dans cette catégorie de "mesures de similarité".

La grande variété de ces algorithmes est due à

– La façon de choisir les prototypes : se limite-t-on à des points appartenant à l'ensemble d'apprentissage (tous ? un sous ensemble ? lesquels ?) ou bien

cherche-t-on à “inventer” des prototypes qui résumeraient l’ensemble d’apprentissage (comment ?).

– Le choix de la distance permettant de mesurer la similarité entre les points. Le choix de loin le plus courant pour les problèmes dans $\mathbb{R}^n$ est de se baser sur la distance Euclidienne.
– Comment l’information de distance aux prototypes est utilisée pour prendre la décision.

Parmi les algorithmes qui conservent comme prototypes la totalité des points de l’ensemble d’apprentissage, on peut inscrire les méthodes statistiques non paramétriques classiques : K plus proches voisins, régression à noyau, estimateur de densité à fenêtres de Parzen. Nous présenterons plus en détails tous ces algorithmes dans la section 5.2. On regroupe parfois ce type de méthodes sous les termes anglais *memory based*, *template matching*, ou encore *lazy learning*. On peut voir que l’optique est très différente de celle des modèles génératifs et de l’approche statistique paramétrique exposés à la section 3.1.

Parmi les algorithmes qui “construisent” leurs prototypes, on peut mentionner l’algorithme du centroïde, qui résume chaque classe par le centroïde des points d’entraînement appartenant à cette classe (voir la section 5.3.4). Les réseaux de neurones de type RBF [75] peuvent aussi être vus comme apprenant un petit nombre de prototypes, et produisant une fonction de décision qui est une combinaison linéaire de noyaux Gaussiens (fonction de la distance aux prototypes).

3.5 Valeur relative de cette taxonomie

Il nous faut préciser que les catégories présentées ci-dessus sont quelque peu artificielles. Il ne faut pas les considérer comme une classification rigide. Pour certains algorithmes, il est difficile de décider d'une unique catégorie (par exemple les mixtures de Gaussiennes : modèle génératif ou basé sur des distances à des prototypes ? Les arbres de décisions : extraction progressive de caractéristiques, ou modélisation d'une surface de décision linéaire par morceaux parallèle aux axes ? Dilemme également pour les RBF, et les SVMs à noyaux). La distinction n'est généralement pas aussi tranchée. Un algorithme peut être étudié sous de nombreux points de vue, autres que celui qui lui a donné naissance, et c'est souvent un exercice fort instructif.

Chapitre 4

Les défi s de la haute dimensionalité

Les données provenant de problèmes d'apprentissage concrets réels apparaissent souvent en haute ou très haute dimension : c.a.d. qu'un grand nombre de variables ont été mesurées pour chaque exemple d'apprentissage. Par exemple le profil d'un client d'une compagnie d'assurance ou d'une banque peut comporter les valeurs de plus d'une centaine de variables informatives.

Or l'apprentissage en haute dimension est un problème difficile, du fait de ce que l'on a nommé le *fléau de la dimensionalité*[1]. Par ailleurs les intuitions géométriques valables en faible dimension peuvent se révéler fausses ou inutiles en haute dimension et certains algorithmes qui fonctionnent très bien en faible dimension peuvent donner de très pauvres performances en haute dimension.

Élaborer des modèles capables de bien généraliser en haute dimension est l'un des plus grands défis de l'apprentissage statistique.

[1] Anglais : *curse of dimensionality*

4.1 Le fléau de la dimensionalité

Le fléau de la dimensionalité [8] fait référence à la croissance exponentielle de l'espace explorable avec le nombre de dimensions. Il suffit de penser par exemple au nombre de coins d'un hyper-cube en dimension n :

– En dimension 2 (un carré) : $2^2 = 4$

– En dimension 3 (un cube) : $2^3 = 8$

– En dimension 100 : $2^{100} = 1267650600228229401496703205376$.

On voit qu'en dimension 100 déjà, le nombre est très largement supérieur à la taille des bases de données d'apprentissage auxquelles on va typiquement avoir à faire. Ce qui n'empêche nullement ces bases de données d'être en dimension 100 ou plus ! Ainsi le nombre d'exemples dont on dispose est ridiculement faible par rapport à la taille de l'espace dans lequel ils sont disposés...

Notez que le nombre de coins d'un hyper-cube en dimensions n est le nombre de combinaisons possibles de valeurs que peuvent prendre n variables binaires, c.a.d. qui ne peuvent prendre *chacune* que deux valeurs possibles. Si les variables peuvent prendre davantage de valeurs, l'effet est encore plus dramatique, et pour des variables continues, cela devient difficile à concevoir !

4.2 Intuitions géométriques en haute dimension

En plus de la taille gigantesque des espaces en haute dimension, l'intuition géométrique qui nous guide en dimension 2 et 3 est souvent trompeuse en haute dimension.

Une particularité est que l'information de distance Euclidienne entre deux points en faible dimension est beaucoup plus informative qu'en haute dimension. On peut le comprendre dans le sens où la distance est finalement une statistique scalaire résumant n variables (les différences entre les n coordonnées des deux points). En haute dimension, seule une valeur de distance proche de 0 est réellement informative, car elle indique que *toutes* les n variables sont proches. Les valeurs de distance élevées indiquent simplement que certaines variables diffèrent, sans contenir aucune information quant à leur identité (lesquelles diffèrent beaucoup et lesquelles peu ou pas du tout, et il y a bien davantage de possibilités quant à leur identité en dimension élevée). Nous qualifions ce phénomène de *myopie* de la distance Euclidienne, car elle ne permet pas de clairement voir loin, n'étant informative que très localement. Cette *myopie* s'accroît avec la dimensionalité, car plus la dimension est élevée, plus la proportion de points éloignés par rapport aux points proches devient écrasante (si l'on suppose des données tirées d'une distribution uniforme dans un hypercube par exemple).

Un autre point qui vaut d'être mentionné est l'importance de l'extrapolation par rapport à l'interpolation. En faible dimension, on a tendance à penser en termes d'interpolation (entre un petit nombre de points d'une courbe par exemple). Mais en haute dimension, la probabilité qu'un point de test appartienne à la fermeture convexe des données d'apprentissage devient très faible. On se trouve donc la plupart du temps en situation d'extrapolation.

4.3 La notion de variété de plus faible dimension

En dépit du fléau de la dimensionalité, en pratique on parvient parfois à obtenir des résultats raisonnables même en très haute dimension (par exemple la base de données de chiffres MNIST qui est en dimension 784). La raison en est que les nombreuses variables observées ne sont généralement pas indépendantes, et que les données d'un problème sont naturellement très loin d'être distribuées uniformément dans un hypercube (sinon on n'arriverait effectivement à rien). Elles reflètent une certaine structure sous-jacente, qu'un algorithme d'apprentissage est capable de plus ou moins bien capturer.

Dans cette optique, une notion qui connaît un regain de popularité, surtout depuis la publication de [78, 95] est l'idée que des données en haute dimension pourraient typiquement être concentrées le long d'une *variété*[2] *non-linéaire de dimension inférieure*. Cette situation est illustrée dans la Figure 4.1. On peut par exemple imaginer qu'un processus sous-jacent effectivement modélisable, dépendrait d'un plus petit nombre m de facteurs que la dimension n de l'espace où les données sont observées. Si l'on suppose en outre que de petites variations continues de ces facteurs devrait se traduire par de petites variations des variables observées, alors ces m facteurs constituent effectivement une paramétrisation d'une variété de dimension m dans l'espace observé de dimension $n > m$.

Cette notion de variété est aussi ce qui sous-tend la technique des *distances tangentes* [87], où le sous-espace tangent en chaque point est *déduit de connaissances à priori* sur des transformations invariantes pour la tâche en question (petites ro-

[2] Anglais : *manifold*

tations ou translations de tous les pixels de l'image d'un chiffre manuscrit, qui ne changent pas sa classe).

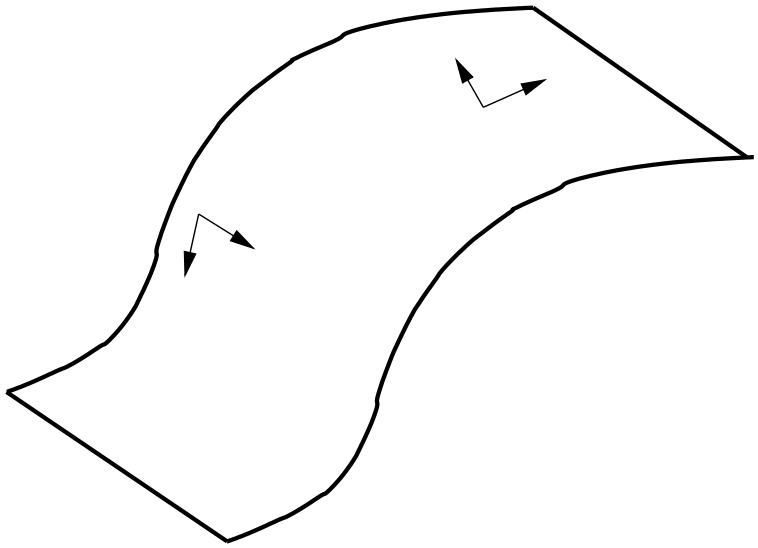

FIG. 4.1 – Illustration du concept de variété. Une variété non-linéaire de dimension 2 dans un espace de dimension 3. Deux points sont représentés, avec pour chacun deux vecteurs définissant le sous-espace tangent en ce point.

Deuxième partie

Modèles à noyaux

Chapitre 5

Méthodes à noyau classiques et modernes

Dans ce chapitre, nous passons brièvement en revue les méthodes à noyau les plus connues, en distinguant d'une part les méthodes classiques d'inspiration non paramétrique, et d'autre part les méthodes à noyau selon l'acception moderne, liée à *l'astuce du noyau*[1].

5.1 Noyaux et distances

La notion mathématique de distance entre deux instances traduit bien notre notion intuitive de "dissimilarité", indiquant à quel point les deux instances sont peu semblables. Les noyaux, qui sont souvent définis à partir d'une distance (par exemple

[1] Anglais : *kernel trick*

$e^{-distance}$) sont qualitativement similaires. C'est à cet aspect que nous faisons référence lorsque nous utilisons le terme *mesure de similarité/dissimilarité* plutôt qu'à une définition formelle de distance.

On considère habituellement un noyau comme une fonctionnelle $K(x; x_{centre})$, dont on peut fixer le centre et en faire ainsi une fonction à valeur réelle. On parlera alors d'un noyau centré en x_{centre}.

Le noyau le plus couramment utilisé est le noyau Gaussien, qui correspond à une densité de loi Normale multivariée. En haute dimension, pour $x \in \mathbb{R}^n$, des paramétrisations plus ou moins riches sont possibles, notamment :

– La Gaussienne sphérique ou isotrope de variance σ^2 :

$$K(x; \mu) = \frac{1}{\sqrt{(2\pi\sigma^2)^n}} e^{-\frac{\|x-\mu\|^2}{2\sigma^2}}$$

où $\|x - \mu\|$ est la distance Euclidienne entre x et μ.

– La Gaussienne pleine, de matrice de covariance C :

$$K(x; \mu) = \frac{1}{\sqrt{(2\pi)^n |C|}} e^{-\frac{1}{2}(x-\mu)' C^{-1} (x-\mu)}$$

où $|C|$ est le déterminant, et C^{-1} l'inverse de la matrice C.

Pour une Gaussienne sphérique, on parlera souvent de sa *largeur* σ.

Notez que les formulations utilisées en pratique pour le noyau Gaussien sphérique diffèrent souvent un peu de celle énoncée ci-dessus. En particulier beaucoup d'algorithmes ne nécessitent pas que le noyau soit normalisé (qu'il intègre à un), et omettent le facteur de normalisation, ou encore ils utilisent une expression un peu différente pour la "largeur". Ainsi ne soyez pas surpris si vous rencontrez une définition plus simple de noyau Gaussien telle que par exemple $K(x; \mu) = e^{-\frac{1}{\sigma}\|x-\mu\|^2}$

5.2 Méthodes à noyau classiques : non paramétriques

On présente ici trois algorithmes classiques importants, s'apparentant aux statistiques non paramétriques, et basés sur des mesures de distance ou des noyaux. Le premier est un algorithme de classification, le second de régression, et le troisième d'estimation de densité.

5.2.1 L'algorithme des k plus proches voisins (KNN)

Le principe de cet algorithme de classification est très simple : On lui fournit un ensemble de données d'apprentissage $\mathcal{D}$, une fonction de distance d, et un entier k. Pour tout nouveau point de test $x \in \mathbb{R}^n$ pour lequel il doit prendre une décision, l'algorithme recherche dans $\mathcal{D}$ les k points les plus proches de x au sens de la distance d, et attribue à x la classe qui est la plus fréquente parmi ces k voisins. Le fait de considérer, dans le cas général, k voisins, plutôt que l'unique voisin le plus proche permet une certaine robustesse aux erreurs d'étiquetage[2].

Une variante de cet algorithme peut être utilisée pour la régression : on attribue alors au point de test la moyenne des valeurs associées à ses K voisins.

5.2.2 La régression à noyau : l'estimateur de Nadaraya-Watson

L'algorithme connu sous le nom d'estimateur de *Nadaraya-Watson*[67], est parfois aussi appelé fenêtres de Parzen de régression.

[2] Anglais : *label noise*

Là encore, on dispose d'un ensemble de données d'apprentissage sous formes de paires *(entrée, sortie)* : $\mathcal{D} = \{(x_1, y_1), (x_2, y_2), ..., (x_l, y_l)\}$. On dispose également d'un noyau K (typiquement un noyau Gaussien sphérique de largeur fixe). La valeur qu'on associe à un point test $x \in \mathbb{R}^n$ est alors simplement une moyenne des valeurs y_i de l'ensemble d'apprentissage, pondérée par le noyau (c'est à dire par la "similarité" entre l'entrée x et les x_i) :

$$y = f(x) = \frac{\sum_{i=1}^{l} K(x, x_i) y_i}{\sum_{i=1}^{l} K(x, x_i)}$$

5.2.3 Les fenêtres de Parzen pour l'estimation de densité

L'estimation de densité étant une tâche non supervisée, on dispose d'un ensemble de données d'apprentissage $\mathcal{D} = \{x_1, x_2, \ldots, x_l\}$, ainsi que d'un noyau K qui doit correspondre à une densité et donc intégrer à 1 (typiquement un noyau Gaussien sphérique de largeur fixe). L'estimateur de densité de Parzen, qui peut être vu comme un "lissage" de la densité empirique provenant des données d'apprentissage est :

$$\hat{p}(x) = \frac{1}{l} \sum_{i=1}^{l} K(x; x_i)$$

5.3 Les méthodes à noyau “modernes”

Après une brève description des SVMs linéaires, nous expliquerons *l’astuce du noyau*[3], qui permet d’étendre les SVMs à des surfaces de décisions non-linéaires, et qui a donné naissance à l’acception moderne de “méthodes à noyau”.

5.3.1 Les machines à vecteurs de support (SVM) linéaires

L’algorithme des machines à vecteurs de support [11, 102] a reçu ces derniers temps beaucoup d’attention de la part de la communauté de recherche en algorithmes d’apprentissage, en partie à cause de ses fondements théoriques solides, et en partie par suite de ses succès pratiques sur des problèmes concrets.

Le principe d’origine est néanmoins fort simple : étant donné un ensemble d’entraînement $\mathcal{D}$ comportant des représentants de deux classes (la classe $+1$ et la classe -1), et que l’on suppose linéairement séparables, l’algorithme cherche la séparation linéaire qui maximise la *marge*. La *marge* peut être définie comme la distance euclidienne minimale entre la surface de séparation (un hyperplan) et le point le plus proche de l’ensemble d’apprentissage[4]. C’est là la version dite *à marge dure*[5] [11] (voir la figure 5.1). Une extension dite *à marge molle*[6] [22] permet de traiter les cas non séparables, où l’on accepte que certains points se situent

[3] Anglais : *kernel trick*

[4] Remarque : Il s’agit de la marge L_2 Euclidienne, aussi appelée *marge géométrique*. Il existe une autre acception de marge dite *marge fonctionnelle* sur laquelle nous nous attarderons dans la section sur les fonctions de coût de marge *(Margin loss functions)* de notre premier article.

[5] Anglais : *hard-margin*

[6] Anglais : *soft-margin*

en deçà de la marge, voire du mauvais côté de la surface de décision, au prix d'une pénalité linéaire, contrôlée par un paramètre de capacité C.

La solution *linéaire* trouvée par cet algorithme peut s'exprimer, tout comme celle du Perceptron, sous la forme suivante :

$f(x) =< w, x > +b$, $w \in \mathbb{R}^n$, $b \in \mathbb{R}$ où $< w, x >$ dénote le produit scalaire usuel entre le vecteur de paramètres w et x.

– l'hyperplan de séparation correspond alors à $f(x) = 0$,
– le signe de la fonction de décision f indique la classe d'un point x,
– $\frac{|f(x)|}{||w||}$ est la distance Euclidienne d'un point x à l'hyperplan.

Nous ne nous attardons pas ici sur les détails de l'algorithme qui permet de trouver cette solution de marge maximale. Signalons simplement que cela revient à résoudre un problème de programmation quadratique sous contrainte, lequel est précisé dans notre premier article au Chapitre 9.

5.3.2 Du linéaire au non-linéaire

Pour des problèmes complexes, une séparation linéaire est rarement suffisante pour obtenir une bonne performance de classification. Des fonctions de décisions plus riches, non linéaires, sont nécessaires.

Un moyen simple pour obtenir des surfaces de décisions non-linéaires avec un algorithme linéaire, consiste à étendre les vecteurs d'entrée. Le principe en est le suivant : plutôt que de chercher une surface de décision linéaire dans l'espace de départ des $x \in \mathbb{R}^n$, on transforme tout d'abord les entrées dans un autre espace (typiquement de dimensionalité beaucoup plus grande) à l'aide d'une fonction

Φ, et on cherche une surface de décision linéaire dans cet espace ($l'espace - \Phi$) plutôt que dans l'espace d'entrée. Cette fonction Φ peut par exemple calculer tous les produits d'ordre p des éléments du vecteur d'entrée. Dans ce cas une surface de décision linéaire dans cet espace transformé correspondra à une surface de décision polynomiale d'ordre p dans l'espace d'entrée de départ.

Illustration :

Soit $x \in \mathbb{R}^3$, c.a.d. $x = (x_{[1]}, x_{[2]}, x_{[3]})$

Note : nous utilisons ici la notation $x_{[i]}$ pour représenter la i^{eme} coordonnée du point x, afin de la distinguer de la notation x_i qui indique le i^{eme} point d'un ensemble de donnée.

Un classifieur linéaire produirait une fonction de décision, linéaire

$$f(x) = < w, x > + b = b + w_{[1]}x_{[1]} + w_{[2]}x_{[2]} + w_{[3]}x_{[3]}$$

Mais en revanche, si on construit un classifieur linéaire $\tilde{f}$, non pas sur l'espace des x, mais sur un espace *étendu* de dimension supérieure, par exemple

$$\tilde{x} = \Phi(x) = (x_{[1]}, x_{[2]}, x_{[3]}, x_{[1]}^2, x_{[2]}^2, x_{[3]}^2, x_{[1]}x_{[2]}, x_{[1]}x_{[3]}, x_{[2]}x_{[3]}) \tag{5.1}$$

Alors on obtient un classifieur, linéaire en $\tilde{x}$, mais non-linéaire en x :

$$\begin{aligned}
\tilde{f}(\tilde{x}) &= < \tilde{w}, \tilde{x} > + b \\
&= b + \tilde{w}_{[1]}x_{[1]} + \tilde{w}_{[2]}x_{[2]} + \tilde{w}_{[3]}x_{[3]} \\
&\quad + \tilde{w}_{[4]}x_{[1]}^2 + \tilde{w}_{[5]}x_{[2]}^2 + \tilde{w}_{[6]}x_{[3]}^2 \\
&\quad + \tilde{w}_{[7]}x_{[1]}x_{[2]} + \tilde{w}_{[8]}x_{[1]}x_{[3]} + \tilde{w}_{[9]}x_{[2]}x_{[3]}
\end{aligned}$$

ce qui, on le voit, correspond à une surface de décision polynomiale d'ordre 2.

Cette façon de procéder n'est pas nouvelle, mais elle posait souvent de nombreux problèmes pratiques. Ainsi le nombre de dimensions de ces $espaces - \Phi$ croît en général exponentiellement avec le nombre de dimensions de l'espace d'entrée, et il devient rapidement impossible en pratique de calculer et stocker les vecteurs étendus.

5.3.3 L'astuce du noyau

Pour de nombreux algorithmes d'apprentissage linéaires, le vecteur de poids w produit par l'algorithme n'est finalement qu'une somme pondérée d'exemples d'apprentissage : $w = \sum_{i=1}^{l} \alpha_i x_i$, et l'algorithme peut être adapté pour trouver α et b en utilisant exclusivement les résultats de produits scalaires entre les x_i. C'est le cas en particulier des SVMs ainsi que du Perceptron.

L'application du modèle à un point test x peut également se faire en n'utilisant que des résultats de produits scalaires avec les x_i :

$$f(x) =< w, x > +b = b + \sum_{i=1}^{l} \alpha_i < x_i, x > \tag{5.2}$$

L'astuce du noyau (introduite pour la première fois par [1]) consiste alors à utiliser un noyau $K(x; x_i)$ pour calculer tous ces produit scalaire dans un espace Φ étendu : $< \Phi(x), \Phi(x_i) >= K(x; x_i)$, mais de telle sorte qu'on n'a jamais besoin de produire explicitement les $\Phi(x)$.

Illustration :

Plaçons nous dans le même cadre que précédemment, $x \in \mathbb{R}^3$, mais en utilisant une expansion similaire mais légèrement différente de celle de l'équation 5.1 :

$$\Phi(x) = (1, \frac{1}{\sqrt{2}}x_{[1]}, \frac{1}{\sqrt{2}}x_{[2]}, \frac{1}{\sqrt{2}}x_{[3]}, \frac{1}{\sqrt{2}}x_{[1]}x_{[2]}, \frac{1}{\sqrt{2}}x_{[1]}x_{[3]}, \frac{1}{\sqrt{2}}x_{[2]}x_{[3]}, x^2_{[1]}, x^2_{[2]}, x^2_{[3]})$$

On peut alors voir que, pour deux points $x \in \mathbb{R}^n, z \in \mathbb{R}^n$:

$$\begin{aligned} < \Phi(x), \Phi(z) > &= 1 + 2x_{[1]}z_{[1]} + 2x_{[2]}z_{[2]} + 2x_{[3]}z_{[3]} \\ &\quad +2x_{[1]}x_{[2]}z_{[1]}z_{[2]} + 2x_{[1]}x_{[3]}z_{[1]}z_{[3]} + 2x_{[2]}x_{[3]}z_{[2]}z_{[3]} \\ &\quad +x^2_{[1]}z^2_{[1]} + x^2_{[2]}z^2_{[2]} + x^2_{[3]}z^2_{[3]} \\ &= (1 + x_{[1]}z_{[1]} + x_{[2]}z_{[2]} + x_{[3]}z_{[3]})^2 \\ &= (1+ < x, z >)^2 \end{aligned}$$

On peut donc calculer $< \Phi(x), \Phi(z) >$ efficacement sans jamais avoir à calculer l'expansion explicitement, en utilisant un noyau "polynomial" d'ordre p : $K(x; z) = (1+ < x, z >)^p$. Dans notre exemple, on avait $p = 2$, mais le principe se généralise à des polynômes d'ordre plus élevé.

D'une manière générale, il correspond un tel $espace - \Phi$ à tout noyau satisfaisant les conditions de Mercer[7] [23]. Par exemple le noyau Gaussien précédemment évoqué correspond à un produit scalaire dans un espace transformé de dimension infinie [74], et est couramment utilisé dans les SVM avec succès.

[7] qui se résument essentiellement par être positif défi ni

5.3.4 Utilisation de l'astuce du noyau dans les algorithmes

Centroïde dans l'espace Φ

Pour illustrer comment on peut concrètement appliquer l'astuce du noyau à un algorithme existant, prenons l'exemple de l'algorithme du centroïde pour la classification.

Le centroïde d'un ensemble de points $T = \{x_1, \ldots, x_l\}$ est

$$c(T) = \frac{1}{l} \sum_{i=1}^{l} x_i.$$

Étant donné un ensemble d'apprentissage fini, on peut facilement calculer le centroïde des points de chaque classe. L'algorithme du centroïde pour la classification attribue alors un point de test x à la classe dont le centroïde est le plus proche (en distance Euclidienne). Lorsqu'on n'a que deux classes, donc deux centroïdes, cela engendre une surface de décision linéaire : l'hyper-plan bissecteur du segment reliant les deux centroïdes.

Le carré de la distance entre un point de test x et un centroïde est :

$$\begin{aligned}
d(x, c(T))^2 &= \|x - c(T)\|^2 \\
&= < x - c(T), x - c(T) > \\
&= < x, x > -2 < x, c(T) > + < c(T), c(T) > \\
&= < x, x > -2 < x, \frac{1}{l} \sum_{i=1}^{l} x_i > + < \frac{1}{l} \sum_{i=1}^{l} x_i, \frac{1}{l} \sum_{i=1}^{l} x_i > \\
&= < x, x > -\frac{2}{l} \sum_{i=1}^{l} < x, x_i > + \frac{1}{l^2} \sum_{i \in 1..l, j \in 1..l} < x_i, x_j >
\end{aligned}$$

On voit dès lors qu'on peut appliquer l'astuce du noyau, pour calculer la distance dans l'espace Φ induit par un noyau K :

$$d_\Phi(x,T) = d(\Phi(x), c(\Phi(T))) = K(x;x) - \frac{2}{l}\sum_{i=1}^{l} K(x;x_i) + \frac{1}{l^2}\sum_{i\in 1..l, j\in 1..l} K(x_i;x_j)$$

Si on utilise cette distance-Φ, avec un noyau Gaussien par exemple, on obtiendra dans le cas de deux classes une surface de décision non-linéaire dans l'espace d'entrée des x (bien qu'elle soit linéaire dans l'espace Φ).

Remarquez que pour un ensemble T donné et K Gaussien, $d_\Phi(x,T) = \text{constante} - 2\hat{p}(x)$ où $\hat{p}(x)$ est l'estimateur de densité de Parzen (pour les données de la classe) décrit à la section 5.2.3 !

Les SVMs à noyau

L'algorithme des SVMs à noyau est construit à partir des SVMs linéaire de la même manière que nous avons appliqué *l'astuce du noyau* au cas du centroïde.

La formulation usuelle de la fonction de décision des SVMs à noyau est :

$$f(x) = b + \sum_{i=1}^{l} \alpha_i y_i K(x;x_i)$$

avec $\alpha_i \in \mathbb{R}^+$ et les $y_i \in \{+1, -1\}$ indiquant la classe du point x_i.

Qui plus est, seuls les α_i qui correspondent à des points qui sont "sur la marge" sont non nuls : ces points sont appelés *vecteurs de support*.

Autres algorithmes

De nombreux algorithmes classiques, essentiellement linéaires, peuvent semblablement être étendus grâce à l'astuce du noyau. Mentionnons notamment les extensions à noyau du Perceptron [37, 45], et de l'analyse en composantes principales [80, 81].

Pour finir notre rapide tour d'horizon des algorithmes à noyau, nous nous devons au moins de mentionner les *Processus Gaussiens*[8] [107], très élégants d'un point de vue théorique, mais dont la complexité calculatoire a beaucoup limité l'applicabilité en pratique. Mais des développements récents [56, 83] pourraient changer la situation.

[8] Anglais : *Gaussian Processes*

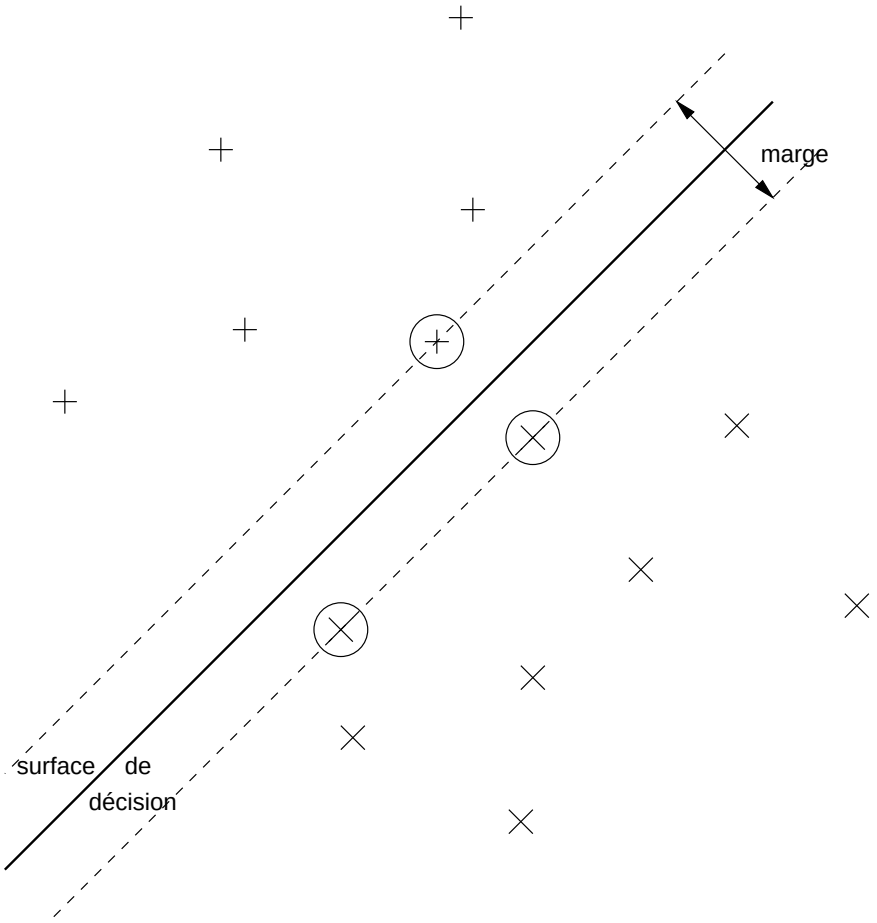

FIG. 5.1 – Surface de décision à marge maximale des SVMs. Les vecteurs de support sont repérés par un cercle.

Chapitre 6

La forme du noyau

6.1 Inteprétation classique ou moderne ?

L'utilisation des noyaux que nous avons qualifiée de "moderne" ne l'est pas vraiment, puisqu'on peut faire remonter l'*astuce* du noyau à [1]. Ce n'est pourtant que depuis son application relativement récente aux SVMs à noyau [11] que cette technique est devenue très populaire, et a connu d'importants développements.

On voit que le point de vue classique non paramétrique (Parzen) et celui de l'astuce du noyau (SVMs) sont au départ très différents. Dans le premier cas, il s'agit de méthodes basées explicitement sur des prototypes, alors qu'un SVM est initialement un modèle discriminant linéaire dont on cherche les paramètres. De ce point de vue les SVMs à noyau sont intrigants, car il n'est pas évident de savoir où les situer sur l'axe méthodes paramétriques $\rightarrow$ méthodes non-paramétriques... Par ailleurs, l'exemple du centroïde à noyau, montre bien qu'on peut parfois mettre

en évidence des liens entre les deux approches (dans ce cas ci, avec l'estimateur de densité de Parzen).

Une des différences les plus fondamentales est peut-être que l'utilisation de l'astuce du noyau impose un certain nombre de contraintes sur la forme du noyau (qui doit absolument être positif défini) et sur la forme de la solution (quel espace-Φ correspondrait à un noyau Gaussien de largeur différente sur chaque point ?) pour pouvoir justifier l'applicabilité de l'astuce.

Aucun des algorithmes que nous proposons et étudions dans notre recherche n'est fondé sur l'astuce du noyau, et il en résulte à notre avis, une flexibilité accrue.

6.2 Importance de la forme du noyau ou de la métrique

L'expérience montre que le choix de la mesure de similarité utilisée (distance[1] pour KNN, noyau pour les fenêtres de Parzen et les SVM) pour résoudre un problème particulier a parfois une influence considérable sur le succès de la méthode. Ce choix est souvent dicté par une série d'essais-erreurs guidés par l'intuition et des connaissances à priori sur le domaine. Cette façon de procéder, qui ternit quelque peu l'aspect "boîte noire clé en main" tant vanté des SVM, est parfois désignée par le qualificatif "*Kernel-engineering*".

[1] Dans notre discussion, nous utilisons indifféremment les termes distance ou métrique, au sens large de "mesure de similarité".

Ainsi, [18] ont étudié l'influence de différents types de noyaux de SVM sur un problème de classification d'images (base de donnée Corel). Leur recherche a débuté après qu'ils aient constaté qu'un simple KNN avec une distance L_1 donnait des résultats significativement meilleurs qu'une SVM avec un noyau Gaussien.

Un autre très bel exemple de développement d'une mesure de similarité qui incorpore des connaissances à priori spécifiques au problème en question (dans ce cas la reconnaissance de caractères) est la "distance tangente" [86, 87]. Cette métrique est invariante par rapport à de petites translations, rotations, agrandissements et réductions des caractères et a été employée avec beaucoup de succès dans un KNN sur la base de donnée NIST de chiffres manuscrits.

Étant donné l'importance qu'a la forme de la métrique ou du noyau sur la performance de ces algorithmes, plusieurs travaux ont étudié la possibilité de les *apprendre* automatiquement. Il s'agit alors d'apprendre les paramètres d'un noyau paramétré ou d'une métrique paramétrée (pouvant servir de base à la définition d'un noyau Gaussien). On peut distinguer essentiellement deux approches, selon que l'on essaye d'apprendre une métrique globale, valable sur tout l'espace, ou bien d'adapter localement les paramètres d'une distance ou d'un noyau dans le voisinage d'un point.

Un exemple classique de métrique *globale* que l'on peut apprendre est la *distance de Mahalanobis* : $d_{Mahalanobis}(x_1, x_2) = \sqrt{(x_1 - x_2)' C^{-1} (x_1 - x_2)}$ où C est la matrice de covariance des données, et constitue les paramètres "appris". Remarquez que cela revient à une distance Euclidienne sur les données pré-traitées : $d_{Mahalanobis}(x_1, x_2) = d_{Euclidienne}(C^{-\frac{1}{2}} x_1, C^{-\frac{1}{2}} x_2)$. On retrouve, sous une forme encore plus simple, ce même principe dans un autre pré-traitement courant que

l'on nomme généralement "normalisation" des données, et qui consiste à diviser chaque coordonnée par son écart type[2]. L'apprentissage de ce genre de métrique globale a notamment fait l'objet d'études dans le but d'améliorer les performances de classification d'algorithmes de type KNN [6, 7, 63], de même que l'apprentissage des paramètres d'un noyau a été exploré dans le but d'améliorer la performance d'algorithmes de type SVM [101, 3] ou d'autres algorithmes à noyau [69]. Des formes paramétriques plus complexes que Mahalanobis ont également été étudiées. Mentionnons notamment que des métriques et noyaux peuvent être définis à partir de modèles génératifs [109, 25, 51], ou incorporés à des architectures de réseaux de neurones afin d'être entraînées de manière discriminante [15, 5, 96]

Dans [104], nous avions nous-même commencé à explorer la voie de l'apprentissage des paramètres d'un *noyau global* en proposant l'architecture de réseau de neurones particulière illustrée à la Figure 6.1. Il s'agit d'une architecture entraînable globalement par descente de gradient permettant d'optimiser simultanément les paramètres θ du noyau K_θ et les poids α qui sont associés à chaque vecteur de support. Mais nous n'avons pas persévéré sur cette voie de recherche, en partie à cause de la complexité du modèle le rendant difficile à entraîner dans des temps raisonnables. Aussi nous nous contentons ici que mentionner ces travaux, en invitant le lecteur intéressé à se référer à l'article en question, mais surtout aux travaux de [55] pour les développements récents de cette autre voie de recherche.

Nos recherches se sont par la suite davantage concentrées sur l'utilisation, dans des algorithmes classiques, de *métriques locales* ou de noyaux à paramètres *appris localement* dans le voisinage d'un point d'intérêt. Ce sont ces recherches,

[2]Ce qui revient à ne conserver que les termes diagonaux de C dans les expressions précédentes

davantage proches des travaux précurseurs de [38], que nous présentons dans les deuxième et troisième articles de cette thèse. Précisons que cette approche de *métrique locale* ou *noyaux à structure locale* cadre bien avec le point de vue des méthodes à noyaux non paramétriques classiques mais qu'il est difficile de concevoir de ce point de vue ce que signifierait *l'astuce du noyau* (quel est l'espace-Φ correspondant quand on n'a pas un noyau unique ?). C'est pourquoi nous considérons *l'astuce du noyau* avant tout comme une astuce élégante, mais qui constitue une limitation intrinsèque d'algorithmes du genre SVM. Les algorithmes que nous allons présenter ici n'y font aucunement appel.

Par ailleurs, nous tenons à attirer l'attention du lecteur sur le résultat théorique important de [110]. Il y est démontré que *n'importe quelle* courbe lisse séparant deux ensembles de points peut correspondre à un hyper-plan séparateur à marge maximale dans un certain espace-Φ, induit par un noyau satisfiant les conditions de Mercer. Autrement dit *n'importe quelle* surface de décision séparatrice lisse, aussi absurde soit-elle, peut être déclarée à marge maximale : il suffit pour cela de choisir le bon noyau ! C'est un résultat marquant, car il relativise grandement l'importance de la maximisation de marge quand on utilise l'astuce du noyau, et montre en revanche le rôle crucial du choix du noyau.

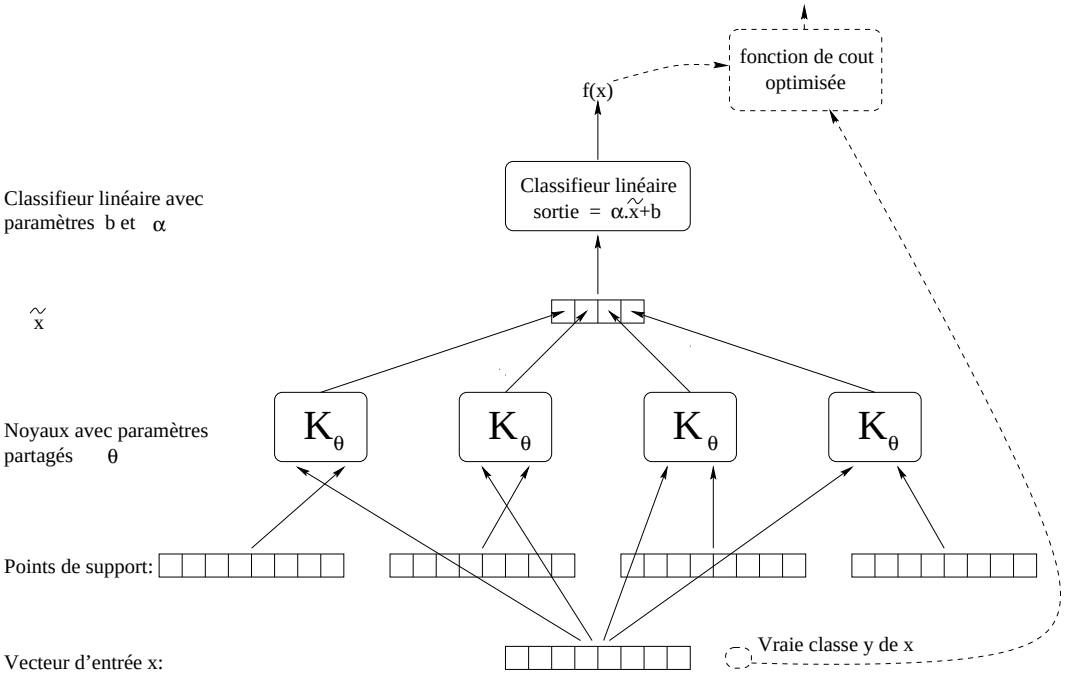

FIG. 6.1 – Architecture de réseaux de neurones pour l'apprentissage des paramètres θ d'un noyau global ainsi que des poids α de la combinaison linéaire.

Troisième partie

Les articles

Chapitre 7

Présentation générale de la recherche et des articles

7.1 Objectifs de la recherche

Lorsque nous avons débuté sur la voie des recherches que nous présentons dans les articles ci-dessous, les Machines à Vecteurs de Support commençaient à jouir d'une popularité grandissante. Cet intérêt qui n'a cessé de croître depuis est sans doute dû à deux facteurs principaux :

(i) L'attrait théorique. L'algorithme est mathématiquement simple, correspondant à un problème d'optimisation convexe admettant une solution unique[1]. En outre il est justifié par des arguments théoriques de théorie de l'apprentissage [102].

[1]Contrairement aux réseaux de neurones, dont l'analyse théorique se heurte à leur complexité, et à la question des minima locaux.

(ii) L'attrait pratique. Les SVMs ont acquis une réputation de solution "clé en main", étant beaucoup moins délicats à entraîner que des réseaux de neurones par exemple, et donnant souvent de très bonnes performances, notamment sur des problèmes en haute dimension.

Cette popularité a donné lieu à quantité de recherches au cours des dernières années : justifications théoriques, extensions, améliorations, applications pratiques des SVMs, sans oublier la réhabilitation de nombreux anciens algorithmes par la simple application de *l'astuce du noyau*.

Notre propre recherche a été motivée par l'objectif premier de développer de nouveaux algorithmes d'apprentissage génériques (c.a.d. n'utilisant pas de connaissances à priori du problème), capables de surpasser les performances des SVMs en pratique. Ce faisant, nous voulions aussi tenter de mieux cerner les caractéristiques qui font qu'un algorithme générique affiche de bonnes performances en haute dimension.

Notre approche dans l'élaboration de ces algorithmes a été pour l'essentiel empirique, guidée par des intuitions plus ou moins inspirées de certaines caractéristiques des SVMs, mais sans directement prendre les SVMs comme point de départ, et en évitant de recourir à l'astuce du noyau. Plutôt que de mener à la découverte d'algorithmes radicalement nouveaux, cette voie nous a conduit à redécouvrir d'anciens algorithmes sous un jour nouveau. Cela nous a permis notamment de proposer des extensions les rendant compétitifs avec les SVMs, et d'accroître notre compréhension du problème de la haute dimensionalité.

7.2 Présentation des articles

Nous présentons trois articles dans l'ordre chronologique des recherches. Tous trois ont été publiés ; il s'agit de :

(i) P. Vincent et Y. Bengio. **Kernel Matching Pursuit.** Publié en 2002 dans le journal *Machine Learning*, volume 48, pp 165-187. Chez Kluwer Academic Publishers.

(ii) P. Vincent et Y. Bengio. **K-Local Hyperplane and Convex Distance Nearest Neighbor Algorithms.** Publié en 2002 dans *Advances in Neural Information Processing Systems 14*, aux éditions MIT Press.

(iii) P. Vincent et Y. Bengio. **Manifold Parzen Windows.** Publié en 2003 dans *Advances in Neural Information Processing Systems 15*, aux éditions MIT Press.

Les trois articles ont pour auteur principal Pascal Vincent, et pour coauteur son directeur de recherche Yoshua Bengio. Dans les trois cas, Pascal Vincent a été à l'origine de l'idée de départ et a mené à terme le travail de recherche : il a conçu, réalisé, implémenté et testé expérimentalement les algorithmes proposés, puis analysé et tiré les conclusions des résultats obtenus, et finalement rédigé l'essentiel de l'article. Yoshua Bengio a contribué au travers de nombreux échanges à raffiner l'idée de départ, et a aidé à la rédaction, relecture et correction des articles.

Nous croyons utile de mentionner que Pascal Vincent, au cours de son travail de doctorat, a également contribué de façon significative à plusieurs autres travaux de recherche dans le domaine de l'apprentissage, ayant donné lieu à des publications, notamment [104, 17, 9, 10, 29]. Mais, étant moins pertinentes par rapport au sujet

des algorithmes à noyaux à structure locale, elles n'ont pas été incluses dans la présente thèse.

Les articles inclus reprennent quasiment à l'identique le texte original des publications. Nous y avons parfois ajouté quelques précisions sous forme de notes de bas de page, ou corrigé une erreur de frappe. Le formatage a par ailleurs été modifié pour être conforme aux standards de présentation de thèse de l'Université de Montréal. Les accords des coauteurs et éditeurs d'inclure ces textes dans cette thèse sont joints en annexe.

Dans les pages qui précèdent chaque article, nous tentons de mettre en évidence le contexte, les questions, les objectifs, et le cheminement intellectuel qui a motivé cette recherche. Nous y résumons aussi les contributions de l'article au domaine.

7.3 Remarque sur le choix des bases de données

Parmi les résultats expérimentaux que nous reportons sur des problèmes réels, le choix des bases de données de chiffres USPS et MNIST a été motivé par les points suivants :

– Nous nous intéressions à la haute dimension, et il s'agit de données en très haute dimension : USPS est en dimension 256 (16×16 pixels) et MNIST en dimension 784 (28×28 pixels).
– La popularité des SVMs en pratique est historiquement due en partie à leur bonne performance sur USPS et MNIST, telle que rapportée dans [82, 58]. Parmi les algorithmes génériques (c.a.d. n'incluant pas de connaissance à priori

du problème) les SVMs à noyau ont longtemps été celui donnant les meilleurs performances sur MNIST.

– Pour autant, on sait qu'il est *possible* de faire mieux que les SVMs sur MNIST (LeNet [57, 12, 58] bat largement les SVMs, mais ce n'est pas un algorithme générique comme les SVMs). Notre ambition était dès lors de faire mieux que les SVMs, ou du moins aussi bien, avec de nouveaux algorithmes *génériques.*
– Autre point important : ces bases sont suffisamment grandes pour que les différences mesurées entre les algorithmes puissent être statistiquement significatives.

Chapitre 8

Présentation du premier article

P. Vincent et Y. Bengio. **Kernel Matching Pursuit.** Publié en 2002 dans le journal *Machine Learning*, volume 48, pp 165-187, chez Kluwer Academic Publishers.

Notez qu'une version préliminaire de cet article a été rendue publique en 2000 sous forme de rapport technique (Rapport technique No. 119, Département d'Informatique et Recherche Opérationnelle, Université de Montréal, 2000).

8.1 Contexte et objectifs de cette recherche

Comme nous l'avons précisé précédemment, nous voulions proposer des alternatives aux SVMs à noyaux, affichant des performances égales ou supérieures en pratique, et par la même occasion essayer de mieux comprendre quelles caractéristiques sont responsables de la bonne performance des SVMs. Plusieurs aspects

des SVMs faisaient l'objet d'études ou étaient supportés par des considérations théoriques, susceptibles de justifier leur bonne performance :

- **La notion de maximisation de la *marge*** (telle que définie à la section 5.3.1) : [102] donne des bornes sur l'erreur généralisation liées à la largeur de la *marge*.
- **Le caractère clairsemé**[1] de la solution, c.a.d. le fait qu'il y ait un petit nombre de points de support. En effet, des rédultats reliant l'erreur de généralisation espérée au caractère clairsemé de la solution existent pour les SVMs [102, 103] ainsi que pour d'autres modèles similaires [61, 34, 45].
- **La forme de la solution** des SVMs, à savoir une combinaison linéaire de noyaux centrés sur un sous-ensemble des points d'apprentissage. En effet un théorème (*representer theorem*[54]) montre que c'est là la forme qu'a la fonction optimale qui minimise un coût régularisé particulièrement intéressant[2]. Dans les SVMs, cette forme résulte de l'utilisation de l'astuce du noyau.

8.2 Motivations d'un contrôle plus précis du nombre de vecteurs de support

Dans les SVM, les seuls centres qui demeurent dans la solution finale sont les *vecteurs de support* trouvés par l'algorithme (ceux pour lesquels $\alpha_i \neq 0$). Le nombre de vecteurs de support trouvé dépend du problème, de la forme et largeur du noyau, ainsi que du paramètre C qui contrôle en partie la capacité des SVM à marge molle, mais est en pratique difficile à contrôler. Des techniques vi-

[1] Anglais : *sparsity*

[2] le terme de régularisation correspond à la norme d'un RKHS (*reproducing Kernel Hilbert Space*). Voir les travaux de [54]

sant à réduire à posteriori le nombre de vecteurs de support ont été développées pour améliorer les temps de réponse lorsque leur nombre est initialement élevé [16]. Ces techniques ont été développées avant tout dans le but d'accélérer les algorithmes dans leur phase d'utilisation sur des données de test. Mais il y a également un lien direct évident entre le nombre de centres utilisés, et la "taille" de l'ensemble de fonctions $\mathcal{F}$ considéré, et donc la *capacité* du modèle. C'est dans cette optique que nous avons commencé à explorer des algorithmes permettant un contrôle stricte du nombre de centres, en vue de contrôler la capacité et ainsi espérer améliorer l'erreur de généralisation.

Précisons que le choix du nombre de centres est couramment utilisé pour contrôler la capacité d'algorithmes tels que les réseaux RBF. Mais contrairement aux RBF, nos études se concentrent sur des solutions où les centres sont, tout comme pour les SVM, un sous-ensemble des points d'apprentissage. Cela afin d'éviter que le nombre de paramètre libres du modèle, et donc sa capacité, ne croisse avec la dimensionalité de l'espace d'entrée.

Ignorant dans un premier temps les considérations de marge, nous avons donc cherché à développer un algorithme qui conserverait la forme de la solution des SVMs, tout en permettant un contrôle stricte du nombre de vecteurs de support. Nous voulions aussi éviter de recourir explicitement à l'astuce du noyau. C'est ainsi que nous avons conçu un simple algorithme glouton[3] constructif, qui ajoute les points de support un par un, utilisé avec une technique d'arrêt prématuré.

[3] Anglais : *greedy*

8.3 Découverte d'algorithmes semblables

Peu après nous nous sommes rendu compte que cette famille d'algorithme était utilisée depuis longtemps dans la communauté du traitement du signal sous le nom de *matching pursuit*[64][4], notamment pour les décompositions de signaux dans certaines bases d'ondelettes. Mais à notre connaissance, ils n'avaient encore jamais été utilisés avec, pour base de fonctions, des noyaux Gaussiens en haute dimension, centrés sur les exemples d'apprentissage. La nouveauté venait d'une utilisation dans un contexte d'apprentissage automatique, avec une technique d'arrêt prématuré basée sur un ensemble de validation pour contrôler la capacité effective du modèle. Nous avons donc logiquement appelé cet algorithme *Kernel Matching Pursuit*.

Mais nos surprises ne s'arrêtèrent pas là : *AdaBoost* [35, 36, 40], se disputait alors la popularité avec les SVMs, et une analyse élégante de cette famille d'algorithme, due à [65, 39], permet de les interpréter comme une forme de descente de gradient dans l'espace des fonctions. Or vu sous cet angle, ils sont très semblables à *matching pursuit*. Nous aurions ainsi tout aussi bien pu nommer notre algorithme *kernel boosting*. Ceci nous a amené à étudier de plus près la notion de *fonctions de coûts de marge*, puisque ces recherches sur *AdaBoost* tentaient alors d'expliquer la bonne performance de ce type d'algorithme par des notions de maximisation de marge, tout comme pour les SVMs. Nous avons ainsi examiné sous cet angle les fonctions de coût quadratique, et quadratique après activation sigmoïde qui sont couramment utilisées dans les réseaux de neurones, révélant ce faisant la similitude avec une autre fonction de coût de marge proposées dans [65].

[4] très similaires à l'algorithme de Projection Pursuit Regression [41]

Finalement il est apparu qu'une des variantes de notre algorithme était quasiment identique à l'algorithme *Orthogonal Least Squares RBF* [21].

8.4 Contributions au domaine

On peut résumer les contributions de cet article par les points suivants :

– La mise en évidence de liens entre les SVMs à noyau, les algorithmes de type *matching pursuit*, les algorithmes de type *boosting*, et l'algorithme *Orthogonal Least Squares RBF*.
– L'interprétation, sous forme de fonction de coût de marge (voir section 9.3.2), de l'erreur quadratique, et de l'erreur quadratique après activation sigmoïde, montrant une similitude avec une fonction de coût de marge suggérée dans [65].
– La révélation qu'un vieil algorithme de construction de RBF [21] permet d'atteindre une performance aussi bonne que les SVMs sur la base USPS, amenant à grandement relativiser les résultats extrêmement positifs (pour les SVMs) de la comparaison effectuée par [82].
– La mise en évidence de l'intérêt des techniques de *pré-fitting* et *back-fitting* (sections 9.2.2 et 9.2.3).
– Avoir proposé une alternative possible aux SVMs, non fondée sur l'astuce du noyau, menant à des solutions bien plus clairsemées, et permettant une flexibilité accrue grâce à l'utilisation d'un dictionnaire de fonctions.

Précisons également que l'algorithme KMP a des exigences en temps de calcul (voir section 9.2.4) et en espace mémoire en principe moindre que les SVMs, le rendant potentiellement applicable à des problèmes de plus grande taille. De

même les *Processus Gaussiens de régression* [5] [107], souffraient, au moment de la parution de notre article, d'un problème de complexité algorithmique bien plus sévère que les SVMs, ce qui en faisait un algorithme séduisant théoriquement, mais rarement applicable en pratique. De façon intéressante, les plus récents développements [56, 83] adoptent tout comme KMP une stratégie gloutonne pour grandement accélérer cette famille d'algorithmes.

[5] Anglais : *Gaussian Processes*

Chapter 9

Kernel Matching Pursuit

Matching Pursuit algorithms learn a function that is a weighted sum of basis functions, by sequentially appending functions to an initially empty basis, to approximate a target function in the least-squares sense. We show how matching pursuit can be extended to use non-squared error loss functions, and how it can be used to build kernel-based solutions to machine learning problems, while keeping control of the sparsity of the solution. We present a version of the algorithm that makes an optimal choice of both the next basis and the weights of all the previously chosen bases. Finally, links to boosting algorithms and RBF training procedures, as well as an extensive experimental comparison with SVMs for classification are given, showing comparable results with typically much sparser models.

9.1 Introduction

Recently, there has been a renewed interest for kernel-based methods, due in great part to the success of the *Support Vector Machine* approach [11, 102]. Kernel-based learning algorithms represent the function value $f(x)$ to be learned with a linear combination of terms of the form $K(x, x_i)$, where x_i is generally the input vector associated to one of the training examples, and K is a symmetric positive definite kernel function.

Support Vector Machines (SVMs) are kernel-based learning algorithms in which only a fraction of the training examples are used in the solution (these are called the Support Vectors), and where the objective of learning is to maximize a margin around the decision surface (in the case of classification).

Matching Pursuit was originally introduced in the signal-processing community as an algorithm "*that decomposes any signal into a linear expansion of waveforms that are selected from a redundant dictionary of functions.*" [64]. It is a general, greedy, sparse function approximation scheme with the squared error loss, which iteratively adds new functions (i.e. basis functions) to the linear expansion. If we take as "dictionary of functions" the functions $d_i(\cdot)$ of the form $K(\cdot, x_i)$ where x_i is the input part of a training example, then the linear expansion has essentially the same form as a Support Vector Machine. Matching Pursuit and its variants were developed primarily in the signal-processing and wavelets community, but there are many interesting links with the research on kernel-based learning algorithms developed in the machine learning community. Connections between a related algorithm (*basis pursuit* [20]) and SVMs had already been reported in [73]. More recently, [89] shows connections between Matching Pursuit, Kernel-PCA, Sparse

Kernel Feature analysis, and how this kind of greedy algorithm can be used to compress the design-matrix in SVMs to allow handling of huge data sets. Another recent work, very much related to ours, that also uses a Matching-Pursuit like algorithm is [90].

Sparsity of representation is an important issue, both for the computational efficiency of the resulting representation, and for its influence on generalization performance (see [45] and [34]). However the sparsity of the solutions found by the SVM algorithm is hardly controllable, and often these solutions are not very sparse.

Our research started as a search for a flexible alternative framework that would allow us to directly control the sparsity (in terms of number of support vectors) of the solution and remove the requirements of positive definiteness of K (and the representation of K as a dot product in a high-dimensional "feature space"[1]). It lead us to uncover connections between greedy Matching Pursuit algorithms, Radial Basis Function training procedures, and boosting algorithms (section 9.4). We will discuss these together with a description of the proposed algorithm and extensions thereof to use margin loss functions.

We first (section 9.2) give an overview of the Matching Pursuit family of algorithms (the basic version and two refinements thereof), as a general framework, taking a machine learning viewpoint. We also give a detailed description of our particular implementation that yields a choice of the next basis function to add to the expansion by minimizing simultaneously across the expansion weights and the choice of the basis function, in a computationally efficient manner.

[1] equivalent to the positive definiteness requirement

We then show (section 9.3) how this framework can be extended, to allow the use of other differentiable loss functions than the squared error to which the original algorithms are limited. This might be more appropriate for some classification problems (although, in our experiments, we have used the squared loss for many classification problems, always with successful results). This is followed by a discussion about margin loss functions, underlining their similarity with more traditional loss functions that are commonly used for neural networks.

In section 9.4 we explain how the matching pursuit family of algorithms can be used to build kernel-based solutions to machine learning problems, and how this relates to other machine learning algorithms, namely SVMs, boosting algorithms, and Radial Basis Function training procedures.

Finally, in section 9.5, we provide an experimental comparison between SVMs and different variants of Matching Pursuit, performed on artificial data, USPS digits classification, and UCI machine learning databases benchmarks. The main experimental result is that Kernel Matching Pursuit algorithms can yield generalization performance as good as Support Vector Machines, but often using significantly fewer support vectors.

9.2 Three flavors of Matching Pursuit

In this section we first describe the basic Matching Pursuit algorithm, as it was introduced by [64], but from a machine learning perspective rather than a signal processing one. We then present two successive refinements of the basic algorithm.

9.2.1 Basic Matching Pursuit

We are given l noisy observations $\{y_1, \ldots, y_l\}$ of a target function $f \in \mathcal{H}$ at points $\{x_1, \ldots, x_l\}$. We are also given a finite dictionary $\mathcal{D} = \{d_1, \ldots, d_M\}$ of M functions in a Hilbert space $\mathcal{H}$, and we are interested in sparse approximations of f that are expansions of the form

$$f_N = \sum_{n=1}^{N} \alpha_n g_n \tag{9.1}$$

where

- N is the number of *basis functions* in the expansion,
- $\{g_1, \ldots, g_N\} \subset \mathcal{D}$ shall be called the *basis* of the expansion,
- $\{\alpha_1, \ldots, \alpha_N\} \in \mathbb{R}^N$ is the set of corresponding *coefficients* of the expansion,
- f_N designs an approximation of f that uses exactly N distinct basis functions taken from the dictionary.

Notice the distinction in notation, between the dictionary functions $\{d_1, \ldots, d_M\}$ ordered as they appear in the dictionary, and the particular dictionary functions $\{g_1, \ldots, g_N\}$ ordered as they appear in the expansion f_N. There is a correspondence between the two, which can be represented by a set of indices $\Gamma = \{\gamma_1, \ldots, \gamma_N\}$ such that $g_i = d_{\gamma_i} \forall i \in \{1..N\}$ with $\gamma_i \in \{1..M\}$. Choosing a basis is equivalent to choosing a set Γ of indices.

We will also make extensive use of the following vector notations:

- For any function $f \in \mathcal{H}$ we will use $\vec{f}$ to represent the l-dimensional vector that corresponds to the evaluation of f on the l training points: $\vec{f} = (f(x_1), \ldots, f(x_l))$.
- $\vec{y} = (y_1, \ldots, y_l)$ is the *target vector*.
- $\vec{R}_N = \vec{y} - \vec{f}_N$ is the *residue*.
- $\langle \vec{h}_1, \vec{h}_2 \rangle$ will be used to represent the usual dot product between two vectors $\vec{h}_1$ and $\vec{h}_2$.
- $\|\vec{h}\|$ will be used to represent the usual L_2 (Euclidean) norm of a vector $\vec{h}$.

The algorithms described below use the dictionary functions as actual functions only when applying the learned approximation on new *test data*. During training, only their values at the training points is relevant, so that they can be understood as working entirely in an l-dimensional vector space.

The basis $\{g_1, \ldots, g_N\} \subset \mathcal{D}$ and the corresponding coefficients $\{\alpha_1, \ldots, \alpha_N\} \in \mathbb{R}^N$ are to be chosen such that they minimize the squared norm of the residue:

$$\|\vec{R}_N\|^2 = \|\vec{y} - \vec{f}_N\|^2 = \sum_{i=1}^{l} \left(y_i - f_N(x_i)\right)^2 .$$

This corresponds to reducing the usual squared "reconstruction" error. Later we will see how to extend these algorithms to other kinds of loss functions (section 9.3), but for now, we shall consider only least-squares approximations.

In the general case, when it is not possible to use particular properties of the family of functions that constitute the dictionary, finding the optimal basis $\{g_1, \ldots, g_N\}$ for a given number N of allowed basis functions implies an exhaustive search

over all possible choices of N basis functions among M ($\frac{M!}{N!(M-N)!}$ possibilities). As it would be computationally prohibitive to try all these combinations, the matching pursuit algorithm proceeds in a greedy, constructive, fashion:

It starts at stage 0 with $\vec{f_0} = 0$, and recursively appends functions to an initially empty basis, at each stage n, trying to reduce the norm of the residue $\vec{R}_n = \vec{y} - \vec{f_n}$.

Given $\vec{f_n}$ we build

$$\vec{f}_{n+1} = \vec{f_n} + \alpha_{n+1}\vec{g}_{n+1} \tag{9.2}$$

by searching for $g_{n+1} \in \mathcal{D}$ and for $\alpha_{n+1} \in \mathbb{R}$ that minimize the residual error, i.e. the squared norm of the next residue:

$$\begin{aligned} \|\vec{R}_{n+1}\|^2 &= \|\vec{y} - \vec{f}_{n+1}\|^2 \\ &= \|\vec{y} - (\vec{f_n} + \alpha_{n+1}\vec{g}_{n+1})\|^2 \\ &= \|\vec{R}_n - \alpha_{n+1}\vec{g}_{n+1}\|^2. \end{aligned}$$

Formally:

$$(g_{n+1}, \alpha_{n+1}) = \underset{(g\in\mathcal{D},\alpha\in\mathbb{R})}{\arg\min} \|\vec{R}_n - \alpha\vec{g}\|^2 \tag{9.3}$$

For any $g \in \mathcal{D}$, the α that minimizes $\|\vec{R}_n - \alpha\vec{g}\|^2$ is given by

$$\begin{aligned} \frac{\partial\|\vec{R}_n - \alpha\vec{g}\|^2}{\partial\alpha} &= 0 \\ -2\langle\vec{g}, \vec{R}_n\rangle + 2\alpha\|\vec{g}\|^2 &= 0 \\ \alpha &= \frac{\langle\vec{g}, \vec{R}_n\rangle}{\|\vec{g}\|^2} \end{aligned} \tag{9.4}$$

For this optimal value of α, we have

$$\begin{aligned} \|\vec{R}_n - \alpha\vec{g}\|^2 &= \|\vec{R}_n - \frac{\langle\vec{g},\vec{R}_n\rangle}{\|\vec{g}\|^2}\vec{g}\|^2 \\ &= \|\vec{R}_n\|^2 - 2\frac{\langle\vec{g},\vec{R}_n\rangle}{\|\vec{g}\|^2}\langle\vec{g},\vec{R}_n\rangle + \left(\frac{\langle\vec{g},\vec{R}_n\rangle}{\|\vec{g}\|^2}\right)^2\|\vec{g}\|^2 \\ &= \|\vec{R}_n\|^2 - \left(\frac{\langle\vec{g},\vec{R}_n\rangle}{\|\vec{g}\|}\right)^2 \end{aligned} \tag{9.5}$$

So the $g \in \mathcal{D}$ that minimizes expression (9.3) is the one that minimizes (9.5), which corresponds to maximizing $\left|\frac{\langle\vec{g},\vec{R}_n\rangle}{\|\vec{g}\|}\right|$. In other words, it is the function in the dictionary whose corresponding vector is "most collinear" with the current residue.

In summary, the g_{n+1} that minimizes expression (9.3) is the one that maximizes $\left|\frac{\langle\vec{g}_{n+1},\vec{R}_n\rangle}{\|\vec{g}_{n+1}\|}\right|$ and the corresponding α is $\alpha_{n+1} = \frac{\langle\vec{g}_{n+1},\vec{R}_n\rangle}{\|\vec{g}_{n+1}\|^2}$.

We have not yet specified how to choose N. Notice that, the algorithm being incremental, we don't necessarily have to fix N ahead of time and try different values to find the best one, we merely have to choose an appropriate criterion to decide when to stop adding new functions to the expansion. In the signal processing literature the algorithm is usually stopped when the *reconstruction error* $\|\vec{R}_N\|^2$ goes below a predefined given threshold. For machine learning problems, we shall rather use the error estimated on an independent validation set[2] to decide when to stop. In any case, N (even though its choice is usually indirect, determined by the early-stopping criterion) can be seen as the primary capacity-control parameter of the algorithm.

[2] or a more computationally intensive cross-validation technique if the data is scarce.

The pseudo-code for the corresponding algorithm is given in figure 9.1 (there are slight differences in the notation, in particular vector $\vec{g}_n$ in the above explanations corresponds to vector $D(.,\gamma_n)$ in the more detailed pseudo-code, and R is used to represent a temporary vector always containing the *current* residue, as we don't need to store all intermediate residues $\vec{R}_0..\vec{R}_N$. We also dropped the arrows, as we only work with vectors and matrices, seen as one and two dimensional arrays, and there is no possible ambiguity with corresponding functions).

9.2.2 Matching Pursuit with back-fitting

In the basic version of the algorithm, not only is the set of basis functions $g_{1..n}$ obtained at every step n suboptimal, but so are also their $\alpha_{1..n}$ coefficients. This can be corrected in a step often called *back-fitting* or *back-projection* and the resulting algorithm is known as *Orthogonal Matching Pursuit (OMP)* [72, 27]:

While still choosing g_{n+1} as previously (equation 9.3), we recompute the optimal set of coefficients $\alpha_{1..n+1}$ at each step instead of only the last α_{n+1}:

$$\alpha_{1..n+1}^{(n+1)} = \underset{\left(\alpha_{1..n+1}\in\mathbb{R}^{n+1}\right)}{\arg\min} \left\| \left(\sum_{k=1}^{n+1} \alpha_k \vec{g}_k \right) - \vec{y} \right\|^2 \tag{9.6}$$

Note that this is just like a linear regression with parameters $\alpha_{1..n+1}$. This *back-projection* step also has a geometrical interpretation:

Let $\mathcal{B}_n$ the sub-space of $\mathbb{R}^l$ spanned by the basis $(\vec{g}_1,\ldots,\vec{g}_n)$ and let $\mathcal{B}_n^{\perp} = \mathbb{R}^l - \mathcal{B}_n$ be its orthogonal complement. Let $P_{\mathcal{B}_n}$ and $P_{\mathcal{B}_n^{\perp}}$ denote the projection operators on these subspaces.

Then, any $\vec{g} \in \mathbb{R}^l$ can be decomposed as $\vec{g} = P_{\mathcal{B}_n}\vec{g} + P_{\mathcal{B}_n^{\perp}}\vec{g}$ (see figure 9.2).

INPUT:

- data set $\{(x_1, y_1), \ldots, (x_l, y_l)\}$
- dictionary of functions $\mathcal{D} = \{d_1, \ldots, d_M\}$
- number N of basis functions desired in the expansion (or, alternatively, a validation set to decide when to stop)

INITIALIZE: current residue vector R and dictionary matrix D

$$R \leftarrow \begin{pmatrix} y_1 \\ \vdots \\ y_l \end{pmatrix} \qquad \text{and} \qquad D \leftarrow \begin{pmatrix} d_1(x_1) & \cdots & d_M(x_1) \\ \vdots & \ddots & \vdots \\ d_1(x_l) & \cdots & d_M(x_l) \end{pmatrix}$$

FOR $n = 1..N$ (or until performance on validation set stops improving):

- $\gamma_n \leftarrow \arg\max_{k=1..M} \left| \frac{\langle D(.,k), R \rangle}{\|D(.,k)\|} \right|$
- $\alpha_n \leftarrow \frac{\langle D(.,\gamma_n), R \rangle}{\|D(.,\gamma_n)\|^2}$
- $R \leftarrow R - \alpha_n D(.,\gamma_n)$

RESULT:

The solution found is defined by $f_N(x) = \sum_{n=1}^{N} \alpha_n d_{\gamma_n}(x)$

Figure 9.1: *Basic Matching Pursuit Algorithm*

Ideally, we want the residue $\vec{R}_n$ to be as small as possible, so given the basis at step n, we want $\vec{f}_n = P_{\mathcal{B}_n}\vec{y}$ and $\vec{R}_n = P_{\mathcal{B}_n^\perp}\vec{y}$. This is what (9.6) insures.

But whenever we append the next $\alpha_{n+1}\vec{g}_{n+1}$ found by (9.3) to the expansion, we actually add its two orthogonal components:

- $P_{\mathcal{B}_n^\perp}\alpha_{n+1}\vec{g}_{n+1}$ contributes to reducing the norm of the residue.
- $P_{\mathcal{B}_n}\alpha_{n+1}\vec{g}_{n+1}$ which increases the norm of the residue.

However, as the latter part belongs to $P_{\mathcal{B}_n}$ it can be compensated for by adjusting the previous coefficients of the expansion: this is what the *back-projection* does.

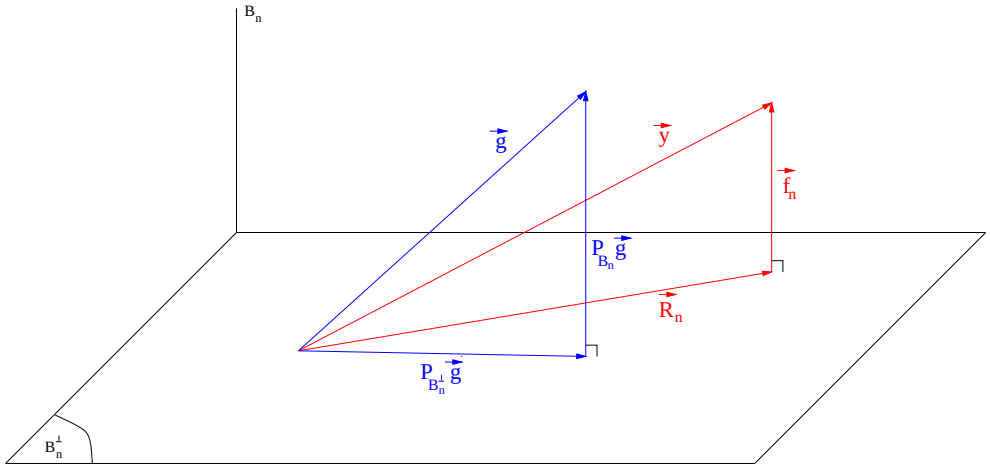

Figure 9.2: *Geometrical interpretation of Matching Pursuit and back-projection*

[27] suggest maintaining an additional orthogonal basis of the $\mathcal{B}_n$ space to facilitate this back-projection, which results in a computationally efficient algorithm[3].

9.2.3 Matching Pursuit with pre-fitting

With *back-fitting*, the choice of the function to append at each step is made regardless of the later possibility to update all weights: as we find g_{n+1} using (9.3) and *only then* optimize (9.6), we might be picking a dictionary function other than the one that would give the best fit.

Instead, it is possible to directly optimize

$$\left(g_{n+1}, \alpha_{1..n+1}^{(n+1)}\right) = \underset{(g\in\mathcal{D},\alpha_{1..n+1}\in\mathbf{R}^{n+1})}{\arg\min} \left\| \left(\sum_{k=1}^{n} \alpha_k \vec{g}_k\right) + \alpha_{n+1}\vec{g} - \vec{y} \right\|^2 \qquad (9.7)$$

We shall call this procedure *pre-fitting* to distinguish it from the former *back-fitting* (as back-fitting is done only *after* the choice of g_{n+1}).

This can be achieved almost as efficiently as *back-fitting*. Our implementation maintains a representation of both the target and all dictionary vectors as a decomposition into their projections on $\mathcal{B}_n$ and $\mathcal{B}_n^{\perp}$:

As before, let $\mathcal{B}_n = span(\vec{g}_1, \ldots, \vec{g}_n)$. We maintain at each step a representation of each dictionary vector $\vec{d}$ as the sum of two orthogonal components:

[3] In our implementation, we used a slightly modified version of this approach, described in the *pre-fitting* algorithm below.

- component $\vec{d}_{\mathcal{B}_n} = P_{\mathcal{B}_n}\vec{d}$ lies in the space $\mathcal{B}_n$ spanned by the current basis and is expressed as a linear combination of current basis vectors (it is an n-dimensional vector).
- component $\vec{d}_{\mathcal{B}_n^\perp} = P_{\mathcal{B}_n^\perp}\vec{d}$ lies in $\mathcal{B}_n$'s orthogonal complement and is expressed in the original l-dimensional vector space coordinates.

We also maintain the same representation for the target $\vec{y}$, namely its decomposition into the current expansion $\vec{f}_n \in \mathcal{B}_n$ plus the orthogonal residue $\vec{R}_n \in \mathcal{B}_n^\perp$.

Pre-fitting is then achieved easily by considering only the components in $\mathcal{B}_n^\perp$: we choose g_{n+1} as the $g \in \mathcal{D}$ whose $\vec{g}_{\mathcal{B}_n^\perp}$ is most collinear with $\vec{R}_n \in \mathcal{B}_n^\perp$. This procedure requires, at every step, only two passes through the dictionary (searching $\vec{g}_{n+1}$, then updating the representation) where basic matching pursuit requires one.

The detailed pseudo-code for this algorithm is given in figure 9.3.

9.2.4 Summary of the three variations of MP

Regardless of the computational tricks that use orthogonality properties for efficient computation, the three versions of matching pursuit differ only in the way the next function to append to the basis is chosen and the α coefficients are updated at each step n:

- **Basic version:** We find the optimal g_n to append to the basis and its optimal α_n, while keeping all other coefficients fixed (equation 9.3).

INPUT:

- data set $\{(x_1, y_1), \ldots, (x_l, y_l)\}$
- dictionary of functions $\mathcal{D} = \{d_1, \ldots, d_M\}$
- number N of basis functions desired in the expansion (or, alternatively, a validation set to decide when to stop)

INITIALIZE: current residue vector R and dictionary components $D_{\mathcal{B}^\perp}$ and $D_{\mathcal{B}}$

$$R \leftarrow \begin{pmatrix} y_1 \\ \vdots \\ y_l \end{pmatrix} \qquad \text{and} \qquad D_{\mathcal{B}^\perp} \leftarrow \begin{pmatrix} d_1(x_1) & \cdots & d_M(x_1) \\ \vdots & \ddots & \vdots \\ d_1(x_l) & \cdots & d_M(x_l) \end{pmatrix}$$

$D_{\mathcal{B}}$ is initially empty, and gets appended an additional row at each step (thus, ignore the expressions that involve $D_{\mathcal{B}}$ during the first iteration when $n = 1$)

FOR $n = 1..N$ (or until performance on validation set stops improving):

- $\gamma_n \leftarrow \underset{k=1..M}{\arg\max} \left| \dfrac{\langle D_{\mathcal{B}^\perp}(.,k), R\rangle}{\|D_{\mathcal{B}^\perp}(.,k)\|} \right|$
- $\alpha_n \leftarrow \dfrac{\langle D_{\mathcal{B}^\perp}(.,\gamma_n), R\rangle}{\|D_{\mathcal{B}^\perp}(.,\gamma_n)\|^2}$
- the $\mathcal{B}^\perp$ component of $\alpha_n d_{\gamma_n}$ reduces the residue:
  $R \leftarrow R - \alpha_n D_{\mathcal{B}^\perp}(.,\gamma_n)$
- compensate for the $\mathcal{B}$ component of $\alpha_n d_{\gamma_n}$ by adjusting previous α:
  $(\alpha_1, \ldots, \alpha_{n-1}) \leftarrow (\alpha_1, \ldots, \alpha_{n-1}) - \alpha_n D_{\mathcal{B}}(.,\gamma_n)'$
- Now update the dictionary representation to take into account the new basis function d_{γ_n}:
  FOR $i = 1..M$ AND $i \neq \gamma_n$:

  $$\beta_i \leftarrow \frac{\langle D_{\mathcal{B}^\perp}(.,\gamma_n), D_{\mathcal{B}^\perp}(.,i)\rangle}{\|D_{\mathcal{B}^\perp}(.,\gamma_n)\|^2}$$
  $$D_{\mathcal{B}^\perp}(.,i) \leftarrow D_{\mathcal{B}^\perp}(.,i) - \beta_i D_{\mathcal{B}^\perp}(.,\gamma_n)$$
  $$D_{\mathcal{B}}(.,i) \leftarrow D_{\mathcal{B}}(.,i) - \beta_i D_{\mathcal{B}}(.,\gamma_n)$$

- $D_{\mathcal{B}^\perp}(.,\gamma_n) \leftarrow 0$
- $D_{\mathcal{B}}(.,\gamma_n) \leftarrow 0$
- $\beta_{\gamma_n} \leftarrow 1$
- $D_{\mathcal{B}} \leftarrow \begin{pmatrix} D_{\mathcal{B}} \\ \beta_1, \ldots, \beta_M \end{pmatrix}$

RESULT:

The solution found is defined by $f_N(x) = \sum_{n=1}^{N} \alpha_n d_{\gamma_n}(x)$

Figure 9.3: *Matching Pursuit with pre-fitting*

- **back-fitting version:** We find the optimal g_n while keeping all coefficients fixed (equation 9.3). Then we find the optimal set of coefficients $\alpha_{1..n}^{(n)}$ for the new basis (equation 9.6).
- **pre-fitting version:** We find at the same time the optimal g_n and the optimal set of coefficients $\alpha_{1..n}^{(n)}$ (equation 9.7).

When making use of orthogonality properties for efficient implementations of the back-fitting and pre-fitting version (as in our previously described implementation of the pre-fitting algorithm), all three algorithms have a computational complexity of the same order $\mathcal{O}(N \cdot M \cdot l)$.

9.3 Extension to non-squared error loss

9.3.1 Gradient descent in function space

It has already been noticed that boosting algorithms are performing a form of gradient descent in function space with respect to particular loss functions [79, 65]. Following [39], the technique can be adapted to extend the Matching Pursuit family of algorithms to optimize arbitrary differentiable loss functions, instead of doing least-squares fitting.

Given a loss function $L(y_i, f_n(x_i))$ that computes the cost of predicting a value of $f_n(x_i)$ when the true target was y_i, we use an alternative residue $\tilde{R}_n$ rather than the usual $\vec{R}_n = \vec{y} - \vec{f}_n$ when searching for the next dictionary element to append to the basis at each step.

$\tilde{R}_n$ is the direction of steepest descent (the gradient) in function space (evaluated at the data points) with respect to L:

$$\tilde{R}_n = \left(-\frac{\partial L\left(y_1, \vec{f}_n(x_1)\right)}{\partial \vec{f}_n(x_1)}, \ldots, -\frac{\partial L\left(y_l, \vec{f}_n(x_l)\right)}{\partial \vec{f}_n(x_l)} \right) \tag{9.8}$$

i.e. $\vec{g}_{n+1}$ is chosen such that it is most collinear with this gradient:

$$g_{n+1} = \arg\max_{g \in \mathcal{D}} \left| \frac{\langle \vec{g}_{n+1}, \tilde{R}_n \rangle}{\|\vec{g}_{n+1}\|} \right| \tag{9.9}$$

A line-minimization procedure can then be used to find the corresponding coefficient

$$\alpha_{n+1} = \arg\min_{\alpha \in \mathbb{R}} \sum_{i=1}^{l} L\left(y_i, f_n(x_i) + \alpha \vec{g}_{n+1}(x_i)\right) \tag{9.10}$$

This would correspond to *basic matching pursuit* (notice how the original squared-error algorithm is recovered when L is the squared error: $L(a,b) = \frac{1}{2}(a-b)^2$).

It is also possible to do *back-fitting*, by re-optimizing all $\alpha_{1..n+1}$ (instead of only α_{n+1}) to minimize the target cost (with a conjugate gradient optimizer for instance):

$$\alpha_{1..n+1}^{(n+1)} = \arg\min_{\left(\alpha_{1..n+1} \in \mathbb{R}^{n+1}\right)} \sum_{i=1}^{l} L\left(y_i, \sum_{k=1}^{n+1} \alpha_k g_k(x_i)\right) \tag{9.11}$$

As this can be quite time-consuming (we cannot use any orthogonality property in this general case), it may be desirable to do it every few steps instead of every single step. The corresponding algorithm is described in more details in the pseudo-code of figure 9.4 (as previously there are slight differences in the notation, in particular g_k in the above explanation corresponds to vector $D(., \gamma_k)$ in the more detailed pseudo-code).

Finally, let's mention that it should in theory also be possible to do *pre-fitting* with an arbitrary loss functions, but finding the optimal $\{g_{k+1} \in \mathcal{D}, \alpha_{1..k+1} \in \mathbb{R}^{k+1}\}$ in the general case (when we cannot use any orthogonal decomposition) would involve solving equation 9.11 in turn for *each* dictionary function in order to choose the next one to append to the basis, which is computationally prohibitive.

9.3.2 Margin loss functions versus traditional loss functions for classification

Now that we have seen how the matching pursuit family of algorithms can be extended to use arbitrary loss functions, let us discuss the merits of various loss functions.

In particular the relationship between loss functions and the notion of *margin* is of primary interest here, as we wanted to build an alternative to SVMs[4].

While the original notion of margin in classification problems comes from the geometrically inspired hard-margin of linear SVMs (the smallest Euclidean distance between the decision surface and the training points), a slightly different perspective has emerged in the boosting community along with the notion of margin loss function. The margin quantity $m = y\hat{f}(x)$[5] of an individual data point (x, y), with $y \in \{-1, +1\}$ can be understood as a confidence measure of its classification by

[4] whose good generalization abilities are believed to be due to margin-maximization.

[5] Called the "functional margin". Strictly speaking, in boosting the margin has to be normalized by the L_1 norm of the coefficients: $m = \frac{y\hat{f}(x)}{\|\alpha\|_1}$ where $\|\alpha\|_1 = \sum \alpha_i$. As it uses the L_1 norm it's sometimes called the L_1 margin, as opposed to the geometrical Euclidean L_2 margin of SVMs. But the purpose of our discussion is qualitative, so let us ignore these specific details for now.

INPUT:

- data set $\{(x_1, y_1), \ldots, (x_l, y_l)\}$
- dictionary of functions $\mathcal{D} = \{d_1, \ldots, d_M\}$
- number N of basis functions desired in the expansion (or, alternatively, a validation set to decide when to stop)
- how often to do a full back-fitting: every p update steps
- a loss function L

INITIALIZE: current approximation $\hat{f}$ and dictionary matrix D
Notice that $\hat{f}$ here is the *current approximation vector*, which changes at every step n, so that $\hat{f}_i$ here corresponds to $f_n(x_i)$ in the accompanying text.

$$\hat{f} = \begin{pmatrix} \hat{f}_0 \\ \vdots \\ \hat{f}_l \end{pmatrix} \leftarrow 0 \qquad \text{and} \qquad D \leftarrow \begin{pmatrix} d_1(x_1) & \cdots & d_M(x_1) \\ \vdots & \ddots & \vdots \\ d_1(x_l) & \cdots & d_M(x_l) \end{pmatrix}$$

FOR $n = 1..N$ (or until performance on validation set stops improving):

- $\tilde{R} \leftarrow \begin{pmatrix} -\frac{\partial L(y_1, \hat{f}_1)}{\partial \hat{f}_1} \\ \vdots \\ -\frac{\partial L(y_l, \hat{f}_n)}{\partial \hat{f}_l} \end{pmatrix}$
- $\gamma_n \leftarrow \underset{k=1..M}{\arg\max} \left| \frac{\langle D(.,k), \tilde{R} \rangle}{\|D(.,k)\|} \right|$
- If n *is not* a multiple of p do a simple line minimization:

$$\alpha_n \leftarrow \underset{\alpha \in \mathbb{R}}{\arg\min} \sum_{i=1}^{l} L(y_i, \hat{f}_i + \alpha D(i, \gamma_n))$$

  and update $\hat{f}$: $\hat{f} \leftarrow \hat{f} + \alpha_n D(., \gamma_n)$
- If n *is* a multiple of p do a full back-fitting (for ex. with gradient descent):

$$\alpha_{1..n} \leftarrow \underset{\alpha_{1..n} \in \mathbb{R}^n}{\arg\min} \sum_{i=1}^{l} L(y_i, \sum_{k=1}^{n} \alpha_k D(i, \gamma_k))$$

  and recompute $\hat{f} \leftarrow \sum_{k=1}^{n} \alpha_k D(., \gamma_k)$

RESULT:

The solution found is defined by $f_N(x) = \sum_{n=1}^{N} \alpha_n d_{\gamma_n}(x)$

Figure 9.4: *Back-fitting Matching Pursuit Algorithm with non-squared loss*

function $\hat{f}$, while the class decided for is given by sign($\hat{f}(x)$). A *margin loss function* is simply a function of this margin quantity m that is being optimized.

It is possible to formulate SVM training such as to show the SVM margin loss function:

Let φ be the mapping into the "feature-space" of SVMs, such that $\langle \varphi(x_i), \varphi(x_j) \rangle = K(x_i, x_j)$.

The SVM solution can be expressed in this feature space as $\hat{f}(x) = \langle w, \varphi(x) \rangle + b$ where $w = \sum_{x_i \in \mathcal{SV}} \alpha_i y_i \varphi(x_i)$, $\mathcal{SV}$ being the set of support vectors.

The SVM problem is usually formulated as minimizing

$$\|w\|^2 + C \sum_{i=1}^{l} \xi_i \tag{9.12}$$

subject to constraints $y_i(\langle w, x_i \rangle + b) \geq 1 - \xi_i$ and $\xi_i \geq 0, \forall i$. C is the "box-constraint" parameter of SVMs, trading off margin with training errors.

The two constraints for each ξ_i can be rewritten as a single one:

$\xi_i \geq \max(0, 1 - y_i(\langle w, x_i \rangle + b))$

or equivalently: $\xi_i \geq [1 - y_i \hat{f}(x_i)]_+$.

The notation $[x]_+$ is to be understood as the function that gives $[x]_+ = x$ when $x > 0$ and 0 otherwise.

As we are minimizing an expression containing a term $\sum_{i=1}^{l} \xi_i$, the inequality constraints over the ξ_i can be changed into equality constraints: $\xi_i = [1 - y_i \hat{f}(x_i)]_+$. Replacing ξ_i in equation (9.12), and multiplying by C we get the following al-

ternative formulation of the SVM problem, where there are no more explicit constraints (they are implicit in the criterion optimized):

Minimize

$$\sum_{i=1}^{l}[1 - y_i\hat{f}(x_i)]_+ + \frac{1}{C}\|w\|^2. \tag{9.13}$$

Let $m = y\hat{f}(x)$ the *individual margin* at point x. (9.13) is clearly the sum of a margin loss function and a regularization term.

It is interesting to compare this *margin loss function* to those used in *boosting algorithms* and to the more traditional cost functions. The loss functions that boosting algorithms optimize are typically expressed as functions of m. Thus AdaBoost [79] uses an exponential (e^{-m}) margin loss function, LogitBoost [40] uses the negative binomial log-likelihood, $\log_2(1 + e^{-2m})$, whose shape is similar to a smoothed version of the soft-margin SVM loss function $[1 - m]_+$, and Doom II [65] approximates a theoretically motivated margin loss with $1-\tanh(m)$. As can be seen in Figure 9.5 (left), all these functions encourage large positive margins, and differ mainly in how they penalize large negative ones. In particular $1 - \tanh(x)$ is expected to be more robust, as it won't penalize outliers to excess.

It is enlightening to compare these with the more traditional loss functions that have been used for neural networks in classification tasks (i.e. $y \in \{-1, +1\}$), when we express them as functions of m.

- Squared loss: $(\hat{f}(x) - y)^2 = (1 - m)^2$

- Squared loss after $\tanh$ with modified target[6]:
  $(\tanh(\hat{f}(x)) - 0.65y)^2 = (0.65 - \tanh(m))^2$

Both are illustrated on figure 9.5 (right). Notice how the squared loss after $\tanh$ appears similar to the margin loss function used in Doom II, except that it slightly increases for large positive margins, which is why it behaves well and does not saturate even with unconstrained weights (boosting algorithms impose further constraints on the weights, here denoted α's).

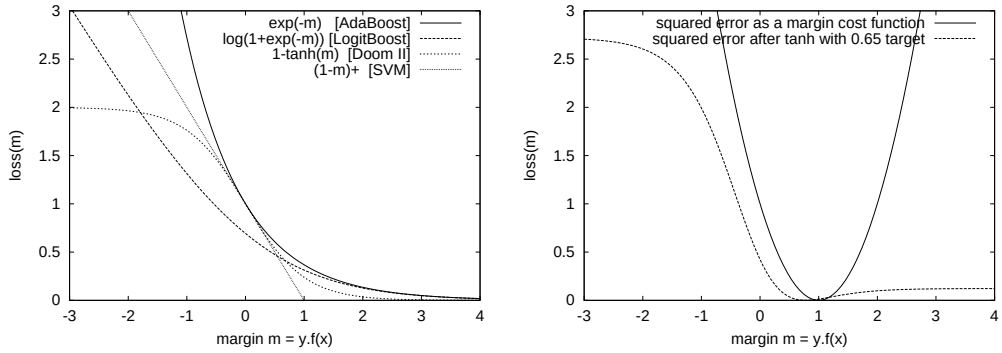

Figure 9.5: *Boosting and SVM margin loss functions (left) vs. traditional loss functions (right) viewed as functions of the margin. Interestingly the last-born of the margin motivated loss functions (used in Doom II) is similar to the traditional squared error after* $\tanh$.

[6] 0.65 is approximately the point of maximum second derivative of the $\tanh$, and was advocated by [59] as a target value for neural networks, to avoid saturating the output units while taking advantage of the non-linearity for improving discrimination of neural networks.

9.4 Kernel Matching Pursuit and links with other paradigms

9.4.1 Matching pursuit with a kernel-based dictionary

Kernel Matching Pursuit (KMP) is simply the idea of applying the Matching Pursuit family of algorithms to problems in machine learning, using a kernel-based dictionary:

Given a kernel function $K : \mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R}$, we use as our dictionary the kernel centered on the training points: $\mathcal{D} = \{d_i = K(\cdot, x_i) | i = 1..l\}$. Optionally, the constant function can also be included in the dictionary, which accounts for a bias term b: the functional form of approximation $\vec{f}_N$ then becomes

$$\vec{f}_N(x) = b + \sum_{n=1}^{N} \alpha_n K(x, x_{\gamma_n}) \tag{9.14}$$

where the $\gamma_{1..N}$ are the indices of the "support points". During training we only consider the values of the dictionary functions at the training points, so that it amounts to doing Matching in a vector-space of dimension l.

When using a squared error loss[7], the complexity of all three variations of KMP (basic, back-fitting and pre-fitting) is $\mathcal{O}(N.M.l) = \mathcal{O}(N.l^2)$ if we use all the training data as candidate support points. But it is also possible to use a random subset of the training points as support candidates (which yields a $M < l$).

[7]The algorithms generalized to arbitrary loss functions can be much more computationally intensive, as they imply a non-quadratic optimization step.

We would also like to emphasize the fact that the use of a dictionary gives a lot of *additional flexibility* to this framework, as it is possible to include any kind of function into it, in particular:

- There is no restriction on the shape of the kernel (no positive-definiteness constraint, could be asymmetrical, etc.).
- The dictionary could include more than a single fixed kernel shape: it could mix different kernel types to choose from at each point, allowing for instance the algorithm to choose among several widths of a Gaussian for each support point (a similar extension has been proposed for SVMs by [106]).
- Similarly, the dictionary could easily be used to constrain the algorithm to use a kernel shape specific to each *class*, based on prior-knowledge.
- The dictionary can incorporate non-kernel based functions (we already mentioned the constant function to recover the bias term b, but this could also be used to incorporate prior knowledge). In this later aspect, the work of [91] on *semi-parametric* SVMs offers an interesting advance in that direction, within the SVM framework. This remains a very interesting avenue for further research.
- For huge data sets, a reduced subset can be used as the dictionary to speed up the training.

However in this study, we restrict ourselves to using a single fixed kernel, so that the resulting functional form is the same as the one obtained with standard SVMs.

9.4.2 Similarities and differences with SVMs

The functional form (9.14) is very similar to the one obtained with the *Support Vector Machine* (SVM) algorithm [11], the main difference being that SVMs impose further constraints on $\alpha_{1..N}$.

However the quantity optimized by the SVM algorithm is quite different from the KMP greedy optimization, especially when using a squared error loss. Consequently the support vectors and coefficients found by the two types of algorithms are usually different (see our experimental results in section 9.5).

Another important difference, and one that was a motivation for this research, is that in KMP, capacity control is achieved by *directly* controlling the sparsity of the solution, i.e. the number N of support vectors, whereas the capacity of SVMs is controlled through the box-constraint parameter C, which has an indirect and hardly controllable influence on sparsity. See [45] for a discussion on the merits of sparsity and margin, and ways to combine them.

9.4.3 Link with Radial Basis Functions

Squared-error KMP with a Gaussian kernel and pre-fitting appears to be identical to a particular Radial Basis Functions training algorithm called *Orthogonal Least Squares RBF* [21] (OLS-RBF).

In [82] SVMs were compared to "classical RBFs", where the RBF centers were chosen by unsupervised k-means clustering, and SVMs gave better results. To our knowledge, however, there has been no experimental comparison between OLS-

RBF and SVMs, although their resulting functional forms are very much alike. Such an empirical comparison is one of the contributions of this paper. Basically our results (section 9.5) show OLS-RBF (i.e. squared-error KMP) to perform as well as Gaussian SVMs, while allowing a tighter control of the number of support vectors used in the solution.

9.4.4 Boosting with kernels

KMP in its basic form generalized to using non-squared error is also very similar to boosting algorithms [35, 40], in which the chosen class of weak learners would be the set of kernels centered on the training points. These algorithms differ mainly in the loss function they optimize, which we have already discussed in section 9.3.2.

In this respect, a very much related research is the work of [88] on *Leveraged Vector Machines*. The proposed boosting algorithm also builds support vector solutions to classification problems using kernel-based weak learners and similarly shows good performance with typically sparser models.

9.4.5 Matching pursuit versus Basis pursuit

Basis Pursuit [20] is an alternative algorithm that essentially attempts to achieve the same goal as Matching Pursuit, namely to build a sparse approximation of a target function using a possibly over-complete dictionary of functions. It is some-

times believed to be a superior approach[8] because contrary to Matching Pursuit, which is a greedy algorithm, Basis Pursuit uses Linear Programming techniques to find the *exact* solution to the following problem:

$$\alpha_{1..M} = \underset{\alpha_{1..M} \in \mathbb{R}^M}{\arg\min} \left\| \left(\sum_{k=1}^{M} \alpha_k \vec{d}_k \right) - \vec{y} \right\|^2 + \lambda \|\alpha\|_1 \tag{9.15}$$

where $\|\alpha\|_1 = \sum_{k=1}^{M} |\alpha_k|$ is used to represent the L_1 norm of $\alpha_{1..M}$.

The added $\lambda\|\alpha\|_1$ penalty term will drive a large number of the coefficients to 0 and thus lead to a sparse solution, whose sparsity can be controlled by appropriately tuning the hyper-parameter λ.

However we would like to point out that, as far as the primary goal is good *sparsity*, i.e. using the smallest *number* of basis functions in the expansion, both algorithms are approximate: Matching Pursuit is greedy, while Basis Pursuit finds an exact solution, but to an approximate problem (the exact problem could be formulated as solving an equation similar to (9.15) but where the L_0 norm would be used in the penalty term instead of the L_1 norm).

In addition Matching Pursuit had a number of advantages over Basis Pursuit in our particular setting:

- It is very simple and computationally efficient, while Basis Pursuit requires the use of sophisticated Linear Programming techniques to tackle large problems.

[8] It is possible to find artificial pathological cases where Matching Pursuit breaks down, but this doesn't seem to be a problem for real-world problems, especially when using the back-fitting or pre-fitting improvements of the original algorithm.

- It is constructive, adding the basis functions one by one to the expansion, which allows us to use a simple early-stopping procedure to control optimal sparsity. In contrast, a Basis Pursuit approach implies having to tune the hyper-parameter λ, running the optimization several times with different values to find the best possible choice.

But other than that, we might as well have used Basis Pursuit and would probably have achieved very similar experimental results. We should also mention the works of [73] which draws an interesting parallel between Basis Pursuit and SVMs, as well as [46] who use Basis Pursuit with ANOVA kernels to obtain sparse models with improved interpretability.

9.4.6 Kernel Matching pursuit versus Kernel Perceptron

The perceptron algorithm [77] and extensions thereof [42] are among the simplest algorithms for building linear classifiers. As it is a dot-product based algorithm, the *kernel trick* introduced by [1] readily applies, allowing a straightforward extension to build non-linear decision surfaces in input-space, in the same way this trick is used for SVMs.

For recent research on the Kernel Perceptron, see the very interesting work of [37], and also [45] who derive theoretical bounds on their generalization error. Kernel Perceptrons are shown to produce solutions that are typically more sparse than SVMs while retaining comparable recognition accuracies.

Both Kernel Matching Pursuit and Kernel Perceptron appear to be simple (they do not involve complex quadratic or linear programming) and efficient greedy

algorithms for finding sparse kernel-based solutions to classification problems. However there are major differences between the two approaches:

- Kernel Matching Pursuit does not use the Kernel trick to implicitly work in a higher-dimensional mapped feature-space: it works directly in input-space. Thus it is possible to use specific Kernels that don't necessarily satisfy Mercer's conditions. This is especially interesting if you think of K as a kind of *similarity measure* between input patterns, that could be engineered to include prior knowledge, or even learned, as it is not always easy, nor desirable, to enforce positive-definiteness in this perspective.

- The perceptron algorithm is initially a classification algorithm, while Matching Pursuit is originally more of a regression algorithm (approximation in the least-squares sense), although the proposed extension to non-squared loss and the discussion on margin-loss functions (see section 9.3) further blurs this distinction. The main reason why we use this algorithm for binary classification tasks rather than regression, although the latter would seem more natural, is that our primary purpose was to compare its performance to classification-SVMs[9].

- Similar to SVMs, the solution found by the Kernel Perceptron algorithm depends only on the retained support vectors, while the coefficients learned by Kernel Matching Pursuit depend on *all* training data, not only on the set of support vectors chosen by the algorithm. This implies that current

[9] A comparison with regression-SVMs should also prove very interesting, but the question of how to compare two regression algorithms that do not optimize the same loss (squared loss for KMP, versus ϵ-insensitive loss for SVMs) first needs to be addressed.

theoretical results on generalization bounds that are derived for sparse SVM or Perceptron solutions [102, 103, 61, 34, 45] cannot be readily applied to KMP. On the other hand, KMP solutions may require less support vectors than Kernel Perceptron for precisely this same reason: the information on all data points is used, without the need that they appear as support vectors in the solution.

9.5 Experimental results on binary classification

Throughout this section:

- any mention of KMP without further specification of the loss function means least-squares KMP (also sometimes written *KMP-mse*)
- *KMP-tanh* refers to KMP using squared error after a hyperbolic tangent with modified targets (which behaves more like a typical *margin loss function* as we discussed earlier in section 9.3.2).
- Unless otherwise specified, we used the *pre-fitting matching pursuit* algorithm of figure 9.3 to train least-squares KMP.
- To train *KMP-tanh* we always used the *back-fitting matching pursuit with non-squared loss* algorithm of figure 9.4 with a conjugate gradient optimizer to optimize the $\alpha_{1..n}$[10].

[10] We tried several frequencies at which to do full back-fitting, but it did not seem to have a strong impact, as long as it was done often enough.

9.5.1 2D experiments

Figure 9.6 shows a simple 2D binary classification problem with the decision surface found by the three versions of squared-error KMP (basic, back-fitting and pre-fitting) and a hard-margin SVM, when using the same Gaussian kernel.

We fixed the number N of support points for the pre-fitting and back-fitting versions to be the same as the number of support points found by the SVM algorithm. The aim of this experiment was to illustrate the following points:

- Basic KMP, after 100 iterations, during which it mostly cycled back to previously chosen support points to improve their weights, is still unable to separate the data points. This shows that the back-fitting and pre-fitting versions are a useful improvement, while the basic algorithm appears to be a bad choice if we want sparse solutions.

- The back-fitting and pre-fitting KMP algorithms are able to find a reasonable solution (the solution found by pre-fitting looks slightly better in terms of margin), but choose different support vectors than SVM, that are not necessarily close to the decision surface (as they are in SVMs). It should be noted that the *Relevance Vector Machine* [98] similarly produces[11] solutions in which the *relevance vectors* do not lie close to the border.

Figure 9.7, where we used a simple dot-product kernel (i.e. linear decision surfaces), illustrates a problem that can arise when using least-squares fit: since the squared error penalizes large positive margins, the decision surface is drawn to-

[11] however in a much more computationally intensive fashion.

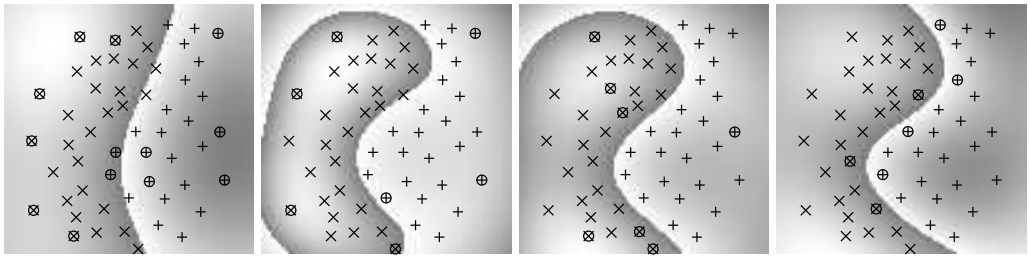

Figure 9.6: *From left to right: 100 iterations of basic KMP, 7 iterations of KMP back-fitting, 7 iterations of KMP pre-fitting, and SVM. Classes are $+$ and $\times$. Support vectors are circled. Pre-fitting KMP and SVM appear to find equally reasonable solutions, though using different support vectors. Only SVM chooses its support vectors close to the decision surface. Back-fitting chooses yet another support set, and its decision surface appears to have a slightly worse margin. As for basic KMP, after 100 iterations during which it mostly cycled back to previously chosen support points to improve their weights, it appears to use more support vectors than the others while still being unable to separate the data points, and is thus a bad choice if we want sparse solutions.*

wards the cluster on the lower right, at the expense of a few misclassified points. As expected, the use of a $\tanh$ loss function appears to correct this problem.

9.5.2 US Postal Service Database

The main purpose of this experiment was to complement the results of [82] with those obtained using KMP-mse, which, as already mentioned, is equivalent to orthogonal least squares RBF [21].

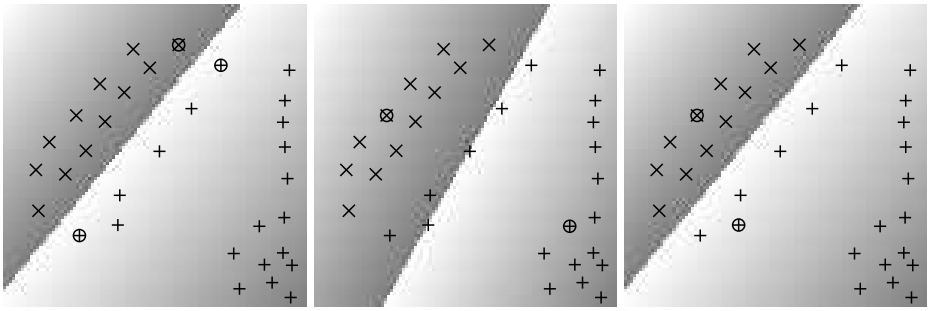

Figure 9.7: *Problem with least squares fit that leads KMP-mse (center) to misclassify points, but does not affect SVMs (left), and is successfully treated by KMP-tanh (right).*

In [82] the RBF centers were chosen by unsupervised *k-means* clustering, in what they referred to as "Classical RBF", and a gradient descent optimization procedure was used to train the kernel weights.

We repeated the experiment using KMP-mse (equivalent to OLS-RBF) to find the support centers, with the same Gaussian Kernel and the same training set (7300 patterns) and independent test set (2007 patterns) of preprocessed handwritten digits. Table 9.1 gives the number of errors obtained by the various algorithms on the tasks consisting of discriminating each digit versus all the others (see [82] for more details). No validation data was used to choose the number of bases (support vectors) for the KMP. Instead, we trained with N equal to the number of support vectors obtained with the SVM, and also with N equal to half that number, to see whether a sparser KMP model would still yield good results. As can be seen, results obtained with KMP are comparable to those obtained for SVMs, contrarily to the results obtained with k-means RBFs, and there is only

a slight loss of performance when using as few as half the number of support vectors.

Table 9.1: *USPS Results: number of errors on the test set (2007 patterns), when using the same number of support vectors as found by SVM (except last row which uses half #sv). Squared error KMP (same as OLS-RBF) appears to perform as well as SVM.*

Digit class	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
#sv	274	104	377	361	334	388	236	235	342	263
SVM	16	8	25	19	29	23	14	12	25	16
k-means RBF	20	16	43	38	46	31	15	18	37	26
KMP (same #sv)	15	15	26	17	30	23	14	14	25	13
KMP (half #sv)	16	15	29	27	29	24	17	16	28	18

9.5.3 Benchmark datasets

We did some further experiments, on 5 well-known datasets from the the UCI machine learning databases, using Gaussian kernels of the form

$$K(x_1, x_2) = e^{-\frac{\|x_1 - x_2\|^2}{\sigma^2}}.$$

A first series of experiments used the machinery of the Delve [76] system to assess performance on the Mushrooms dataset. Hyper-parameters (the σ of the kernel, the box-constraint parameter C for soft-margin SVM and the number of support

points for KMP) were chosen automatically for each run using 10-fold cross-validation.

The results for varying sizes of the training set are summarized in table 9.2. The p-values reported in the table are those computed automatically by the Delve system[12].

Table 9.2: *Results obtained on the mushrooms data set with the Delve system. KMP requires less support vectors, while none of the differences in error rates are signifi cant.*

size of train	KMP error	SVM error	p-value (t-test)	KMP #s.v.	SVM #s.v.
64	6.28%	4.54%	0.24	17	63
128	2.51%	2.61%	0.82	28	105
256	1.09%	1.14%	0.81	41	244
512	0.20%	0.30%	0.35	70	443
1024	0.05%	0.07%	0.39	127	483

For Wisconsin Breast Cancer, Sonar, Pima Indians Diabetes and Ionosphere, we used a slightly different procedure.

The σ of the Kernel was first fixed to a reasonable value for the given data set[13].

[12]For each size, the delve system did its estimations based on 8 disjoint training sets of the given size and 8 disjoint test sets of size 503, except for 1024, in which case it used 4 disjoint training sets of size 1024 and 4 test sets of size 1007.

[13]These were chosen by trial and error using SVMs with a validation set and several values of C, and keeping what seemed the best σ, thus this choice was made at the advantage of SVMs

Then we used the following procedure: the dataset was randomly split into three equal-sized subsets for training, validation and testing. SVM, KMP-mse and KMP-tanh were then trained on the training set while the validation set was used to choose the optimal box-constraint parameter C for SVMs[14], and to do early stopping (decide on the number N of s.v.) for KMP. Finally the trained models were tested on the independent test set.

This procedure was repeated 50 times over 50 different random splits of the dataset into train/validation/test to estimate confidence measures (p-values were computed using the resampled t-test studied in [68]). Table 9.3 reports the average error rate measured on the test sets, and the rounded average number of support vectors found by each algorithm.

As can be seen from these experiments, the error rates obtained are comparable, but the KMP versions appear to require much fewer support vectors than SVMs. On these datasets, however (contrary to what we saw previously on 2D artificial data), KMP-tanh did not seem to give any significant improvement over KMP-mse. Even in other experiments where we added label noise, KMP-tanh didn't seem to improve generalization performance[15].

(although they did not seem too sensitive to it) rather than KMP. The values used were: 4.0 for Wisconsin Breast Cancer, 6.0 for Pima Indians Diabetes, 2.0 for Ionosphere and Sonar.

[14] Values of 0.02, 0.05, 0.07, 0.1, 0.5, 1, 2, 3, 5, 10, 20, 100 were tried for C.

[15] We do not give a detailed account of these experiments here, as their primary intent was to show that the $\tanh$ error function could have an advantage over squared error in presence of label noise, but the results were inconclusive.

Table 9.3: *Results on 4 UCI datasets. Again, error rates are not significantly different (values in parentheses are the p-values for the difference with SVMs), but KMPs require much fewer support vectors.*

Dataset	SVM error	KMP-mse error	KMP-tanh error	SVM #s.v.	KMP-mse #s.v.	KMP-tanh #s.v.
Wisc. Cancer	3.41%	3.40% (0.49)	3.49% (0.45)	42	7	21
Sonar	20.6%	21.0% (0.45)	26.6% (0.16)	46	39	14
Pima Indians	24.1%	23.9% (0.44)	24.0% (0.49)	146	7	27
Ionosphere	6.51%	6.87% (0.41)	6.85% (0.40)	68	50	41

9.6 Conclusion

We have shown how Matching Pursuit provides an interesting and flexible framework to build and study alternative kernel-based methods, how it can be extended to use arbitrary differentiable loss functions, and how it relates to SVMs, RBF training procedures, and boosting methods.

We have also provided experimental evidence that such greedy constructive algorithms can perform as well as SVMs, while allowing a better control of the sparsity of the solution, and thus often lead to solutions with far fewer support vectors.

It should also be mentioned that the use of a dictionary gives a lot of flexibility, as it can be extended in a direct and straightforward manner, allowing for instance, to mix several kernel shapes to choose from (similar to the SVM extension proposed by [106]), or to include other non-kernel functions based on prior knowledge

(similar to the work of [91] on semi-parametric SVMs). This is a promising avenue for further research.

In addition to the computational advantages brought by the sparsity of the models obtained with the kernel matching pursuit algorithms, one might suspect that generalization error also depends (monotonically) on the number of support vectors (other things being equal). This was observed empirically in our experiments, but future work should attempt to obtain generalization error bounds in terms of the number of support vectors. Note that leave-one-out SVM bounds [102, 103] cannot be used here because the α_i coefficients depend on all the examples, not only a subset (the support vectors). Sparsity has been successfully exploited to obtain bounds for other SVM-like models [61, 34, 45], in particular the kernel perceptron, again taking advantage of the dependence on a subset of the examples. A related direction to pursue might be to take advantage of the data-dependent structural risk minimization results of [84].

Chapitre 10

Présentation du deuxième article

P. Vincent et Y. Bengio. **K-Local Hyperplane and Convex Distance Nearest Neighbor Algorithms.** Publié en 2002 dans *Advances in Neural Information Processing Systems 14*, aux éditions MIT Press.

10.1 Objectifs de cette recherche

En entraînant des SVMs à noyau sur des problèmes concrets en haute dimension, on se rend compte que bien souvent les solutions obtenues ne sont guère clairsemées. Quand on utilise des noyaux localisés tels que les noyaux Gaussiens (le noyau le plus souvent utilisé en pratique avec succès), on ne peut dès lors s'empêcher de trouver une forte ressemblance, du moins dans la forme du résultat, avec les techniques non-paramétriques classiques telles que celle des fenêtres de Parzen ou celle des K plus proches voisins (KNN). Mais les expériences sur

des problèmes réels indiquent souvent des résultats bien meilleurs avec les SVMs qu'avec KNN.

Nous avons voulu chercher à comprendre un peu mieux pourquoi, et à voir si on ne pouvait pas d'une certaine manière "réparer" KNN, afin que les performances en haute dimension égalent ou dépassent celles des SVMs. La différence qualitative entre les surfaces de décisions lisses généralement produites par les SVMs à noyau et la surface en "zig-zag" produite par KNN, telles qu'illustrées à la Figure 11.1 nous a fourni l'intuition de départ, lançant une réflexion sur la notion de maximisation de la *marge locale* dans l'espace d'entrée que nous évoquons dans l'introduction de l'article, et nous amenant à définir la distance d'un point à une classe comme la distance à la variété linéaire supportée par les voisins de ce point appartenant à ladite classe.

10.2 Contribution au domaine

La contribution de cet article est double :

– La conception de deux variantes de l'algorithme des K plus proches voisins apportant des améliorations considérables de la performance sur certains problèmes en haute dimension, au point de battre les SVMs.
– La mise en évidence que dans les problèmes en haute dimension, il peut y avoir beaucoup à gagner en tenant compte des directions principales locales dans les données. Cela semble corroborer l'hypothèse de concentration des données le long d'une variété de dimension inférieure.

Chapter 11

K-Local Hyperplane and Convex Distance Nearest Neighbor Algorithms

Guided by an initial idea of building a complex (non linear) decision surface with maximal *local margin* in input space, we give a possible geometrical intuition as to why K-Nearest Neighbor (KNN) algorithms often perform more poorly than SVMs on classification tasks. We then propose modified K-Nearest Neighbor algorithms to overcome the perceived problem. The approach is similar in spirit to *Tangent Distance*, but with invariances inferred from the local neighborhood rather than prior knowledge. Experimental results on real world classification tasks suggest that the modified KNN algorithms often give a dramatic improvement over standard KNN and perform as well or better than SVMs.

11.1 Motivation

The notion of *margin* for classification tasks has been largely popularized by the success of the Support Vector Machine (SVM) [11, 102] approach. The *margin* of SVMs has a nice geometric interpretation[1]: it can be defined informally as (twice) the smallest Euclidean distance between the decision surface and the closest training point. The decision surface produced by the original SVM algorithm is the hyperplane that maximizes this distance while still correctly separating the two classes. While the notion of keeping the largest possible safety *margin* between the decision surface and the data points seems very reasonable and intuitively appealing, questions arise when extending the approach to building more complex, non-linear decision surfaces.

Non-linear SVMs usually use the "kernel trick" to achieve their non-linearity. This conceptually corresponds to first mapping the input into a higher-dimensional feature space with some non-linear transformation and building a maximum-margin hyperplane (a linear decision surface) there. The "trick" is that this mapping is never computed directly, but implicitly induced by a kernel. In this setting, the margin being maximized is still the smallest Euclidean distance between the decision surface and the training points, but this time measured in some strange, sometimes infinite dimensional, kernel-induced feature space rather than the original input space. It is less clear whether maximizing the margin in this new space, is meaningful in general (see [110]).

[1] for the purpose of this discussion, we consider the original hard-margin SVM algorithm for two linearly separable classes.

A different approach is to try and build a non-linear decision surface with maximal distance to the closest data point as measured directly in input space (as proposed in [100]). We could for instance restrict ourselves to a certain class of decision functions and try to find the function with maximal margin among this class. But let us take this even further. Extending the idea of building a correctly separating non-linear decision surface as far away as possible from the data points, we define the notion of *local margin* as the Euclidean distance, in input space, between a given point on the decision surface and the closest training point. Now would it be possible to find an algorithm that could produce a decision surface which correctly separates the classes and such that the *local margin* is everywhere maximal along its surface? Surprisingly, the plain old Nearest Neighbor algorithm (1NN) [24] does precisely this!

So why does 1NN in practice often perform worse than SVMs? One typical explanation, is that it has too much capacity, compared to SVM, that the class of function it can produce is too rich. But, considering it has *infinite* capacity (VC-dimension), 1NN is still performing quite well... This study is an attempt to better understand what is happening, based on geometrical intuition, and to derive an improved Nearest Neighbor algorithm from this understanding.

11.2 Fixing a broken Nearest Neighbor algorithm

11.2.1 Setting and definitions

The setting is that of a classical classification problem in $\mathbb{R}^n$ (the *input space*).

We are given a *training set* $\mathcal{S}$ of l points $\{x_1, \ldots, x_l\},\ x_i \in \mathbb{R}^n$ and their corresponding class label $\{y_1 = y(x_1), \ldots, y_l = y(x_l)\},\ y_i \in \mathcal{C},\ \mathcal{C} = \{1, \ldots, N_c\}$ where N_c is the number of different classes. The (x, y) pairs are assumed to be samples drawn from an unknown distribution $P(X, Y)$. Barring duplicate inputs, the class labels associated to each $x \in \mathcal{S}$ define a partition of $\mathcal{S}$: let $\mathcal{S}_c = \{x \in \mathcal{S} \mid y(x) = c\}$.

The problem is to find a *decision function* $\tilde{f} : \mathbb{R}^n \to \mathcal{C}$ that will generalize well on new points drawn from $P(X, Y)$. $\tilde{f}$ should ideally minimize the *expected classification error*, i.e. minimize $E_P[I_{\tilde{f}(X) \neq Y}]$ where E_P denotes the expectation with respect to $P(X, Y)$ and $I_{\tilde{f}(x) \neq y}$ denotes the indicator function, whose value is 1 if $\tilde{f}(x) \neq y$ and 0 otherwise.

In the previous and following discussion, we often refer to the concept of *decision surface*, also known as *decision boundary*. The function $\tilde{f}$ corresponding to a given algorithm defines for any class $c \in \mathcal{C}$ two regions of the input space: the region $R_c = \{x \in \mathbb{R}^n \mid \tilde{f}(x) = c\}$ and its complement $\mathbb{R}^n - R_c$. The *decision surface* for class c is the "boundary" between those two regions, i.e. the contour of R_c, and can be seen as a $n - 1$ dimensional manifold (a "surface" in $\mathbb{R}^n$) possibly made of several disconnected components. For simplicity, when we mention *the decision surface* in our discussion we consider only the case of two class discrimination, in which there is a single decision surface.

When we mention a *test point*, we mean a point $x \in \mathbb{R}^n$ that does not belong to the training set $\mathcal{S}$ and for which the algorithm is to decide on a class $\tilde{f}(x)$.

By *distance*, we mean the usual Euclidean distance in input-space $\mathbb{R}^n$. The distance between two points a and b will be written $d(a, b)$ or alternatively $\|a - b\|$.

The distance between a single point x and a set of points S is the distance to the closest point of the set: $d(x, S) = \min_{p \in S} d(x, p)$.

The *K-neighborhood* $\mathcal{V}^K(x)$ of a test point x is the set of the K points of $\mathcal{S}$ whose distance to x is smallest.

The *K-c-neighborhood* $\mathcal{V}_c^K(x)$ of a test point x is the set of K points of $\mathcal{S}_c$ whose distance to x is smallest.

By *Nearest Neighbor* algorithm (1NN) we mean the following algorithm: the class of a test point x is decided to be the same as the class of its closest neighbor in S.

By *K-Nearest Neighbor* algorithm (KNN) we mean the following algorithm: the class of a test point x is decided to be the same as the class appearing most frequently among the K-neighborhood of x.

11.2.2 The intuition

Figure 11.1 illustrates a possible intuition about why SVMs outperforms 1NNs when we have a finite number of samples. For classification tasks where the classes are considered to be mostly separable,[2] we often like to think of each class as residing close to a lower-dimensional manifold (in the high dimensional input space) which can reasonably be considered locally linear[3]. In the case of a finite number of samples, "missing" samples would appear as "holes" introducing artifacts in the decision surface produced by classical Nearest Neighbor algorithms.

[2] By "mostly separable" we mean that the Bayes error is almost zero, and the optimal decision surface has not too many disconnected components.

[3] i.e. each class has a probability density with a "support" that is a lower-dimensional manifold, and with the probability quickly fading, away from this support.

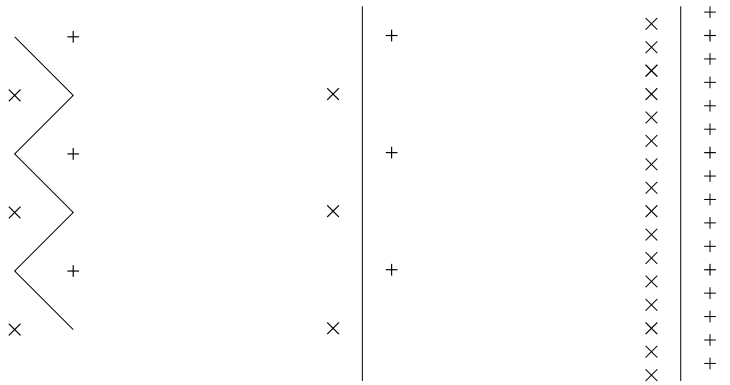

Figure 11.1: *A local view of the decision surface produced by the Nearest Neighbor (left) and SVM (center) algorithms, and how the Nearest Neighbor solution gets closer to the SVM solution in the limit, if the support for the density of each class is a manifold which can be considered locally linear (right).*

Thus the decision surface, while having the largest possible *local margin* with regard to the training points, is likely to have a poor small *local margin* with respect to yet unseen samples falling close to the locally linear manifold, and will thus result in poor generalization performance. This problem fundamentally remains with the K Nearest Neighbor (KNN) variant of the algorithm, but, as can be seen on the figure, it does not seem to affect the decision surface produced by SVMs (as the surface is constrained to a particular smooth form, a straight line or hyperplane in the case of linear SVMs). It is interesting to notice, if the assumption of locally linear class manifolds holds, how the 1NN solution approaches the SVM solution in the limit as we increase the number of samples.

To fix this problem, the idea is to somehow *fantasize* the missing points, based on a local linear approximation of the manifold of each class. This leads to modified Nearest Neighbor algorithms described in the next sections.[4]

11.2.3 The basic algorithm

Given a test point x, we are really interested in finding the closest neighbor, not among the training set $\mathcal{S}$, but among an abstract, virtually enriched training set that would contain all the *fantasized* "missing" points of the manifold of each class, locally approximated by an affine subspace. We shall thus consider, for each class c, the local affine subspace that passes through the K points of the K-c neighborhood of x. This affine subspace is typically $K-1$ dimensional or less, and we will somewhat abusively call it the "local hyperplane".[5]

Formally, the local hyperplane can be defined as

$$LH_c^K(x) = \left\{ p \;\middle|\; p = \sum_{k=1}^{K} \alpha_k N_k,\ \alpha \in \mathbb{R}^K,\ \sum_{k=1}^{K} \alpha_k = 1 \right\} \tag{11.1}$$

where $\{N_1, \ldots, N_k\} = \mathcal{V}_c^K(x)$.

Another way to define this hyperplane, that gets rid of the constraint $\sum \alpha_k = 1$, is to take a reference point within the hyperplane as an origin, for instance the

[4] Note that although we never generate the "fantasy" points explicitly, the proposed algorithms are really equivalent to classical 1NN with fantasized points.

[5] Strictly speaking a hyperplane in an n dimensional input space is an $n-1$ affine subspace, while our "local hyperplanes" can have fewer dimensions.

centroid[6] $\bar{N} = \frac{1}{K}\sum_{k=1}^{K} N_k$. This same hyperplane can then be expressed as

$$LH_c^K(x) = \left\{ p \;\middle|\; p = \bar{N} + \sum_{k=1}^{K} \alpha_k \overrightarrow{V}_k,\ \alpha \in \mathbb{R}^K \right\} \tag{11.2}$$

where $\overrightarrow{V}_k = N_k - \bar{N}$.

Our *modified nearest neighbor algorithm* then associates a test point x to the class c whose hyperplane $LH_c^K(x)$ is closest to x. Formally $\tilde{f}(x) = \arg\min_{c \in \mathcal{C}} d(x, LH_c^K(x))$, where $d(x, LH_c^K(x))$ is logically called *K-local Hyperplane Distance*, hence the name *K-local Hyperplane Distance Nearest Neighbor* algorithm (HKNN in short).

Computing, for each class c

$$\begin{aligned} d(x, LH_c^K(x)) &= \min_{p \in LH_c^K(x)} \|x - p\| \\ &= \min_{\alpha \in \mathbb{R}^K} \left\| x - \bar{N} - \sum_{k=1}^{K} \alpha_k \overrightarrow{V}_k \right\| \end{aligned} \tag{11.3}$$

amounts to solving a linear system in α, that can be easily expressed in matrix form as:

$$(V' \cdot V) \cdot \alpha = V' \cdot (x - \bar{N}) \tag{11.4}$$

where x and $\bar{N}$ are n dimensional column vectors, $\alpha = (\alpha_1, \ldots, \alpha_K)'$, and V is a $n \times K$ matrix whose columns are the $\overrightarrow{V}_k$ vectors defined earlier.[7]

[6] We could be using one of the K neighbors as the reference point, but this formulation with the centroid will prove useful later.

[7] Actually there is an infinite number of solutions to this system since the $\overrightarrow{V}_k$ are linearly dependent: remember that the initial formulation had an equality constraint and thus only $K - 1$ effective degrees of freedom. But we are interested in $d(x, LH_c^K(x))$ not in α so any solution will do. Alternatively, we can remove one of the $\overrightarrow{V}_k$ from the system so that it has a unique solution.

11.2.4 Links with other paradigms

The proposed HKNN algorithm is very similar in spirit to the *Tangent Distance Algorithm* [87]. $LH_c^K(x)$ can be seen as a tangent hyperplane representing a set of local directions of transformation (any linear combination of the $\overrightarrow{V}_k$ vectors) that do not affect the class identity. These are *invariances*. The main difference is that in HKNN these invariances are inferred directly from the local neighborhood in the training set, whereas in Tangent Distance, they are based on prior knowledge. It should be interesting (and relatively easy) to combine both approaches for improved performance when prior knowledge is available.

Previous work on nearest-neighbor variations based on other locally-defined metrics can be found in [85, 66, 38, 47], and is very much related to the more general paradigm of *Local Learning Algorithms* [13, 4, 69].

We should also mention close similarities between our approach and the recently proposed *Local Linear Embedding* [78] method for dimensionality reduction.

The idea of fantasizing points around the training points in order to define the decision surface is also very close to methods based on estimating the class-conditional input density [100, 19].

Besides, it is interesting to look at HKNN from a different, less geometrical angle: it can be understood as choosing the class that achieves the best reconstruction (the smallest reconstruction error) of the test pattern through a linear combination of K particular prototypes of that class (the K neighbors). From this point of view, the algorithm is very similar to the *Nearest Feature Line* (NFL) [60] method. They differ in the fact that NFL considers all pairs of points for its search rather than

the local K neighbors, thus looking at many (l^2) lines (i.e. 2 dimensional affine subspaces), rather than at a single $K-1$ dimensional one.

11.3 Fixing the basic HKNN algorithm

11.3.1 Problem arising for large K

One problem with the basic HKNN algorithm, as previously described, arises as we increase the value of K, i.e. the number of points considered in the neighborhood of the test point. In a typical high dimensional setting, *exact* colinearities between input patterns are rare, which means that as soon as $K > n$, any pattern of $\mathbb{R}^n$ (including nonsensical ones) can be produced by a linear combination of the K neighbors. The "actual" dimensionality of the manifold may be much less than K. This is due to "near-colinearities" producing directions associated to small eigenvalues of the covariance matrix $V' \cdot V$ that are but noise, that can lead the algorithm to mistake those noise directions for "invariances", and may hurt its performance even for smaller values of K. Another related issue is that the linear approximation of the class manifold by a hyperplane is valid only locally, so we might want to restrict the "fantasizing" of class members to a smaller region of the hyperplane. We considered two ways of dealing with these problems.[8]

[8] A third interesting avenue, which we did not have time to explore, would be to keep only the most relevant principal components of V, ignoring those corresponding to small eigenvalues.

11.3.2 The convex hull solution

One way to avoid the above mentioned problems is to restrict ourselves to considering the *convex hull* of the neighbors, rather than the whole hyperplane they support (of which the convex hull is a subset). This corresponds to adding a constraint of $\alpha_k \geq 0, \forall k$ to equation (11.1). Unlike the problem of computing the distance to the hyperplane, the distance to the convex hull cannot be found by solving a simple linear system, but typically requires solving a quadratic programming problem (very similar to the one of SVMs). While this is more complex to implement, it should be mentioned that the problems to be solved are of a relatively small dimension of order K, and that the time of the whole algorithm will very likely still be dominated by the search of the K nearest neighbors within each class. This algorithm will be referred to as *K-local Convex Distance Nearest Neighbor Algorithm* (CKNN in short).

11.3.3 The "weight decay" penalty solution

This consists in incorporating a penalty term to equation (11.3) to penalize large values of α (i.e. it penalizes moving away from the centroid, especially in non essential directions):

$$d'(x, LH_c^K(x))^2 = \min_{\alpha \in \mathbf{R}^K} \left\| x - \bar{N} - \sum_{k=1}^{K} \alpha_k \overrightarrow{V}_k \right\|^2 + \lambda \sum_{k=1}^{K} \alpha_k^2 \qquad (11.5)$$

The solution for α is given by solving the linear system $(V' \cdot V + \lambda I_n) \cdot \alpha = V' \cdot (x - \bar{N})$ where I_n is the $n \times n$ identity matrix. This is equation (11.4) with an additional diagonal term. The resulting algorithm is a generalization of HKNN (basic HKNN corresponds to $\lambda = 0$).

11.4 Experimental results

We performed a number of experiments, to highlight different properties of the algorithms:

- A first 2D toy example (see Figure 11.2) graphically illustrates the qualitative differences in the decision surfaces produced by KNN, linear SVM and CKNN.
- Table 11.1 gives quantitative results on two real-world digit OCR tasks, allowing to compare the performance of the different old and new algorithms.
- Figure 11.3 illustrates the problem arising with large K, mentioned in Section 11.3, and shows that the two proposed solutions: CKNN and HKNN with an added weight decay λ, allow to overcome it.
- In our final experiment, we wanted to see if the good performance of the new algorithms absolutely depended on having all the training points at hand, as this has a direct impact on speed. So we checked what performance we could get out of HKNN and CKNN when using only a small but representative subset of the training points, namely the set of support vectors found by a Gaussian Kernel SVM. The results obtained for MNIST are given in Table 11.2, and look very encouraging. HKNN appears to be able to perform as well or better than SVMs *without* requiring more data points than SVMs.

Table 11.1: *Test-error obtained on the USPS and MNIST digit classifi cation tasks by KNN, SVM (using a Gaussian Kernel), HKNN and CKNN. Hyper parameters were tuned on a separate validation set. Both HKNN and CKNN appear to perform much better than original KNN, and even compare favorably to SVMs.*

Data Set	Algorithm	Test Error	Parameters used
USPS	KNN	4.98%	$K = 1$
(6291 train,	SVM	4.33%	$\sigma = 8, C = 100$
1000 valid.,	HKNN	3.93%	$K = 15, \lambda = 30$
2007 test points)	CKNN	3.98%	$K = 20$
MNIST	KNN	2.95%	$K = 3$
(50000 train,	SVM	1.30%	$\sigma = 6.47, C = 100$
10000 valid.,	HKNN	1.26%	$K = 65, \lambda = 10$
10000 test points)	CKNN	1.46%	$K = 70$

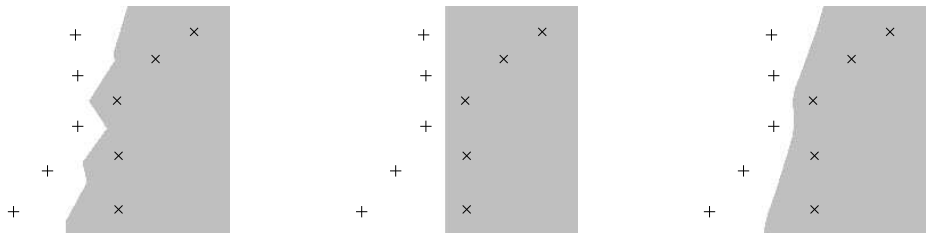

Figure 11.2: *2D illustration of the decision surfaces produced by KNN (left, K=1), linear SVM (middle), and CKNN (right, K=2). The "holes" are again visible in KNN. CKNN doesn't suffer from this, but keeps the objective of* **maximizing the margin locally**.

11.5 Conclusion

From a few geometrical intuitions, we have derived two modified versions of the KNN algorithm that look very promising. HKNN is especially attractive: it is very simple to implement on top of a KNN system, as it only requires the additional step of solving a small and simple linear system, and appears to greatly boost the performance of standard KNN even above the level of SVMs. The proposed algorithms share the advantages of KNN (no training required, ideal for fast adaptation, natural handling of the multi-class case) and its drawbacks (requires large memory, slow testing). However our latest result also indicate the possibility of substantially reducing the reference set in memory without loosing on accuracy. This suggests that the algorithm indeed captures essential information in the data, and that our initial intuition on the nature of the flaw of KNN may well be at least partially correct.

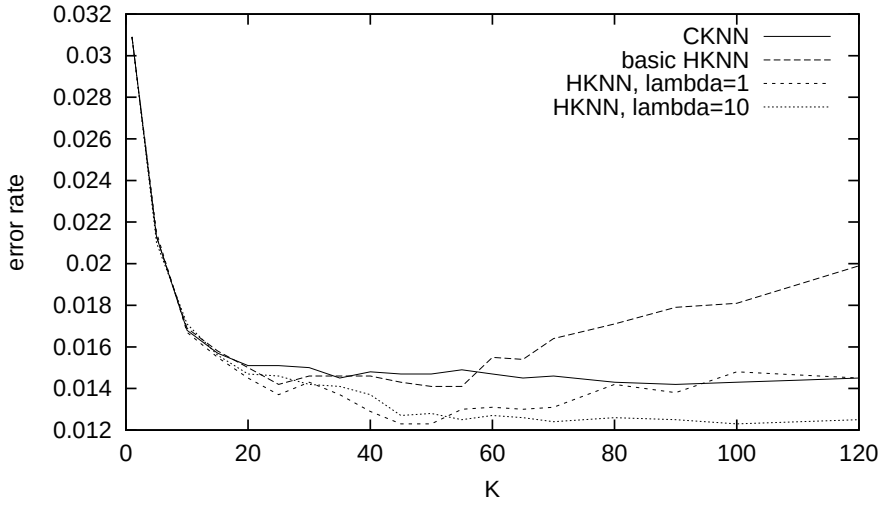

Figure 11.3: *Error rate on MNIST as a function of K for CKNN, and HKNN with different values of λ. As can be seen the basic HKNN algorithm performs poorly for large values of K. As expected, CKNN is relatively unaffected by this problem, and HKNN can be made robust through the added "weight decay" penalty controlled by λ.*

Table 11.2: *Test-error obtained on MNIST with HKNN and CKNN when using a reduced training set made of the 16712 support vectors retained by the best Gaussian Kernel SVM. This corresponds to 28% of the initial 60000 training patterns. Performance is even better than when using the whole dataset. But here, hyper parameters K and λ were chosen with the test set, as we did not have a separate validation set in this setting. It is nevertheless remarkable that comparable performances can be achieved with far fewer points.*

Data Set	Algorithm	Test Error	Parameters used
MNIST (16712 train s.v.,	HKNN	1.23%	$K = 60, \lambda = 10$
10000 test points)	CKNN	1.36%	$K = 45$

Chapitre 12

Présentation du troisième article

P. Vincent et Y. Bengio. **Manifold Parzen Windows.** Publié en 2003 dans *Advances in Neural Information Processing Systems 15*, aux éditions MIT Press.

12.1 Contexte et objectifs de cette recherche

Notre recherche sur *K-Local Hyperplane and Convex Distance Nearest Neighbor Algorithms* ayant prouvé qu'il était possible de grandement améliorer l'algorithme non paramétrique de classification KNN, il était naturel de tenter d'appliquer la même intuition géométrique à son pendant pour l'estimation de densité : les fenêtres de Parzen. C'est ce que nous réalisons dans cet article, en utilisant des Gaussiennes aplaties orientées selon les directions principales apparaissant dans les données du voisinage de chaque point d'entrainement.

L'idée que, en haute dimension, le support des données pouvait être une variété de dimension inférieure, a été largement popularisée par la parution de deux algorithmes de réduction de dimensionalité : *Local Linear Embedding* [78] et *Iso-Map* [95]. Nos recherches également peuvent être comprises comme la prise en compte cette belle intuition géométrique pour "réparer" des algorithmes non paramétriques classiques (KNN et Parzen). Cela dit, l'orientation de nos recherches n'est pas née au départ de considérations aussi claires concernant les variétés. Leur point de départ a été une simple tentative de supprimer l'artefact de *zig-zag* de la surface de décision produite par KNN. La notion de variété de dimension inférieure ne nous est apparue évidente que par la suite, clarifiant grandement notre intuition initiale.

12.2 Remarque sur le choix de la spirale

L'exemple de la spirale, utilisé dans l'article, a souvent été critiqué comme étant très artificiel, et ne reflétant en rien une situation que l'on pourrait retrouver dans des problèmes concrets réels. Mais il s'agit d'un très bel exemple de variété de dimension 1 dans un espace de dimension 2. Nous espérons que la discussion sur les variétés de plus faible dimension de la section 4.3 aura pu vous convaincre de son utilité, et permis de comprendre en quoi il reflète une caractéristique susceptible de jouer un rôle important dans les problèmes en haute dimension.

Par ailleurs, notez que nos algorithmes ne supposent jamais que les données résident réellement exactement sur une variété de dimension fixe donnée. Ils suffit en principe que la distribution des données soit localement plus concentrée dans

certaines directions que dans d'autres pour que les bénéfices de ces algorithmes se fassent ressentir.

12.3 Contribution au domaine

L'algorithme de *Manifold Parzen* est une extension naturelle de l'estimateur de Parzen classique. La contribution de cet article est donc essentiellement d'avoir su régler un grand nombre de détails pratiques de sa mise en oeuvre, notamment concernant la modélisation et représentation efficace de Gaussiennes aplaties en haute dimension, à partir d'un voisinage. Nous tenons également à signaler que cela constitue une prouesse technique d'avoir réussi à appliquer un algorithme aussi gourmand en mémoire et en temps de calcul à des problèmes concrets en haute dimension de cette taille. Ces expériences n'auraient sans doute pas été possibles sur le matériel d'il y a quelques années.

Aussi la grande contribution de cet article est avant tout de montrer que l'algorithme proposé est viable et donne d'excellent résultats. Là encore, la prise en compte des directions principales locales en haute dimension paraît améliorer grandement les résultats par rapport à l'estimateur classique. Il est notamment remarquable d'avoir réussi, avec un algorithme générique d'estimation de densité, à battre les SVMs, l'algorithme de classification représentant l'état de l'art en matière d'entraînement discriminant en haute dimension.

Chapter 13

Manifold Parzen Windows

The similarity between objects is a fundamental element of many learning algorithms. Most non-parametric methods take this similarity to be fixed, but much recent work has shown the advantages of learning it, in particular to exploit the local invariances in the data or to capture the possibly non-linear manifold on which most of the data lies. We propose a new non-parametric kernel density estimation method which captures the local structure of an underlying manifold through the leading eigenvectors of regularized local covariance matrices. Experiments in density estimation show significant improvements with respect to Parzen density estimators. The density estimators can also be used within Bayes classifiers, yielding classification rates similar to SVMs and much superior to the Parzen classifier.

13.1 Introduction

In [105], while attempting to better understand and bridge the gap between the good performance of the popular Support Vector Machines and the more traditional K-NN (K Nearest Neighbors) for classification problems, we had suggested a modified Nearest-Neighbor algorithm. This algorithm, which was able to slightly outperform SVMs on several real-world problems, was based on the geometric intuition that the classes actually lived "close to" a lower dimensional non-linear manifold in the high dimensional input space. When this was not properly taken into account, as with traditional K-NN, the sparsity of the data points due to having a finite number of training samples would cause "holes" or "zig-zag" artifacts in the resulting decision surface, as illustrated in Figure 13.1.

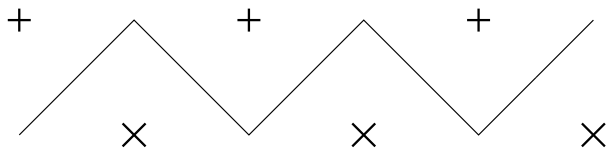

Figure 13.1: A local view of the decision surface, with "holes", produced by the Nearest Neighbor when the data have a local structure (horizontal direction).

The present work is based on the same underlying geometric intuition, but applied to the well known Parzen windows [71] non-parametric method for density estimation, using Gaussian kernels.

Most of the time, Parzen Windows estimates are built using a "spherical Gaussian" with a single scalar variance (or *width*) parameter σ^2. It is also possible to use a "diagonal Gaussian", i.e. with a diagonal covariance matrix, or even a "full Gaussian" with a full covariance matrix, usually set to be proportional to the

global empirical covariance of the training data. However these are equivalent to using a spherical Gaussian on preprocessed, *normalized* data (i.e. normalized by subtracting the empirical sample mean, and multiplying by the inverse sample covariance). Whatever the shape of the kernel, if, as is customary, a *fixed shape* is used, merely centered on every training point, the shape can only compensate for the *global structure* (such as global covariance) of the data.

Now if the true density that we want to model is indeed "close to" a non-linear lower dimensional manifold embedded in the higher dimensional input space, in the sense that most of the probability density is concentrated around such a manifold (with a small noise component away from it), then using Parzen Windows with a spherical or fixed-shape Gaussian is probably not the most appropriate method, for the following reason.

While the true density mass, in the vicinity of a particular training point x_i, will be mostly concentrated in a few local directions along the manifold, a spherical Gaussian centered on that point will spread its density mass equally along all input space directions, thus giving too much probability to irrelevant regions of space and too little along the manifold. This is likely to result in an excessive "bumpyness" of the thus modeled density, much like the "holes" and "zig-zag" artifacts observed in KNN (see Fig. 13.1 and Fig. 13.2).

If the true density in the vicinity of x_i is concentrated along a lower dimensional manifold, then it should be possible to infer the local direction of that manifold from the neighborhood of x_i, and then anchor on x_i a Gaussian "pancake" parameterized in such a way that it spreads mostly along the directions of the manifold, and is almost flat along the other directions. The resulting model is a mixture

of Gaussian "pancakes", similar to [48], mixtures of probabilistic PCAs [99] or mixtures of factor analyzers [44, 43], in the same way that the most traditional Parzen Windows is a mixture of spherical Gaussians. But it remains a memory-based method, with a Gaussian kernel centered on each training points, yet with a *differently shaped* kernel for each point.

13.2 The Manifold Parzen Windows algorithm

In the following we formally define and justify in detail the proposed algorithm. Let $\mathcal{X}$ be an n-dimensional random variable with values in $\mathbb{R}^n$, and an *unknown probability density function* $p_{\mathcal{X}}(.)$. Our *training set* contains l samples of that random variable, collected in a $l \times n$ matrix X whose row x_i is the i-th sample. Our goal is to estimate the density $p_{\mathcal{X}}$. Our estimator $\hat{p}_{mp}(.)$ has the form of a mixture of Gaussians, but unlike the Parzen density estimator, its covariances C_i are not necessarily spherical and not necessarily identical everywhere:

$$\hat{p}_{mp}(x) = \frac{1}{l} \sum_{i=1}^{l} \mathcal{N}_{x_i, C_i}(x), \tag{13.1}$$

where $\mathcal{N}_{\mu,C}(x)$ is the multivariate Gaussian density with mean vector μ and covariance matrix C:

$$\mathcal{N}_{\mu,C}(x) = \frac{1}{\sqrt{(2\pi)^n |C|}} e^{-\frac{1}{2}(x-\mu)' C^{-1} (x-\mu)} \tag{13.2}$$

where $|C|$ is the determinant of C. How should we select the individual covariances C_i? From the above discussion, we expect that if there is an underlying "non-linear principal manifold", those gaussians would be "pancakes" aligned with the plane locally tangent to this underlying manifold. The only available

information (in the absence of further prior knowledge) about this tangent plane can be gathered from the training samples int the neighborhood of x_i. In other words, we are interested in computing the *principal directions* of the samples in the neighborhood of x_i.

For generality, we can define a *soft neighborhood* of x_i with a neighborhood kernel $\mathcal{K}(x; x_i)$ that will associate an influence weight to any point x in the neighborhood of x_i. We can then compute the weighted covariance matrix

$$C_{\mathcal{K}_i} = \frac{\sum_{j=1..l, j \neq i} \mathcal{K}(x_j; x_i)(x_j - x_i)(x_j - x_i)'}{\sum_{j=1..l, j \neq i} \mathcal{K}(x_j; x_i)} \tag{13.3}$$

where $(x_j - x_i)'(x_j - x_i)$ denotes the outer product.

$\mathcal{K}(x, x_i)$ could be a spherical Gaussian centered on x_i for instance, or any other positive definite kernel, possibly incorporating prior knowledge as to what constitutes a reasonable neighborhood for point x_i. Notice that if $\mathcal{K}(x, x_i)$ is a constant (uniform kernel), $C_{\mathcal{K}_i}$ is the global training sample covariance. As an important special case, we can define a *hard k-neighborhood* for training sample x_i by assigning a weight of 1 to any point no further than the k-th nearest neighbor of x_i among the training set, according to some metric such as the Euclidean distance in input space, and assigning a weight of 0 to points further than the k-th neighbor. In that case, $C_{\mathcal{K}_i}$ is the unweighted covariance of the k nearest neighbors of x_i.

Notice what is happening here: we start with a possibly rough prior notion of neighborhood, such as one based on the ordinary Euclidean distance in input space, and use this to compute a local covariance matrix, which implicitly defines a refined local notion of neighborhood, taking into account the local direction observed in the training samples.

Now that we have a way of computing a local covariance matrix for each training point, we might be tempted to use this directly in equations 13.2 and 13.1. But a number of problems must first be addressed:

- Equation 13.2 requires the **inverse** covariance matrix, whereas $C_{\mathcal{K}_i}$ is likely to be ill-conditioned. This situation will definitely arise if we use a *hard k-neighborhood* with $k < n$. In this case we get a Gaussian that is totally flat outside of the affine subspace spanned by x_i and its k neighbors, and it does not constitute a proper density in $\mathbb{R}^n$. A common way to deal with this problem is to add a small isotropic (spherical) Gaussian noise of variance σ^2 in all directions, which is done by simply adding σ^2 to the diagonal of the covariance matrix: $C_i = C_{\mathcal{K}_i} + \sigma^2 I$.

- Even if we regularize C_i by adding σ^2, when we deal with high dimensional spaces, it would be prohibitive in computation time and storage to keep and use the full inverse covariance matrix as expressed in 13.2. This would in effect multiply both the time and storage requirement of the already expensive ordinary Parzen Windows by $n+1$. So instead, we use a different, more compact representation of the inverse Gaussian, by storing only the eigenvectors associated with the first few largest eigenvalues of C_i, as described below.

The eigen-decomposition of a covariance matrix C can be expressed as: $C = VDV'$, where the columns of V are the orthonormal eigenvectors and D is a diagonal matrix with the eigenvalues $\lambda_1 \ldots \lambda_n$, that we will suppose sorted in decreasing order, without loss of generality.

The first d eigenvectors with largest eigenvalues correspond to the *principal directions* of the local neighborhood, i.e. the high variance local directions of the supposed underlying d-dimensional manifold (but the true underlying dimension is unknown and may actually vary across space). The last few eigenvalues and eigenvectors are but noise directions with a small variance. So we may, without too much risk, force those last few components to the same low noise level σ^2. We have done this by zeroing the last $n-d$ eigenvalues (by considering only the first d leading eigenvalues) and then adding σ^2 to all eigenvalues. This allows us to store only the first d eigenvectors, and to later compute $\mathcal{N}_{\mu,C}(x)$ in time $\mathcal{O}(n \cdot d)$ instead of $\mathcal{O}(n^2)$. Thus both the storage requirement and the computational cost when estimating the density at a test point is only about $d+1$ times that of ordinary Parzen. It can easily be shown that such an approximation of the covariance matrix yields to the following computation of $\mathcal{N}_{\mu,C}(x)$:

Algorithm LocalGaussian($x, x_i, V_i, \lambda_i, d, \sigma^2$**)**

Input: test vector $x \in \mathbb{R}^n$, training vector $x_i \in \mathbb{R}^n$, d eigenvalues λ_{ij}, d eigenvectors in the columns of V_i, dimension d, and the regularization hyper-parameter σ^2.

(1) $r = d\log(2\pi) + \sum_{j=1}^{d}\log(\lambda_j + \sigma^2) + (n-d)\log(\sigma^2)$

(2) $q = \frac{1}{\sigma^2}||x - x_i||^2 + \sum_{j=1}^{d}(\frac{1}{\lambda_j} - \frac{1}{\sigma^2})||V_j'(x - x_i)||^2$

Output: Gaussian density $e^{-0.5(r+q)}$

In the case of the *hard k-neighborhood*, the training algorithm pre-computes the local principal directions $V_i$of the k nearest neighbors of each training point i (in practice we compute them with a SVD rather than an eigen-decomposition of

the covariance matrix, see below). Note that with $d = 0$, we trivially obtain the traditional Parzen windows estimator.

Algorithm MParzen::Train(X, d, k, σ^2)

Input: training set matrix X with l rows $x_i \in \mathbb{R}^n$, chosen number of principal directions d, chosen number of neighbors $k \geq d$, and regularization hyper-parameter σ^2.

(1) For $i \in \{1, 2, \ldots, l\}$

(2) Collect k nearest neighbors x_j of x_i, and put $x_j - x_i$ in the rows of matrix M.

(3) Perform a partial singular value decomposition of M, to obtain the leading d singular values s_j ($j \in \{1, \ldots, d\}$) and singular column vectors $V_{i \cdot j}$ of M.

(4) For $j \in \{1, \ldots, d\}$, let $\lambda_{ij} = \sigma^2 + \frac{s_j^2}{l}$

Output: The model $\mathcal{M}$ = $(X, V, \lambda, k, d, \sigma^2)$, where V is an $l \times n \times d$ tensor that collects all the eigenvectors and λ is a $l \times d$ matrix with all the eigenvalues.

Algorithm MParzen::Test($x, \mathcal{M}$)

Input: test point x and model $\mathcal{M} = (X, V, \lambda, k, d, \sigma^2)$.

(1) $s \leftarrow 0$

(2) For $i \in \{1, 2, \ldots, l\}$

(3) $s \leftarrow s+$ LocalGaussian$(x, x_i, V_i, \lambda_i, d, \sigma^2)$

Output: manifold Parzen estimator $\hat{p}_{mp}(x) = \frac{s}{l}$.

13.3 Related work

As we have already pointed out, Manifold Parzen Windows, like traditional Parzen Windows and so many other density estimation algorithms, results in defining the density as a mixture of Gaussians. What differs is mostly *how* those Gaussians and their parameters are chosen. The idea of having a parameterization of each Gaussian that orients it along the local principal directions also underlies the already mentioned work on mixtures of Gaussian pancakes [48], mixtures of probabilistic PCAs [99], and mixtures of factor analysers [44, 43]. All these algorithms typically model the density using a relatively small number of Gaussians, whose centers and parameters must be learnt with some iterative optimisation algorithm such as EM (procedures which are known to be sensitive to local minima traps). By contrast our approach is, like the original Parzen windows, heavily memory-based. It avoids the problem of optimizing the centers by assigning a Gaussian to every training point, and uses simple analytic SVD to compute the local principal directions for each.

Another successful memory-based approach that uses local directions and inspired our work is the tangent distance algorithm [87]. While this approach was initially aimed at solving classification tasks with a nearest neighbor paradigm, some work has already been done in developing it into a probabilistic interpretation for mixtures with a few gaussians, as well as for full-fledged kernel density estimation [53, 26]. The main difference between our approach and the above is that the Manifold Parzen estimator does not require prior knowledge, as it infers the local directions directly from the data, although it should be easy to also incorporate prior knowledge if available.

We should also mention similarities between our approach and the *Local Linear Embedding* and recent related dimensionality reduction methods [78, 94, 28, 14]. There are also links with previous work on locally-defined *metrics* for nearest-neighbors [85, 66, 38, 47]. Lastly, it can also be seen as an extension along the line of traditional variable and adaptive kernel estimators that adapt the kernel width locally (see [50] for a survey).

13.4 Experimental results

Throughout this whole section, when we mention Parzen Windows (sometimes abbreviated *Parzen*), we mean ordinary Parzen windows using a spherical Gaussian kernel with a single hyper-parameter σ, the width of the Gaussian.

When we mention Manifold Parzen Windows (sometimes abbreviated *MParzen*), we used a *hard k-neighborhood*, so that the hyper-parameters are: the number of neighbors k, the number of retained principal components d, and the additional isotropic Gaussian noise parameter σ.

When measuring the quality of a density estimator $\hat{p}(x)$, we used the average negative log likelihood: ANLL $= -\frac{1}{m}\sum_{i=1}^{m} \log \hat{p}(x_i)$ with the m examples x_i from a test set.

13.4.1 Experiment on 2D artificial data

A training set of 300 points, a validation set of 300 points and a test set of 10000 points were generated from the following distribution of two dimensional (x, y)

points:

$$x = 0.04\ t\ \sin(t) + \epsilon_x,\ y = 0.04\ t\ \cos(t) + \epsilon_y$$

where $t \sim U(3, 15)$, $\epsilon_x \sim N(0, 0.01)$, $\epsilon_y \sim N(0, 0.01)$, $U(a, b)$ is uniform in the interval (a, b) and $N(\mu, \sigma)$ is a normal density.

We trained an ordinary Parzen, as well as MParzen with $d = 1$ and $d = 2$ on the training set, tuning the hyper-parameters to achieve best performance on the validation set. Figure 13.2 shows the training set and gives a good idea of the densities produced by both kinds of algorithms (as the visual representation for MParzen with $d = 1$ and $d = 2$ did not appear very different, we show only the case $d = 1$). The graphic reveals the anticipated "bumpyness" artifacts of ordinary Parzen, and shows that MParzen is indeed able to better concentrate the probability density along the manifold, even when the training data is scarce.

Quantitative comparative results of the two models are reported in table 13.1

Table 13.1: Comparative results on the artificial data (standard errors are in parenthesis).

Algorithm	Parameters used	ANLL on test-set
Parzen	$\sigma = 0.0173$	-1.183 (0.016)
MParzen	$d = 1$, $k = 11$, $\sigma = 0.09$	-1.466 (0.009)
MParzen	$d = 2$, $k = 10$, $\sigma = 0.00001$	-1.419 (0.009)

Several points are worth noticing:

- Both *MParzen* models seem to achieve a lower ANLL than ordinary Parzen (even though the underlying manifold really has dimension $d = 1$), and with more consistency over the test sets (lower standard error).

- The optimal width σ for ordinary *Parzen* is much larger than the noise parameter of the true generating model (0.01), probably because of the finite sample size.

- The optimal regularization parameter σ for *MParzen* with $d = 1$ (i.e. supposing a one-dimensional underlying manifold) is very close to the actual noise parameter of the true generating model. This suggests that it was able to capture the underlying structure quite well. Also it is the best of the three models, which is not surprising, since the true model is indeed a one dimensional manifold with an added isotropic Gaussian noise.

- The optimal additional noise parameter σ for *MParzen* with $d = 2$ (i.e. supposing a two-dimensional underlying manifold) is close to 0, which suggests that the model was able to capture all the noise in the second "principal direction".

13.4.2 Density estimation on OCR data

In order to compare the performance of both algorithms for density estimation on a real-world problem, we estimated the density of one class of the MNIST OCR data set, namely the "2" digit. The available data for this class was divided into 5400 training points, 558 validation points and 1032 test points. Hyper-parameters were tuned on the validation set. The results are summarized in Table 13.2, using the

performance measures introduced above (average negative log-likelihood). Note that the improvement with respect to Parzen windows is extremely large and of course statistically significant.

Table 13.2: Density estimation of class '2' in the MNIST data set. Standard errors in parenthesis.

Algorithm	Parameters used	validation ANLL	test ANLL
Parzen	$\sigma = 0.19$	-197.27 (4.18)	-197.19 (3.55)
MParzen	$d = 50$, $k = 80$, $\sigma = 0.09$	-696.42 (5.94)	-695.15 (5.21)

13.4.3 Classification performance

To obtain a probabilistic classifier with a density estimator we train an estimator $\hat{p}_c(x) = \hat{p}(x|c)$ for each class c, and apply Bayes' rule to obtain $\hat{P}(c|x) = \frac{\hat{p}(x|c)\hat{P}(c)}{\sum_{c'} \hat{p}(x|c')\hat{P}(c')}$. When measuring the quality of a probabilistic classifier $\hat{P}(c|x)$, we used the negative conditional log likelihood: ANCLL $\stackrel{\text{def}}{=} -\frac{1}{m}\sum_{i=1}^{m} \log \hat{P}(c_i|x_i)$, with the m examples (c_i, x_i) (correct class, input) from a test set.

This method was applied to both the Parzen and the Manifold Parzen density estimators, which were compared with state-of-the-art Gaussian SVMs on the full USPS data set. The original training set (7291) was split into a training (first 6291) and validation set (last 1000), used to tune hyper-parameters. The classification errors for all three methods are compared in Table 13.3, where the hyper-parameters are chosen based on validation classification error. The log-likelihoods are compared in Table 13.4, where the hyper-parameters are chosen based on validation ANCLL. Hyper-parameters for SVMs are the box constraint C and the Gaussian

width σ. MParzen has the lowest classification error and ANCLL of the three algorithms.

Table 13.3: *Classification error obtained on USPS with SVM, Parzen windows and Manifold Parzen windows classifiers.*

Algorithm	validation error	test error	parameters
SVM	1.2%	4.68%	$C = 100$, $\sigma = 8$
Parzen	1.8%	5.08%	$\sigma = 0.8$
MParzen	0.9%	4.08%	$d = 11$, $k = 11$, $\sigma^2 = 0.1$

Table 13.4: *Comparative negative conditional log likelihood obtained on USPS.*

Algorithm	valid ANCLL	test ANCLL	parameters
Parzen	0.1022	0.3478	$\sigma = 0.8$
MParzen	0.0658	0.3384	$d = 17$, $k = 17$, $\sigma^2 = 0.75$

13.5 Conclusion

The rapid increase in computational power now allows to experiment with sophisticated non-parametric models such as those presented here. They have allowed to show the usefulness of learning the local structure of the data through a regularized covariance matrix estimated for each data point. By taking advantage of local structure, the new kernel density estimation method outperforms the Parzen windows estimator. Classifiers built from this density estimator yield state-of-the-art knowledge-free performance, which is remarkable for a not discriminatively

trained classifier. Besides, in some applications, the accurate estimation of probabilities can be crucial, e.g. when the classes are highly imbalanced.

Future work should consider other alternative methods of estimating the local covariance matrix, for example as suggested here using a weighted estimator, or taking advantage of prior knowledge (e.g. the Tangent distance directions).

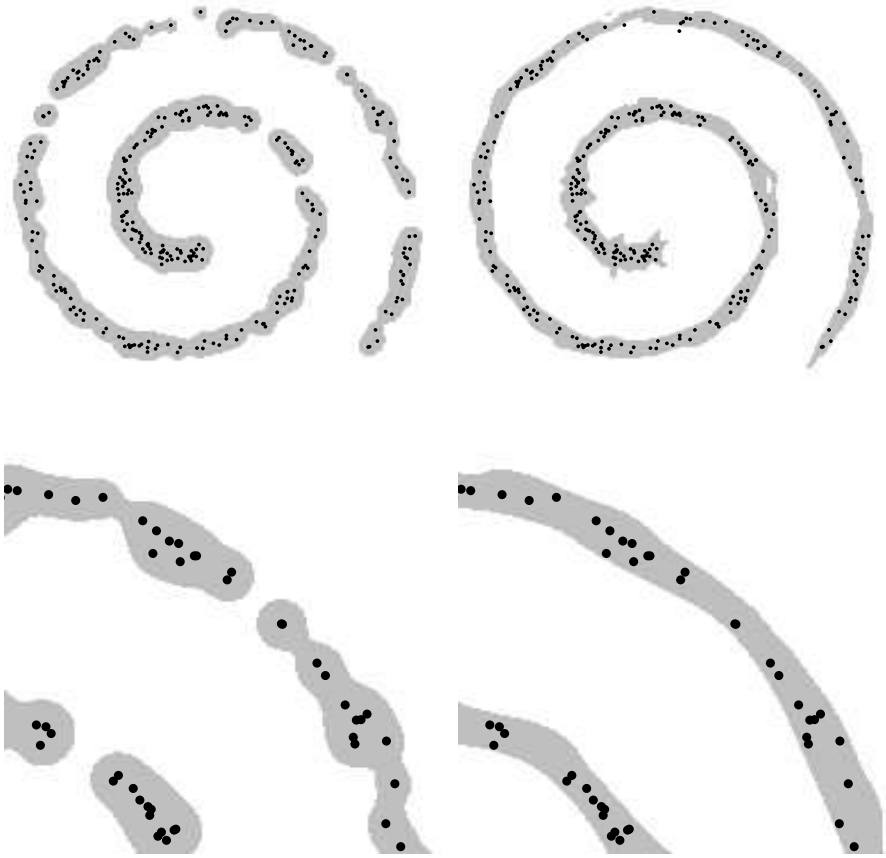

Figure 13.2: Illustration of the density estimated by ordinary Parzen Windows (left) and Manifold Parzen Windows (right). The two images on the bottom are a zoomed area of the corresponding image at the top. The 300 training points are represented as black dots and the area where the estimated density $\hat{p}_{mp}(x)$ is above 1.0 is painted in gray. The excessive "bumpyness" and holes produced by ordinary Parzen windows model can clearly be seen, whereas Manifold Parzen density is better aligned with the underlying manifold, allowing it to even successfully "extrapolate" in regions with few data points but high true density.

Quatrième partie

Synthèse

Chapitre 14

Discussion et synthèse

14.1 Synthèse des algorithmes proposés

Nous venons de présenter trois familles d'algorithmes d'apprentissage capables de performances comparables ou supérieures aux SVMs lorsqu'ils sont appliqués à la classification. Dans les trois cas, il s'agit de modèles à noyaux, au sens large, avec des noyaux centrés sur les exemples d'apprentissage.

Les trois algorithmes sont néanmoins de nature assez différente :

– *Kernel Matching Pursuit* est plus naturellement un algorithme de *régression*. En effet, l'optimisation d'un coût quadratique avec cette famille d'algorithmes est le plus naturel et efficace. Nous avons bien sûr proposé des variantes permettant d'optimiser des fonctions de coût arbitraires, dont certaines fonctions de coût de marge théoriquement bien appropriée pour la classification, mais il n'en reste pas moins que l'algorithme est initialement conçu pour la régression, même si

cela ne nous a pas empêcher d'obtenir de très bons résultats sur des problèmes de classification (notre but étant de battre les résultats d'un classifieur SVM).

– Les algorithmes de *K-Local Hyperplane and Convex Distance Nearest Neighbor* sont véritablement conçus pour la *classification.*
– Quant à *Manifold Parzen Windows*, il s'agit évidemment d'*estimation de densité*.

Les objectifs visés par la recherche et les techniques employées diffèrent également :

– Pour *Kernel Matching Pursuit* on cherchait à contrôler précisément le nombre de points de support, tout en construisant une solution ayant exactement la même forme analytique qu'une SVM Gaussienne, c.a.d. une somme pondérée de Gaussiennes isotropes centrées sur un petit sous-ensemble des points d'apprentissage.
– Pour nos variantes de KNN et Parzen, nous voulions avant tout voir si on pouvait améliorer la performance des méthodes non paramétriques classiques (les "réparer"), en tenant compte localement des directions principales des données dans le voisinage d'un point. Les fonctions résultantes n'ont pas la même forme analytique qu'une SVM Gaussienne, et ne sont clairement pas clairsemées.

Mais le fait important est que tous ces algorithmes ont obtenu de très bonnes performances sur des problèmes en haute dimension, capables d'égaler ou de surpasser les SVMs. Chacun à sa manière constitue une alternative, ne faisant pas appel à l'astuce du noyau. Par ailleurs les deux derniers articles suggèrent qu'en haute dimension il est important de prendre en considération la *structure locale* apparaissant dans les données dans le voisinage d'un point.

D'un point de vue pratique, ces algorithmes ont chacun leurs avantages et inconvénients :

– *Kernel Matching Pursuit* peut offrir un avantage pratique important par rapport aux SVMs, car un nombre réduit de point de support signifie une occupation mémoire et des temps de calculs réduits d'autant. Par ailleurs le fait qu'on optimise un coût quadratique peut s'avérer utile, dans les cas où le coût ϵ-sensitif (voir [102]) des SVMs de régression n'est pas approprié. Tout dépend de l'objectif recherché : le minimiseur du coût quadratique (un coût L_2) est *l'espérance conditionnelle*, alors que le minimiseur d'un coût L_1 (ce vers quoi tend le coût ϵ-sensitif quand $\epsilon \rightarrow 0$) est la *médiane conditionnelle*. Par exemple pour la distribution particulière des données de réclamation d'assurance, le coût minimisé par les SVMs de régression n'est pas approprié [29]. Enfin, la simplicité et flexibilité du principe du dictionnaire permet facilement d'aller au delà de la forme des SVMs, laissant libre champ à l'expérimentation, pour par exemple y inclure des fonctions suggérées par des connaissances à priori sur le problème.

– *K-Local Hyperplane Nearest Neighbor* hérite de KNN ses avantages (pas d'entraînement nécessaire, rendant facile l'adaptation incrémentale), et ses inconvénients (grande occupation mémoire liée à la nécessité de conserver la totalité de l'ensemble d'entraînement, lenteur à l'utilisation car il faut parcourir la totalité de cet ensemble pour chaque exemple de test). Mais il permet parfois d'améliorer dramatiquement la performance (taux d'erreur de test) sans augmenter excessivement les ressources nécessaires.

– *Manifold Parzen Windows* est quant à lui très gourmand en ressources, puisqu'il nécessite une phase d'entraînement importante, et surtout une occupation mémoire de $d+1$ fois la taille de l'ensemble d'entraînement (où d est le nombre

de directions principales que l'on souhaite conserver). Ceci peut empêcher son utilisation avec des ensembles de taille importante en haute dimension.

14.2 A propos du caractère clairsemé

Dans Kernel Matching Pursuit notre objectif premier était d'obtenir des solutions davantage clairsemées qu'avec les SVMs. Ce n'était clairement pas l'objectif des deux autres articles, puisque nous y prenions comme point de départ des algorithmes non-paramétriques utilisant la totalité des données comme prototypes. Mais rien n'empêche d'utiliser avec ces variantes, des techniques d'élagage (voir par exemple [108]) qui ont été proposées pour réduire le nombre de prototypes nécessaires avec KNN et Parzen. Au contraire, il est même fort probable que la prise en compte de la directionnalité locale permette de réduire encore davantage le nombre de prototypes nécessaires, sans pour autant pénaliser la performance de généralisation. C'est là une direction de recherche qui vaut certainement d'être explorée.

Dans ce même ordre d'idée, on pourrait sans problème appliquer l'algorithme de *Kernel Matching Pursuit* en utilisant comme dictionnaire de fonctions, non pas une Gaussienne sphérique de variance fixe centrée sur chaque point, mais des Gaussiennes orientées selon les directions principales apparentes dans le voisinage, comme celles utilisées dans *Manifold Parzen*. Ainsi on devrait pouvoir prendre en compte la *structure locale* des données, tout en contrôlant strictement le nombre de composantes.

14.3 Un pendant probabiliste à HKNN pour l'estimation de densité

La recherche sur *Manifold Parzen* a au départ été motivée par le désir de développer, dans un cadre davantage probabiliste et plus propre, l'intuition qui avait mené au succès de l'algorithme HKNN. En effet, la distance de HKNN devait initialement être uniquement la distance d'un point à une variété linéaire, mais elle s'est vue affublée d'un terme fort utile de *weight decay* pour pénaliser les projections éloignées du centre du voisinage. Cette forme tendait de plus en plus à ressembler au résultat de l'évaluation d'une Gaussienne aplatie orientée...

Pourtant bâti sur la même intuition de départ, *Manifold Parzen* ne peut néanmoins pas être considéré comme la directe transposition probabiliste de HKNN. Un HKNN probabiliste devrait conserver le point de vue "transductif" et construire un modèle Gaussien local, basé uniquement sur le *voisinage du point de test*, alors que *Manifold Parzen* est une mixture de l Gaussiennes construites chacune à partir du voisinage d'un point d'entraînement. Un tel algorithme d'estimation de densité, contrairement à *Manifold Parzen*, ne nécessiterait ni une longue phase d'entraînement, ni le stockage prohibitif de l Gaussiennes paramétrée chacune par d vecteurs propres. Ce serait un grand avantage en pratique.

Précisons ce que pourrait être cet algorithme :
Soit X une variable aléatoire de densité de probabilité inconnue $p(x)$, dont on a tiré les points $\{x_1, \ldots, x_l\}$.
Soit x_0 un point de test pour lequel on veut estimer $p(x_0)$.
On définit un noyau de pondération $K_{x_0}(x)$ centré en x_0 (par exemple un noyau

Gaussien de largeur fixe ou multiple de la distance au k^{ieme} voisin).

On définit la densité produit entre deux densités g et h comme

$$(g \otimes h)(x_0) = \frac{1}{\int g(x)h(x)dx} g(x_0)h(x_0).$$

Soit $h_{x_0} = p \otimes K_{x_0}$ et soit

$$Z_{x_0} = \int p(x)K_{x_0}(x)dx = E[K_{x_0}(X)]_{X\sim p} \tag{14.1}$$

de sorte que $h_{x_0}(x) = \frac{1}{Z_{x_0}} p(x)K_{x_0}(x)$.

En particulier, en x_0 on peut alors écrire

$$p(x_0) = \frac{h_{x_0}(x_0)Z_{x_0}}{K_{x_0}(x_0)}$$

A partir de cette expression, en utilisant des approximations de Z_{x_0} et $h_{x_0}(x_0)$, on construit l'estimateur de densité local :

$$\hat{p}(x_0) = \frac{\hat{h}_{x_0}(x_0)\hat{Z}_{x_0}}{K_{x_0}(x_0)}$$

où $\hat{Z}_{x_0}$ est obtenu en remplaçant l'espérance dans l'équation 14.1 par la moyenne empirique :

$$\hat{Z}_{x_0} = \frac{1}{l}\sum_{i=1}^{l} K_{x_0}(x_i)$$

L'ensemble des points $\{x_1, \ldots, x_l\}$, pondérés par K_{x_0} est un échantillon pondéré correspondant à la densité $h_{x_0} = p \otimes K_{x_0}$. Nous décidons donc de calculer $\hat{h}_{x_0}(x_0)$ en ajustant[1], à cet échantillon pondéré, les paramètres d'un modèle simple, susceptible de bien approximer $p \otimes K_{x_0}$ près de x_0. Le modèle idéal si K_{x_0} est une

[1] Anglais : *fi tting*

Gaussienne sphérique et p est concentré le long d'une variété de plus faible dimension, est une Gaussienne potentiellement aplatie (du même genre que celles utilisées dans *Manifold Parzen*). Il suffit alors de l'évaluer en x_0 pour obtenir $\hat{h}_{x_0}(x_0)$.

Autrement dit :

$$\hat{h}_{x_0}(x_0) = \mathcal{N}(\hat{E}[X]_{X\sim h_{x_0}}, \hat{V}[X]_{X\sim h_{x_0}})(x_0)$$

Avec $\hat{E}[X]_{X\sim h_{x_0}} = \frac{1}{\sum_{i=1}^{l} K_{x_0}(x_i)} \sum_{i=1}^{l} K_{x_0}(x_i) x_i$
et $\hat{V}[X]_{X\sim h_{x_0}} = \frac{1}{\sum_{i=1}^{l} K_{x_0}(x_i)} \sum_{i=1}^{l} K_{x_0}(x_i) x_i x_i' - \hat{E}[X]^2$.

L'estimateur $\hat{p}$ correspond à un calcul de densité localement pondérée, présentant quelques similitudes avec les modèles de vraisemblance localement pondérée proposés dans [49, 62] pour l'estimation de densité. L'approche est néanmoins assez différente (les deux modèles de vraisemblance cités utilisent une forme assez particulière de pondération de la vraisemblance qui semble très différente.) pour mériter d'être étudiée de plus près, et correspond, davantage que *Manifold Parzen*, à l'idée que nous nous faisons d'un pendant probabiliste à l'algorithme HKNN. Qui plus est, les exigences en terme de mémoire et de temps de calcul sont comparables à celles de HKNN et bien moins contraignantes que celles de *Manifold Parzen*.

14.4 Conclusion

Au travers des trois articles présentés, nous avons montré qu'il est possible de développer des méthodes à noyau, *sans recourir à l'astuce du noyau*, qui sont

capables d'égaler ou surpasser les performances des SVMs. De ce point de vue, on peut à présent dire que les SVMs à noyau ont beaucoup en commun avec les méthodes "à prototypes", ou *memory based*, non seulement de par la forme analytique similaire de leur solution, mais également par les performances atteintes.

Pour l'essentiel, les algorithmes que nous avons développé sont de simples variantes d'algorithmes classiques, néanmoins les améliorations proposées ont permis un gain considérable de performance sur certains problèmes en haute dimension. Ainsi il n'est désormais plus possible de prétendre aussi facilement qu'auparavant que les SVMs sont bien supérieures à des algorithmes de type KNN par exemple. De ce point de vue, cette thèse est une tentative de réhabilitation des méthodes non paramétriques classiques, et une invitation lancée à la communauté de recherche en apprentissage automatique à les examiner de plus près.

Mais le point le plus important à retenir, est qu'une amélioration de performances très importante a pu être obtenue par rapport à KNN et Parzen, en les modifiant simplement pour qu'ils prennent en compte les *directions locales* apparaissant dans les données d'un voisinage, tel que suggéré par *l'hypothèse de variété*, selon laquelle les données seraient davantage concentrées le long de variétés de dimension inférieure. Approfondir les implications de cette *hypothèse de variété* semble être le meilleure espoir que nous ayons pour espérer battre le fléau de la dimensionalité. Nous avons montré deux exemples du succès de cette approche, et avons suggéré quelques pistes pour aller plus loin, mais il y a certainement bien d'autres façons d'incorporer cette notion dans les algorithmes d'apprentissage statistiques, anciens comme nouveaux.

Bibliography

[1] M. Aizerman, E. Braverman, and L. Rozonoer. Theoretical foundations of the potential function method in pattern recognition learning. *Automation and Remote Control*, 25:821–837, 1964.

[2] J. Aldrich. R.A. Fisher and the making of maximum likelihood 1912-22. Technical Report 9504, University of Southampton, Department of Economics, 1995.

[3] S. Amari and S. Wu. Improving support vector machine classifiers by modifying kernel functions. *Neural Networks*, 1999. to appear.

[4] C. G. Atkeson, A. W. Moore, and S. Schaal. Locally weighted learning for control. *Artificial Intelligence Review*, 11:75–113, 1997.

[5] P. Baldi and Y. Chauvin. Neural networks for fingerprint recognition. *Neural Computation*, 5(3):402–418, 1993.

[6] J. Baxter. The canonical distortion measure for vector quantization and function approximation. In *Proc. 14th International Conference on Machine Learning*, pages 39–47. Morgan Kaufmann, 1997.

[7] J. Baxter and P. Bartlett. The canonical distortion measure in feature space and 1-NN classification. In M. Jordan, M. Kearns, and S. Solla, editors, *Advances in Neural Information Processing Systems*, volume 10. MIT Press, 1998.

[8] R. Bellman. *Adaptive Control Processes: A Guided Tour*. Princeton University Press, New Jersey, 1961.

[9] Y. Bengio, R. Ducharme, and P. Vincent. A neural probabilistic language model. In T. K. Leen, T. G. Dietterich, and V. Tresp, editors, *Advances in Neural Information Processing Systems 13*, pages 932–938. MIT Press, 2001.

[10] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin. A neural probabilistic language model. *Journal of Machine Learning Research*, 3:1137–1155, 2003.

[11] B. Boser, I. Guyon, and V. Vapnik. A training algorithm for optimal margin classifiers. In *Fifth Annual Workshop on Computational Learning Theory*, pages 144–152, Pittsburgh, 1992.

[12] L. Bottou, C. Cortes, J. Denker, H. Drucker, I. Guyon, L. Jackel, Y. LeCun, U. Muller, E. Sackinger, P. Simard, and V. Vapnik. Comparison of classifier methods: a case study in handwritten digit recognition. In *International Conference on Pattern Recognition*, Jerusalem, Israel, 1994.

[13] L. Bottou and V. Vapnik. Local learning algorithms. *Neural Computation*, 4(6):888–900, 1992.

[14] M. Brand. Charting a manifold. In S. Becker, S. Thrun, and K. Obermayer, editors, *Advances in Neural Information Processing Systems*, volume 15. The MIT Press, 2003.

[15] J. Bromley, J. Benz, L. Bottou, I. Guyon, L. Jackel, Y. LeCun, C. Moore, E. Sackinger, and R. Shah. Signature verification using a siamese time delay neural network. In *Advances in Pattern Recognition Systems using Neural Network Technologies*, pages 669–687. World Scientific, Singapore, 1993.

[16] C. J. C. Burges and B. Schölkopf. Improving the accuracy and speed of support vector machines. In M. Mozer, M. Jordan, and T. Petsche, editors, *Advances in Neural Information Processing Systems*, volume 9, page 375. MIT Press, 1997.

[17] N. Chapados, Y. Bengio, P. Vincent, J. Ghosn, C. Dugas, I. Takeuchi, and L. Meng. Estimating car insurance premia: a case study in high-dimensional data inference. In T. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems*, volume 14, Cambridge, MA, 2002. The MIT Press.

[18] O. Chapelle, P. Haffner, and V. Vapnik. Svms for histogram-based image classification. *IEEE Transactions on Neural Networks*, 1999. accepted, special issue on Support Vectors.

[19] O. Chapelle, J. Weston, L. Bottou, and V. Vapnik. Vicinal risk minimization. In T. Leen, T. Dietterich, and V. Tresp, editors, *Advances in Neural Information Processing Systems*, volume 13, pages 416–422, 2001.

[20] S. Chen. *Basis Pursuit*. PhD thesis, Department of Statistics, Stanford University, 1995.

[21] S. Chen, F. Cowan, and P. Grant. Orthogonal least squares learning algorithm for radial basis function networks. *IEEE Transactions on Neural Networks*, 2(2):302–309, 1991.

[22] C. Cortes and V. Vapnik. Soft margin classifiers. *Machine Learning*, 20:273–297, 1995.

[23] A. Courant and D. Hilbert. *Methods of Mathematical Physics*. Wiley Interscience, New York, 1951.

[24] T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, 1967.

[25] I. J. Cox, J. Ghosn, and P. N. Yianilos. Feature-based face recognition using mixture-distance. In *Proceedings IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 209–216, 1996.

[26] J. Dahmen, D. Keysers, M. Pitz, and H. Ney. Structured covariance matrices for statistical image object recognition. In *22nd Symposium of the German Association for Pattern Recognition*, Kiel, Germany, 2000.

[27] G. Davis, S. Mallat, and Z. Zhang. Adaptive time-frequency decompositions. *Optical Engineering*, 33(7):2183–2191, July 1994.

[28] V. de Silva and J. Tenenbaum. Global versus local methods in nonlinear dimensionality reduction. In S. Becker, S. Thrun, and K. Obermayer,

editors, *Advances in Neural Information Processing Systems*, volume 15, pages 705–712, Cambridge, MA, 2003. The MIT Press.

[29] C. Dugas, Y. Bengio, N. Chapados, P. Vincent, G. Denoncourt, and C. Fournier. *Intelligent Techniques for the Insurance Industry*, chapter Statistical Learning Algorithms Applied to Automobile Insurance Ratemaking. World Scientific, 2003.

[30] R. A. Fisher. On an absolute citerion for frequency curves. *Messenger of Mathematics*, 41:155–160, 1912.

[31] R. A. Fisher. Frequency distribution of the values of the correlation coefficient in samples from an indefinitely large population. *Biometrika*, 10:507–521, 1915.

[32] R. A. Fisher. On the mathematical foundations of theoretical statistics. *Philosophical Transactions of the Royal Society of London*, A222:309–368, 1922.

[33] R. A. Fisher. Theory of statistical estimation. *Proceedings of the Cambridge Philosophical Society*, 22:700–725, 1925.

[34] S. Floyd and M. Warmuth. Sample compression, learnability, and the vapnik-chervonenkis dimension. *Machine Learning*, 21(3):269–304, 1995.

[35] Y. Freund and R. E. Schapire. Experiments with a new boosting algorithm. In *Machine Learning: Proceedings of Thirteenth International Conference*, pages 148–156, 1996.

[36] Y. Freund and R. E. Schapire. A decision theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Science*, 55(1):119–139, 1997.

[37] Y. Freund and R. E. Schapire. Large margin classification using the perceptron algorithm. In *Proc. 11th Annu. Conf. on Comput. Learning Theory*, pages 209–217. ACM Press, New York, NY, 1998.

[38] J. Friedman. Flexible metric nearest neighbor classification. Technical Report 113, Stanford University Statistics Department, 1994.

[39] J. Friedman. Greedy function approximation: a gradient boosting machine, 1999. IMS 1999 Reitz Lecture, February 24, 1999, Dept. of Statistics, Stanford University.

[40] J. Friedman, T. Hastie, and R. Tibshirani. Additive logistic regression: a statistical view of boosting. Technical report, August 1998, Department of Statistics, Stanford University, 1998.

[41] J. H. Friedman and W. Stuetzle. Projection pursuit regression. *J. American Statistical Association*, 76(376):817–823, Dec. 1981.

[42] S. Gallant. Optimal linear discriminants. In *Eighth International Conference on Pattern Recognition*, pages 849–852, Paris 1986, 1986. IEEE, New York.

[43] Z. Ghahramani and M. J. Beal. Variational inference for Bayesian mixtures of factor analysers. In *Advances in Neural Information Processing Systems 12*, Cambridge, MA, 2000. MIT Press.

[44] Z. Ghahramani and G. Hinton. The EM algorithm for mixtures of factor analyzers. Technical Report CRG-TR-96-1, Dpt. of Comp. Sci., Univ. of Toronto, 21 1996.

[45] T. Graepel, R. Herbrich, and J. Shawe-Taylor. Generalization error bounds for sparse linear classifiers. In *Thirteenth Annual Conference on Computational Learning Theory, 2000*, page in press. Morgan Kaufmann, 2000.

[46] S. Gunn and J. Kandola. Structural modelling with sparse kernels. *Machine Learning*, special issue on New Methods for Model Combination and Model Selection:to appear, 2001.

[47] T. Hastie and R. Tibshirani. Discriminant adaptive nearest neighbor classification and regression. In D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, editors, *Advances in Neural Information Processing Systems*, volume 8, pages 409–415. The MIT Press, 1996.

[48] G. Hinton, M. Revow, and P. Dayan. Recognizing handwritten digits using mixtures of linear models. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems 7*, pages 1015–1022. MIT Press, Cambridge, MA, 1995.

[49] N. L. Hjort and M. C. Jones. Locally parametric nonparametric density estimation. *Annals of Statistics*, 24(4):1619–1647, 1996.

[50] A. Inzenman. Recent developments in nonparametric density estimation. *Journal of the American Statistical Association*, 86(413):205–224, 1991.

[51] T. Jaakkola and D. Haussler. Exploiting generative models in discriminative classifiers, 1998.

[52] I. Jolliffe. *Principal Component Analysis*. Springer-Verlag, New York, 1986.

[53] D. Keysers, J. Dahmen, and H. Ney. A probabilistic view on tangent distance. In *22nd Symposium of the German Association for Pattern Recognition*, Kiel, Germany, 2000.

[54] G. Kimeldorf and G. Wahba. Some results on tchebychean spline functions. *Journal of Mathematics Analysis and Applications*, 33:82–95, 1971.

[55] G. R. Lanckriet, N. Cristianini, P. Bartlett, L. E. Ghaoui, and M. I. Jordan. Learning the kernel matrix with semidefinite programming. *Journal of Machine Learning Research*, 5:27–72, 2004.

[56] N. Lawrence, M. Seeger, and R. Herbrich. Fast sparse gaussian process methods: The informative vector machine. In S. Becker, S. Thrun, and K. Obermayer, editors, *Advances in Neural Information Processing Systems*, volume 15, pages 609–616. The MIT Press, 2003.

[57] Y. LeCun, B. Boser, J. Denker, D. Henderson, R. Howard, W. Hubbard, and L. Jackel. Handwritten digit recognition with a back-propagation network. In D. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 396–404, Denver, CO, 1990. Morgan Kaufmann, San Mateo.

[58] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998.

[59] Y. LeCun, L. Bottou, G. Orr, and K.-R. Müller. Efficient backprop. In G. Orr and K.-R. Müller, editors, *Neural Networks: Tricks of the Trade*, pages 9–50. Springer, 1998.

[60] S. Li and J. Lu. Face recognition using the nearest feature line method. *IEEE Transactions on Neural Networks*, 10(2):439–443, 1999.

[61] N. Littlestone and M. Warmuth. Relating data compression and learnability, 1986. Unpublished manuscript. University of California Santa Cruz. An extended version can be found in (Floyd and Warmuth 95).

[62] C. R. Loader. Local lieklihood density estimation. *Annals of Statistics*, 24(4):1602–1618, 1996.

[63] D. Lowe. Similarity metric learning for a variable-kernel classifier. *Neural Computation*, 7(1):72–85, 1995.

[64] S. Mallat and Z. Zhang. Matching pursuit with time-frequency dictionaries. *IEEE Trans. Signal Proc.*, 41(12):3397–3415, Dec. 1993.

[65] L. Mason, J. Baxter, P. Bartlett, and M. Frean. Boosting algorithms as gradient descent. In S. Solla, T. Leen, and K.-R. Müller, editors, *Advances in Neural Information Processing Systems*, volume 12, pages 512–518. MIT Press, 2000.

[66] J. Myles and D. Hand. The multi-class measure problem in nearest neighbour discrimination rules. *Pattern Recognition*, 23:1291–1297, 1990.

[67] E. A. Nadaraya. On nonparametric estimates of density functions and regression curves. *Theory of Applied Probability*, 10:186–190, 1965.

[68] C. Nadeau and Y. Bengio. Inference for the generalization error. In S. Solla, T. Leen, and K.-R. Müller, editors, *Advances in Neural Information Processing Systems*, volume 12, pages 307–313. MIT Press, 2000.

[69] D. Ormoneit and T. Hastie. Optimal kernel shapes for local linear regression. In S. Solla, T. Leen, and K.-R. Müller, editors, *Advances in Neural Information Processing Systems*, volume 12. MIT Press, 2000.

[70] G. Orr and K.-R. Muller, editors. *Neural networks: tricks of the trade*, volume 1524 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, NY, USA, 1998.

[71] E. Parzen. On the estimation of a probability density function and mode. *Annals of Mathematical Statistics*, 33:1064–1076, 1962.

[72] Y. Pati, R. Rezaiifar, and P. Krishnaprasad. Orthogonal matching pursuit: Recursive function approximation with applications to wavelet decomposition. In *Proceedings of the 27 th Annual Asilomar Conference on Signals, Systems, and Computers*, pages 40–44, Nov. 1993.

[73] T. Poggio and F. Girosi. A sparse representation for function approximation. *Neural Computation*, 10(6):1445–1454, 1998.

[74] M. Pontil and A. Verri. Properties of support vector machines. Technical Report AI Memo 1612, MIT, 1998.

[75] M. Powell. Radial basis functions for multivariable interpolation: A review, 1987.

[76] C. Rasmussen, R. Neal, G. Hinton, D. van Camp, Z. Ghahramani, R. Kustra, and R. Tibshirani. The DELVE manual, 1996. DELVE can be found at http://www.cs.toronto.edu/~delve.

[77] F. Rosenblatt. The perceptron — a perceiving and recognizing automaton. Technical Report 85-460-1, Cornell Aeronautical Laboratory, Ithaca, N.Y., 1957.

[78] S. Roweis and L. Saul. Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290(5500):2323–2326, Dec. 2000.

[79] R. E. Schapire, Y. Freund, P. Bartlett, and W. S. Lee. Boosting the margin: A new explanation for the effectiveness of voting methods. *The Annals of Statistics*, 26(5):1651–1686, 1998.

[80] A. S. Schölkopf, B. and K.-R. Müller. Nonlinear component analysis as a kernel eigenvalue problem. Technical Report 44, Max Planck Institute for Biological Cybernetics, Tübingen, Germany, 1996.

[81] B. Schölkopf, A. Smola, and K.-R. Müller. Nonlinear component analysis as a kernel eigenvalue problem. *Neural Computation*, 10:1299–1319, 1998.

[82] B. Schölkopf, K. Sung, C. Burges, F. Girosi, P. Niyogi, T. Poggio, and V. Vapnik. Comparing support vector machines with gaussian kernels to radial basis function classifiers. *IEEE Transactions on Signal Processing*, 45:2758–2765, 1997.

[83] M. Seeger, C. Williams, and N. Lawrence. Fast forward selection to speed up sparse gaussian process regression. In *Workshop on AI and Statistics*, volume 9, 2003.

[84] J. Shawe-Taylor, P. Bartlett, R. Williamson, and M. Anthony. Structural risk minimization over data-dependent hierarchies. *IEEE Transactions on Information Theory*, 44(5):1926–1940, 1998.

[85] R. D. Short and K. Fukunaga. The optimal distance measure for nearest neighbor classification. *IEEE Transactions on Information Theory*, 27:622–627, 1981.

[86] P. Simard, Y. LeCun, and J. Denker. Efficient pattern recognition using a new transformation distance. In S. J. Hanson, J. D. Cowan, and C. L. Giles, editors, *Advances in Neural Information Processing Systems 5*, pages 50–58, Denver, CO, 1993. Morgan Kaufmann, San Mateo.

[87] P. Y. Simard, Y. A. LeCun, J. S. Denker, and B. Victorri. Transformation invariance in pattern recognition — tangent distance and tangent propagation. *Lecture Notes in Computer Science*, 1524, 1998.

[88] Y. Singer. Leveraged vector machines. In S. Solla, T. Leen, and K.-R. Müller, editors, *Advances in Neural Information Processing Systems*, volume 12, pages 610–616. MIT Press, 2000.

[89] A. Smola and B. Schölkopf. Sparse greedy matrix approximation for machine learning. In P. Langley, editor, *International Conference on Machine Learning*, pages 911–918, San Francisco, 2000. Morgan Kaufmann.

[90] A. J. Smola and P. Bartlett. Sparse greedy gaussian process regression. In T. Leen, T. Dietterich, and V. Tresp, editors, *Advances in Neural Information Processing Systems*, volume 13, 2001. To appear.

[91] A. J. Smola, T. Friess, and B. Schölkopf. Semiparametric support vector and linear programming machines. In M. Kearns, S. Solla, and D. Cohn, editors, *Advances in Neural Information Processing Systems*, volume 11, pages 585–591. MIT Press, 1999.

[92] M. Stone. Cross-validatory choice and assesment of statistical predictions. *Journal of the Royal Statistical Society*, 36:111–147, 1974.

[93] R. Sutton and A. Barto. *An Introdiction to Reinforcement Learining*. MIT Press, 1998.

[94] Y. W. Teh and S. Roweis. Automatic alignment of local representations. In S. Becker, S. Thrun, and K. Obermayer, editors, *Advances in Neural Information Processing Systems*, volume 15. The MIT Press, 2003.

[95] J. Tenenbaum, V. de Silva, and J. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319–2323, Dec. 2000.

[96] T. Thrun and T. Mitchell. Learning one more thing. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI)*, San Mateo, CA, Aug. 1995. Morgan Kaufmann.

[97] A. Tikhonov and V. Arsenin. *Solutions of Ill-posed Problems*. W.H. Winston, Washington D.C., 1977.

[98] M. Tipping. The relevance vector machine. In S. Solla, T. Leen, and K.-R. Müller, editors, *Advances in Neural Information Processing Systems*, volume 12, pages 652–658. MIT Press, 2000.

[99] M. Tipping and C. Bishop. Mixtures of probabilistic principal component analysers. *Neural Computation*, 11(2):443–482, 1999.

[100] S. Tong and D. Koller. Restricted bayes optimal classifiers. In *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI)*, pages 658–664, Austin, Texas, 2000.

[101] K. Tsuda. Optimal hyperplane classifier based on entropy number bound. In *ICANN'99*, pages 419–424, 1999.

[102] V. Vapnik. *The Nature of Statistical Learning Theory*. Springer, New York, 1995.

[103] V. Vapnik. *Statistical Learning Theory*. Wiley, Lecture Notes in Economics and Mathematical Systems, volume 454, 1998.

[104] P. Vincent and Y. Bengio. A neural support vector network architecture with adaptive kernels. In *Proceedings of the International Joint Conference on Neural Network, IJCNN'2000*, volume 5, pages 5187–5192, 2000.

[105] P. Vincent and Y. Bengio. K-local hyperplane and convex distance nearest neighbor algorithms. In T. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems*, volume 14, Cambridge, MA, 2002. The MIT Press.

[106] J. Weston, A. Gammerman, M. Stitson, V. Vapnik, V. Vovk, and C. Watkins. Density estimation using support vector machines. In B. Schölkopf, C. J. C. Burges, and A. J. Smola, editors, *Advances in Kernel Methods — Support Vector Learning*, pages 293–306, Cambridge, MA, 1999. MIT Press.

[107] C. K. I. Williams and C. E. Rasmussen. Gaussian processes for regression. In D. Touretzky, M. Mozer, and M. Hasselmo, editors, *Advances in Neural Information Processing Systems*, volume 8. The MIT Press, 1995.

[108] D. R. Wilson and T. R. Martinez. Instance pruning techniques. In *Proc. 14th International Conference on Machine Learning*, pages 403–411. Morgan Kaufmann, 1997.

[109] P. N. Yianilos. Metric learning via normal mixtures. Technical report, NEC Research Institute, Princeton, NJ, October 1995.

[110] B. Zhang. Is the maximal margin hyperplane special in a feature space? Technical Report HPL-2001-89, Hewlett-Packards Labs, 2001.

Annexe A

Autorisations des coauteurs