Une boîte à outils rapide et simple pour les SVM*

Gaëlle Loosli[1], Stéphane Canu[1], S.V.N Vishwanathan[2],
Alexander J. Smola[2], Manojit Chattopadhyay[2]

[1] Laboratoire Perception, Systèmes, Information - FRE CNRS 2645
B.P. 08 - Place Emile Blondel
76131 - Mont Saint Aignan Cedex - France
gloosli@insa-rouen.fr et http://asi.insa-rouen.fr/∼gloosli/

[2] National ICT for Australia.
Canberra, ACT 0200 - Australia
vishy@axiom.anu.edu.au et http://mlg.anu.edu.au/∼vishy/

Résumé : Si les SVM (Support Vector Machines, ou Séparateurs à Vaste Marge) sont aujourd'hui reconnus comme l'une des meilleures méthodes d'apprentissage, ils restent considérés comme lents. Nous proposons ici une boîte à outils Matlab permettant d'utiliser simplement et rapidement les SVM grâce à une méthode de gradient projeté particulièrement bien adaptée au problème : *SimpleSVM* (Vishwanathan *et al.*, 2003). Nous avons choisi de coder cet algorithme dans l'environnement Matlab afin de profiter de sa convivialité tout en s'assurant une bonne efficacité. La comparaison de notre solution avec l'état de l'art dans le domaine *SMO (Sequential Minimal Optimization)*, montre qu'il s'agit là d'une solution dans certains cas plus rapide et d'une complexité moindre. Pour illustrer la simplicité et la rapidité de notre méthode, nous montrons enfin que sur la base de données MNIST, il a été possible d'obtenir des résultats satisfaisants en un temps relativement court (une heure et demi de calcul sur un PC sous linux pour construire 45 classifieurs binaires sur 60.000 exemples en dimension 576).

Mots-clés : Support Vector Machine, Séparateur à Vaste Marge, SVM, Apprentissage, Boîte à outils Matlab, Contraintes actives, Gradient projeté, MNIST.

1 Introduction

La phase de mise en œuvre d'un algorithme est l'ultime test permettant d'éprouver ses qualités. Elle peut révéler une certaine instabilité ou un autre défaut fatal, ayant échapé à l'analyse théorique, souvent focalisée sur les aspects positifs de la méthode.
Nous allons présenter ici les différents aspects liés à la mise en œuvre d'un algorithme permettant de calculer la solution du problème des Séparateurs à Vaste Marge (SVM

*Travail réalisé dans le cadre de la visite de G. Loosli et S. Canu au RSISE & NICTA à Canberra, Australie.

ou en anglais *Support Vector Machine*). Cet algorithme du type « contraintes actives » ou « gradient projeté », publié sous le nom de *SimpleSVM* (Vishwanathan *et al.*, 2003) (Gill *et al.*, 1991), a passé l'épreuve avec succès. Il s'est révelé être non seulement très efficace en terme de temps de calcul et de précision mais aussi fertile en extensions potentielles. Nous avons choisi comme première cible le langage Matlab car cet environnement est simple d'utilisation et particulièrement bien adapté au prototypage. L'ensemble des programmes dont nous allons parler est disponible en ligne[1]. Le portage sur d'autres plateformes comme R est en cours de réalisation.
Le succès de la méthode des SVM a entrainé le développement de nombreux algorithmes permettant leur mise en œuvre. La figure 10.1 de (Schölkopf & Smola, 2002) donne un excellent point de vue sur ces différentes méthodes et leurs conditions d'utilisation. Parmi elles, l'algorithme *SMO (Sequential Minimal Optimization)* est souvent considéré comme le plus efficace. Il a semblé intéressant de comparer l'efficacité de notre algorithme avec celle de SMO. Afin de s'affranchir des contraintes liées au codage et à la machine nous avons choisi de montrer sur un problème donné comment les temps de calculs respectifs évoluaient en fonction de la taille du problème. Dans cet exemple la complexité de notre approche est $\mathcal{O}(n^{1,2})$ contre $\mathcal{O}(n^{2,5})$ pour SMO. Sa simplicité d'utilisation a été mise à l'épreuve d'un problème de taille significative : la reconnaissance de caractères manuscrits avec la base de données MNIST (LeCun *et al.*, 1998) qui contient 60.000 exemples en dimension 576 pour discriminer 10 classes. Sur cet exemple, il a été possible, en une heure et demie de temps de calcul sur un PC classique, d'obtenir des résultats raisonnables. Sur ce problème, la rapidité de la méthode ainsi que son caractère réentrant (c'est-à-dire qu'elle permet un démarrage à chaud et plus l'initialisation est proche de la solution plus la métode converge rapidement) ont permis d'utiliser une technique de validation croisée pour identifier les hyper paramètres du modèle. Ce caractère réentrant fait de *SimpleSVM* une méthode adaptée à l'utilisation en ligne.

L'article est organisé de la manière suivante : après avoir rappelé la nature des SVM et les contraintes algorithmiques associées, nous présenterons l'algorithme *simpleSVM* en donnant une preuve de sa convergence. Nous discuterons ensuite les différents détails liés à sa programmation en Matlab. Enfin nous illustrerons à travers deux exprériences la rapidité et la simplicité de notre algorithme.
Avant de rentrer dans le détail de la mise en œuvre proprement dite et pour en saisir les tenants et les aboutissants, il convient de commencer par poser le problème d'optimisation lié aux SVM et d'en étudier certaines caractéristiques.

2 Séparateurs à Vaste Marge

2.1 Minimisation quadratique sous contraintes

Il est possible, pour expliquer les SVM, de commencer en se donnant un noyau. Un noyau $k(x, y)$ est une fonction de deux variables, symétrique et positive (pour plus de détails se reporter à (Atteia & Gaches, 1999)). À partir du noyau est construit l'ensemble

[1] `http://asi.insa-rouen.fr/~gloosli`

$\mathcal{H}_0$ de ses combinaisons linéaires finies :

$$\mathcal{H}_0 = \left\{ f : \mathbb{R}^d \to \mathbb{R} \middle| \; \exists k \in \mathbb{N}, \mathbf{a} \in \mathbb{R}^k, \{\mathbf{x}_i\}_{i=1}^k \in \mathbb{R}^d \; ; \; f(\mathbf{x}) = \sum_{i=1}^{k} a_i k(\mathbf{x}, \mathbf{x}_i) \right\}$$

Pour tout couple de fonctions $f \in \mathcal{H}_0$ et $g \in \mathcal{H}_0$ s'écrivant $f(\mathbf{x}) = \sum_{i=1}^{k} f_i k(\mathbf{x}, \mathbf{x}_i)$ et $g(\mathbf{x}) = \sum_{i=1}^{l} g_i k(\mathbf{x}, \mathbf{x}'_i)$, la forme bilinéaire $\langle f, g \rangle_{\mathcal{H}_0} = \sum_{i=1}^{k} \sum_{j=1}^{l} f_i g_j k(\mathbf{x}_i, \mathbf{x}'_j)$ définit un produit scalaire sur $\mathcal{H}_0$. On note $\|f\|^2_{\mathcal{H}_0} = \langle f, f \rangle_{\mathcal{H}_0}$ la norme associée. $\mathcal{H}_0$ muni du produit scalaire $\langle ., . \rangle_{\mathcal{H}_0}$ est un pré-hilbertien dans lequel la propriété de reproduction est vérifiée puisque $\langle f(.), k(\mathbf{x}, .) \rangle_{\mathcal{H}_0} = f(\mathbf{x})$. Il reste à le compléter convenablement pour obtenir l'espace de Hilbert à noyau reproduisant $\mathcal{H} = \overline{\mathcal{H}_0}$ muni du même produit scalaire étendu.
Dans ce cadre, le séparateur à vaste marge du problème de discrimination à deux classes associé à l'échantillon $(\mathbf{x}_i, y_i), i \in [1, m]$ avec le codage $y_i \in \{-1, 1\}$, est la solution du problème d'optimisation sous contrainte suivant pour $f \in \mathcal{H}$:

$$\left\{ \begin{array}{lll} & \min\limits_{f,b,\boldsymbol{\xi}} & \frac{1}{2}\|f\|^2_{\mathcal{H}} + C \sum\limits_{i=1}^{m} \xi_i \\ \text{avec} & & y_i(f(\mathbf{x}_i) + b) \geq 1 - \xi_i \qquad i \in [1, m] \\ \text{et} & & \xi_i \geq 0, \qquad\qquad\qquad\quad i \in [1, m] \end{array} \right. \tag{1}$$

où C est un scalaire permettant de régler la régularité de la fonction de décision, b un scalaire appelé le biais et les ξ_i des variables d'écart.
La solution de ce problème est aussi le point selle du lagrangien :

$$\mathcal{L}(f, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = \frac{1}{2}\|f\|^2_{\mathcal{H}} + C \sum_{i=1}^{m} \xi_i - \sum_{i=1}^{m} \alpha_i \big(y_i(f(\mathbf{x}_i + b)) - 1 + \xi_i \big) - \sum_{i=1}^{m} \beta_i \xi_i \tag{2}$$

avec $\alpha_i > 0$ et $\beta_i > 0$. On tire une partie des conditions de Kuhn et Tucker :

$$\left\{ \begin{array}{rcl} \partial_f \mathcal{L}(f, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta}) & = & 0 \\ \partial_b \mathcal{L}(f, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta}) & = & 0 \\ \partial_{\boldsymbol{\xi}} \mathcal{L}(f, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta}) & = & 0 \end{array} \right. \Leftrightarrow \left\{ \begin{array}{rcll} f(\mathbf{x}) - \sum\limits_{i=1}^{m} \alpha_i y_i k(\mathbf{x}, \mathbf{x}_i) & = & 0 & \\ \sum\limits_{i=1}^{m} \alpha_i y_i & = & 0 & \\ C - \alpha_i - \beta_i & = & 0 & i \in [1, m] \end{array} \right.$$

On en déduit que $f(\mathbf{x}) = \sum_{i=1}^{m} \alpha_i y_i k(\mathbf{x}, \mathbf{x}_i)$. C'est grâce à cette relation que l'on peut éliminer f du lagrangien pour aboutir à la formulation duale suivante, parfois appelée le dual de Wolfe :

$$\left\{ \begin{array}{lll} & \max\limits_{\boldsymbol{\alpha} \in \mathbf{R}^m} & -\frac{1}{2}\boldsymbol{\alpha}^\top G \boldsymbol{\alpha} + \mathbf{e}^\top \boldsymbol{\alpha} \\ \text{avec} & & \boldsymbol{\alpha}^\top \mathbf{y} = 0 \\ \text{et} & & 0 \leq \alpha_i \leq C \qquad\qquad i \in [1, m] \end{array} \right. \tag{3}$$

où G est la matrice d'influence de terme général $G_{ij} = y_i y_j k(\mathbf{x_i}, \mathbf{x_j})$ et $\mathbf{e} = [1, \ldots, 1]^\top$.
La solution du problème SVM sera alors donnée par la résolution d'un problème d'optimisation quadratique en dimension m sous contraintes de boîte.

2.2 Relations entre variables primales et duales

Dans le problème précédent, chaque inconnue α_i peut être interprétée comme l'influence de l'exemple $(\mathbf{x}_i, y_i)$ dans la solution. Du fait que seuls les points frontière sont importants pour la discrimination et qu'ils sont *a priori* peu nombreux, un grand nombre de ces coefficients α vont être nuls (Steinwart, 2003). Il sera alors pertinent de répartir les m inconnues du problème en trois groupes de points $[1, m] = I_s \cup I_c \cup I_0$, définis en fonction de la valeur du multiplicateur de Lagrange α associé :

- $[I_s]$ le groupe des points **supports** est celui des vecteurs supports candidats, c'est-à-dire pour lesquels $0 < \alpha_i < C$. Ces points sont à l'intérieur de la boîte et satisfont les contraintes. Il est aussi appelé l'ensemble de travail (*working set*), car il contient les variables sur lesquelles il faut « travailler » pour connaître leur valeur,
- $[I_c]$ le groupe des points **saturés** est celui des vecteurs pour lequel $\alpha_i = C$. Ces points sont sur le bord de la boîte. Si ces points apparaissent trop proches de la classe opposée, voire complètement mélangés, limiter leur contribution à la frontière de décision permet de régulariser la solution. Dans la solution finale ces points auront tous la valeur de leur α fixée à C : ils seront contraints et contribueront à la décision,
- $[I_0]$ le groupe des points **inactifs** est celui des points pour lesquels $\alpha_i = 0$. Ces points se situent également sur l'arête de la boîte. Dans ce cas, les vecteurs sont loin de la frontière entre les classes et ne contribuent en rien à la solution. Dans la solution finale ces points auront tous la valeur de leur α fixée à 0 : ils seront contraints.

Ces trois ensembles nous aident à reformuler le problème d'optimisation :

$$\left\{\begin{array}{lll} \max\limits_{\boldsymbol{\alpha}\in\mathbf{R}^m} & -\frac{1}{2}\boldsymbol{\alpha}^\top G\boldsymbol{\alpha} + \mathbf{e}^\top\boldsymbol{\alpha} & \\ \text{avec} & \boldsymbol{\alpha}^\top\mathbf{y} = 0 & \\ \text{et} & 0 < \alpha_i < C & i \in I_s \\ \text{et} & \alpha_i = C & i \in I_c \\ \text{et} & \alpha_i = 0 & i \in I_0 \end{array}\right. \tag{4}$$

La relation entre les paramètres de la formulation duale que nous venons d'obtenir (équation 4) et les paramètres initiaux du problème (équation 1) est importante. Il est possible de la rendre explicite en passant une fois encore par l'écriture d'un lagrangien, cette fois celui du problème dual (équation 4) vu comme un problème de minimisation :

$$\mathcal{L}(\boldsymbol{\alpha}, \lambda, \boldsymbol{\mu}, \boldsymbol{\nu}) = \tfrac{1}{2}\boldsymbol{\alpha}^\top G\boldsymbol{\alpha} - \mathbf{e}^\top\boldsymbol{\alpha} - \lambda\boldsymbol{\alpha}^\top\mathbf{y} - \boldsymbol{\nu}^\top\boldsymbol{\alpha} + \boldsymbol{\mu}^\top(\boldsymbol{\alpha} - C\mathbf{e}) \tag{5}$$

où les multiplicateurs de Lagrange $\boldsymbol{\alpha}, \boldsymbol{\mu}$ et $\boldsymbol{\nu}$ doivent être positifs. Ce lagrangien nous ramène au problème primal et peut être comparé à celui obtenu à partir du problème primal (équation 2) après avoir remplacé la variable f par $\boldsymbol{\alpha}$ mais avant simplification, soit :

$$\mathcal{L}(\boldsymbol{\alpha}, b, \boldsymbol{\xi}, \boldsymbol{\beta}) = \tfrac{1}{2}\boldsymbol{\alpha}^\top G\boldsymbol{\alpha} - \mathbf{e}^\top\boldsymbol{\alpha} - b\boldsymbol{\alpha}^\top\mathbf{y} + \boldsymbol{\xi}^\top(\boldsymbol{\alpha} - \boldsymbol{\beta} + C\mathbf{e}) \tag{6}$$

Pour faire apparaître les équivalences entre paramètres, les trois cas possibles sont considérés (I_s, I_0 et I_c).

– $[0 < \alpha < C]$: dans ce cas la condition de Kuhn et Tucker stipulant l'annulation du gradient du lagrangien par rapport à $\boldsymbol{\alpha}$ s'écrit : $G\boldsymbol{\alpha} + \lambda\mathbf{y} - \mathbf{e} = 0$. Dans ce cas aussi la contrainte initiale est saturée, c'est-à-dire $y_i(f(\mathbf{x}_i) + b) = 1$ que l'on peut réécrire en utilisant les mêmes notations que précédemment $G\boldsymbol{\alpha} + b\mathbf{y} - \mathbf{e} = 0$. Le multiplicateur de Lagrange λ associé à la contrainte d'égalité est donc égal au biais $\lambda = b$.
– $[\alpha = C]$: dans ce cas $\boldsymbol{\xi} \neq 0, \boldsymbol{\nu} = 0$ et $\boldsymbol{\mu} = -G\boldsymbol{\alpha} - b\mathbf{y} + \mathbf{e} = \boldsymbol{\xi}$. Si le multiplicateur de Lagrange est positif, la variable d'écart l'est aussi. Par conséquent $y_i(f(\mathbf{x}_i) + b) < 1$ doit être vérifié pour ce cas.
– $[\alpha = 0]$: dans ce cas $\boldsymbol{\xi} = \boldsymbol{\mu} = 0$ et $\boldsymbol{\nu} = G\boldsymbol{\alpha} + b\mathbf{y} - \mathbf{e}$. Si le multiplicateur de Lagrange est positif, alors $y_i(f(\mathbf{x}_i) + b) > 1$ doit être vérifié pour ce cas.

Ces relations sont résumées dans le tableau (1). Vérifier ces conditions d'optimalité revient à calculer $G\boldsymbol{\alpha} + b\mathbf{y} - \mathbf{e}$ puis à s'assurer que pour tous les points non supports ($\alpha = 0$) cette quantité soit positive et que pour tous les points support saturés ($\alpha = C$) cette quantité soit bien négative.

Ensemble	Contraintes initiales	Contraintes primales	Contraintes Duales
I_s	$y_i(f(\mathbf{x}_i) + b) = 1$	$0 < \alpha < C, \quad \xi = 0$	$\nu = 0, \quad \mu = 0$
I_c	$y_i(f(\mathbf{x}_i) + b) = 1 - \xi_i$	$\alpha = C, \quad \xi > 0$	$\nu = 0, \quad \mu > 0$
I_0	$y_i(f(\mathbf{x}_i) + b) > 1$	$\alpha = 0, \quad \xi = 0$	$\nu > 0, \quad \mu = 0$

TAB. 1 – Situation des contraintes pour les trois types de variables.

Pour résoudre efficacement le problème, il est astucieux de prendre en compte la situation des points au regard des contraintes vérifiées ou non.

2.3 Les grandes lignes de la méthode proposée

L'objectif de tout algorithme SVM est double : il faut d'une part arriver à répartir l'ensemble d'apprentissage dans ces trois groupes, puis une fois la répartition connue, résoudre le problème. Il s'avère que cette seconde phase est relativement plus facile.
Supposons que l'on connaisse la répartition des points (I_s, I_C et I_0 donnés) : les contraintes d'inégalités ne sont plus nécessaires (elles sont contenues implicitement dans la définition des trois ensembles de points). Seules les valeurs des α_i pour $i \in I_s$ demeurent inconnues, car par définition $\alpha_i = C$ pour $i \in I_c$ et $\alpha_i = 0$ pour $i \in I_0$. La valeur des α_i pour $i \in I_s$ est ainsi donnée par la solution du problème d'optimisation suivant :

$$\begin{cases} \max\limits_{\boldsymbol{\alpha} \in \mathbf{R}^{|I_s|}} & -\frac{1}{2}\boldsymbol{\alpha}^\top G_s \boldsymbol{\alpha} + \mathbf{e}_s^\top \boldsymbol{\alpha} \\ \text{avec} & \boldsymbol{\alpha}^\top \mathbf{y}_s + C\mathbf{e}_c^\top \mathbf{y}_c = 0 \end{cases} \tag{7}$$

avec $\mathbf{e}_s = \mathbf{e}(I_s) + 2CG(I_s, I_c)\mathbf{e}(I_c)$, $G_s = G(I_s, I_s)$, $\mathbf{y}_s = \mathbf{y}(I_s)$, $\mathbf{y}_c = \mathbf{y}(I_c)$ et $\mathbf{e}_c$ un vecteur de un. Il est ici important de noter que la dimension de ce problème est égale au cardinal de l'ensemble I_s (qui peut être bien inférieur à m, la dimension initiale du problème). Les conditions de Kuhn et Tucker nous donnent le système à résoudre pour obtenir la valeur des coefficients $\boldsymbol{\alpha}$ encore inconnus :

$$\begin{pmatrix} G_s & \mathbf{y}_s \\ \mathbf{y}_s^\top & 0 \end{pmatrix} \begin{pmatrix} \boldsymbol{\alpha} \\ \lambda \end{pmatrix} = \begin{pmatrix} \mathbf{e}_s \\ -C\mathbf{e}_c^\top \mathbf{y}_c \end{pmatrix} \tag{8}$$

Si la solution de ce système contient une composante violant les contraintes (une composante négative ou supérieure à C), elle indique que la répartition initiale des points I_s, I_c et I_0 est fausse. Cette composante violant les contraintes doit être enlevé de I_s pour passer dans I_c ou I_0.
Si maintenant toutes les composantes de la solution $\boldsymbol{\alpha}$ vérifient les contraintes, il n'est pas encore certain que nous ayons trouvé l'optimum du problème initial. Il nous faut encore vérifier que les contraintes de positivité sont respectées pour les multiplicateurs de Lagrange associés (*cf.* paragraphe précédent). Alors, nous allons vérifier que pour tous les points de I_c, $G\boldsymbol{\alpha}+b\mathbf{y}-\mathbf{e} < 0$, et que pour tous les points de I_0, $G\boldsymbol{\alpha}+b\mathbf{y}-\mathbf{e} > 0$. Si tel n'est pas le cas, le point violant les contraintes devra changer d'appartenance et quitter I_c ou I_0 pour passer dans I_s.
Nous venons là de donner le principe de notre algorithme. Il s'agit d'un algorithme itératif qui va, à chaque itération, ajouter ou enlever un seul point de l'ensemble I_s. Nous verrons qu'à chaque itération le coût décroît strictement, garantissant ainsi la convergence de la méthode. Enfin, puisqu'entre chaque itération les matrices G_s ne diffèrent que d'une ligne et d'une colonne, il est possible de calculer la nouvelle solution directement à partir de l'ancienne, réduisant ainsi la complexité de chaque itération de $\mathcal{O}(n^3)$ à $\mathcal{O}(n^2)$.

3 *SimpleSVM* - Méthode des contraintes actives

Dans cette partie nous allons introduire formellement l'algorithme *SimpleSVM* et détailler sont fonctionnement. Cet algorithme met en œuvre la méthode des contraintes actives pour la résolution d'un problème de programmation quadratique sous contraintes avec des contraintes « de boîte ». Ce type de contrainte a un intérêt : il est facile de projeter dessus.

Une autre manière d'introduire l'algorithme est de le présenter comme une méthode de gradient projeté. Le principe consiste à partir de l'intérieur du domaine des contraintes admissibles et d'y rester en recherchant à chaque itération une direction de descente admissible qui nous amène sur la frontière du domaine. Une telle direction peut être obtenue en projetant le gradient sur cette frontière ((Minoux, 1983) page 197). C'est cette projection qui va nous amener à faire passer un point de I_s vers I_0 ou I_c.

3.1 Activation d'une contrainte : projection

Il faut ici faire attention de ne pas mélanger contrainte active et point actif dans la solution. Si l'on active une contrainte, cela veut dire que l'on fixe α à l'une des bornes (0 ou C) et par conséquent que le point correspondant est inactif dans le classifieur.
Supposons que l'on connaisse un point admissible $\widetilde{\boldsymbol{\alpha}}_{\text{old}} = (\boldsymbol{\alpha}_{\text{old}}, \lambda_{\text{old}})$. L'optimum non contraint $\widetilde{\boldsymbol{\alpha}}^*$ est donné par l'équation (8) :

$$\widetilde{\boldsymbol{\alpha}}^* = H^{-1}\widetilde{\mathbf{e}} \qquad \text{avec} \quad H = \begin{pmatrix} G_s & \mathbf{y}_s \\ \mathbf{y}_s^\top & 0 \end{pmatrix} \quad \text{et} \quad \widetilde{\mathbf{e}} = \begin{pmatrix} \mathbf{e}_s \\ -C\mathbf{e}_c^\top \mathbf{y}_c \end{pmatrix} \qquad (9)$$

On choisit la direction de descente $\mathbf{d} = \widetilde{\boldsymbol{\alpha}}^* - \widetilde{\boldsymbol{\alpha}}_{\text{old}}$, celle du gradient, qui garantit que le coût va décroître. Afin de rester dans le domaine admissible $0 \leq \boldsymbol{\alpha} \leq C$, nous allons

recenser toutes les composantes de $\boldsymbol{\alpha}^*$ violant les contraintes, et rechercher le pas de descente t^* minimisant le coût tout en nous maintenant dans le domaine admissible, c'est-à-dire :

$$t^* = \max_t \left(\boldsymbol{\alpha} = \boldsymbol{\alpha}_{\text{old}} + t\mathbf{d} \mid 0 \leq \boldsymbol{\alpha} \leq C\right)$$

Pratiquement, le pas de descente est donné directement par la recherche du minimum suivant :

$$t^* = \min_t \left(-\frac{\boldsymbol{\alpha}_{\text{old}}}{\mathbf{d}}, \frac{C - \boldsymbol{\alpha}_{\text{old}}}{\mathbf{d}} \right)$$

Cela revient à projeter sur la frontière de la boîte et à éliminer de l'ensemble I_s la variable correspondant au plus grand pas possible. La nouvelle solution est alors :

$$\widetilde{\boldsymbol{\alpha}}_{\text{new}} = \widetilde{\boldsymbol{\alpha}}_{\text{old}} + t^*\mathbf{d} \tag{10}$$

Proposition 3.1 (Décroisance du coût)
Le coût diminue strictement à chaque itération (étape 3.1 de l'algorithme 1).

Démonstration. Cette propriété découle de la convexité du coût et nous donnons ici le détail. Considérons le coût :

$$q(\boldsymbol{\alpha}) = \tfrac{1}{2}\boldsymbol{\alpha}^\top H\boldsymbol{\alpha} - \boldsymbol{\alpha}^\top\widetilde{\mathbf{e}}$$

la direction de descente $\mathbf{d}$ est celle du gradient, elle vérifie :

$$\begin{aligned} \mathbf{d} &= \widetilde{\boldsymbol{\alpha}}^* - \widetilde{\boldsymbol{\alpha}}_{\text{old}} \\ &= H^{-1}\widetilde{\mathbf{e}} - \widetilde{\boldsymbol{\alpha}}_{\text{old}} \end{aligned}$$

et donc $G\mathbf{d} = \mathbf{e} - G\widetilde{\boldsymbol{\alpha}}_{\text{old}}$. On a alors :

$$\begin{aligned} q(\widetilde{\boldsymbol{\alpha}}_{\text{new}}) - q(\widetilde{\boldsymbol{\alpha}}_{\text{old}}) &= q(\widetilde{\boldsymbol{\alpha}}_{\text{old}} + t\mathbf{d}) - q(\widetilde{\boldsymbol{\alpha}}_{\text{old}}) \\ &= \tfrac{1}{2}\, t^2\mathbf{d}^\top H\mathbf{d} + t(H\widetilde{\boldsymbol{\alpha}}_{\text{old}} - \widetilde{\mathbf{e}})^\top\mathbf{d} \\ &= \tfrac{1}{2}\, t^2\mathbf{d}^\top H\mathbf{d} - t\mathbf{d}^\top H\mathbf{d} \\ &= \tfrac{1}{2}t^2 - t\ \ \mathbf{d}^\top H\mathbf{d} \qquad\qquad < 0 \end{aligned}$$

car le premier terme est négatif pour $0 < t < 2$ et le second est positif puisque la matrice G est définie positive et que le vecteur $\mathbf{d}$ n'a pas toutes ses composantes nulles sauf la dernière. □

3.2 Désactivation d'une contrainte

Désactiver une contrainte signifie permettre au α concerné de prendre une valeur libre. Cela revient à faire passer le vecteur correspondant dans le groupe I_s, des vecteurs supports candidats.
Cela s'avère indispensable lorsque le vecteur $\widetilde{\boldsymbol{\alpha}}^*$ calculé précédemment est admissible (lorsque $t = 1$). Dans ce cas, pour qu'il existe encore une direction de descente (pour que le coût continue à décroître) il faut trouver soit un point de I_0 pour lequel $y_i(f(\mathbf{x}_i)+ b) - 1 < 0$, soit un point de I_c pour lequel $y_i(f(\mathbf{x}_i) + b) - 1 > 0$. Ces contraintes correspondent au fait que le gradient est convexe au point concerné et qu'il est possible de diminuer le coût en entrant dans le domaine admissible.

De manière intuitive, on vérifie dans le primal que la solution candidate classe bien les points du jeu d'apprentissage. Si ce n'est pas le cas, alors il faut réajuster la solution en ajoutant au groupe des actifs un des points mal « classé » (au sens de I_s, I_0 et I_c).

3.3 Convergence

L'algorithme (1) résume le principe des itérations de l'algorithme du *simpleSVM*. Nous allons reprendre la preuve de convergence de l'algorithme donnée dans (Vishwanathan *et al.*, 2003).

Algorithme 1 : *simpleSVM*

1. (I_s, I_0,I_c) $\leftarrow$ initialiser
2. (α,λ) $\leftarrow$ resoudreSystemeSansContrainte(I_s) `équation (9)`

si $\min \alpha < 0$

  3.1 projeter α à l'intérieur du domaine admissible `équation (10)`

  3.2 transférer le point associé de I_s vers I_0 ou I_c

  aller à 2.

sinon

  4. calculer les multiplicateurs de Lagrange $\boldsymbol{\mu}(I_c)$ et $\boldsymbol{\nu}(I_0)$, `tableau (1)`

  si $\min \boldsymbol{\nu}(I_0) < \epsilon$ ou $\min \boldsymbol{\mu}(I_c) < \epsilon$

    5. transférer un point associé de I_0 ou I_c vers I_s

    aller à 2.

  sinon le minimum est atteint

fin si

L'algorithme s'arrête lorsque le vecteur $\widetilde{\boldsymbol{\alpha}}^*$ calculé précédemment est admissible et tous les points de I_0 et I_c vérifient leurs contraintes. Il n'existe alors plus de direction de descente admissible. Comme la solution dépend de la répartition des points dans les ensembles I_s, I_0 et I_c, qu'il n'existe qu'un nombre fini de points donc de combinaisons et que le coût associé à l'algorithme décroît strictement à chaque itération, l'algorithme ne peut pas boucler et va atteindre la solution globale en un temps fini.

4 Une boîte à outils en Matlab

Nous avons mis en œuvre l'algorithme présenté précédemment sous la forme d'un ensemble de programmes constituant une boîte à outils Matlab[2]. Nous allons maintenant décrire cet ensemble de programmes. Les deux fonctions les plus importantes sont `svm <- SVMClass(Xi,yi,svmi)` et `y <- SVMVal(x,svm)`. Elles permettent respectivement d'apprendre à partir d'un échantillon (`Xi,yi`) et de calculer `y`, l'évaluation du SVM au point `x`. Le paramètre d'entré `svmi` est facultatif mais il est très utile, car il permet d'utiliser le caractère réentrant de l'algorithme en initialisant notre algorithme avec un point proche de la solution.

4.1 Initialisation

Le principe général de l'algorithme est par défaut de commencer par très peu de points. Ainsi, les premières itérations sont très rapides. La stratégie la plus simple pour initialiser le processus est le tirage aléatoire d'un point par classe. Elle est utilisée par défaut. Il existe d'autres critères pour choisir un point par classe. On peut par exemple les choisir

[2] Disponible sur `http://asi.insa-rouen.fr/~gloosli/`

de sorte que la distance qui les sépare soit la plus courte possible. On est ainsi sûr de commencer avec des points qui seront vecteurs supports (Roobaert, 2000).
On peut aussi procéder en projetant la solution calculée sans contraintes sur un sous-ensemble des points, à l'intérieur du domaine admissible, constituant ainsi trois groupes de points (saturés à C, actifs et inactifs). On met dans le groupe des saturés tous les points tels que $\alpha \geq C$, dans le groupe des inactifs tous les points tels que $\alpha \leq 0$ et les derniers comme solution candidate dans le groupe des supports.

4.2 Résolution du système linéaire

Le cœur de l'algorithme, et aussi la partie la plus gourmande en temps de calcul est la résolution du système de la forme :

$$\begin{pmatrix} G & \mathbf{y} \\ \mathbf{y}^\top & 0 \end{pmatrix} \begin{pmatrix} \boldsymbol{\alpha} \\ \lambda \end{pmatrix} = \begin{pmatrix} \mathbf{e} \\ f \end{pmatrix}. \tag{11}$$

De façon à pouvoir garder le bénéfice de la positivité de la matrice G, le système est reformulé en découplant les variables. En définissant $M = G^{-1}\mathbf{y}$ et $N = G^{-1}\mathbf{e}$, $\boldsymbol{\alpha}$ et λ sont donnés par :

$$\begin{cases} \lambda = \dfrac{f - \mathbf{y}^\top M}{\mathbf{y}^\top N} \\ \boldsymbol{\alpha} = N - \lambda M \end{cases} \tag{12}$$

Deux stratégies différentes peuvent être utilisées pour résoudre le système linéaire de type $Gx = b$. Puisque nous savons la matrice G définie positive, la factorisation de Cholesky est la plus efficace. Mais dans le cas où la matrice G est singulière il est préférable d'utiliser la factorisation QR.

En pratique pour résoudre notre problème, voici comment utiliser la décomposition de Cholesky :

```
U = chol(G);
M = U\Ut\y;           % Ut désignant la transposée de U
N = U\Ut\e;
beta=(yt*M)/(yt*N);
alpha=N-beta*M;
```

La factorisation QR s'écrit $G = QR$ avec R une matrice triangulaire supérieure et Q une matrice orthogonale ($Q^\top Q = I$). Nous utilisons cette décomposition de manière classique. Elle s'écrit en Matlab :

```
[Q,R] = qr(G);
M=R\Qt*y;              % Qt désignant la transposée de Q
N=R\Qt*e;
```

4.3 Résolution Dynamique

Sachant que nous avons besoin de refaire ce calcul à chaque itération de l'algorithme avec seulement un point de différence par rapport à l'itération précédente, il est intéressant de regarder les méthodes incrémentales dont la complexité est $\mathcal{O}(n^2)$ plutôt que les méthodes directes en $\mathcal{O}(n^3)$. Les deux algorithmes de factorisation (Cholesky et QR) peuvent être traités de la sorte. Il est aussi possible de calculer incrémentalement G^{-1} l'inverse de la matrice G, en utilisant la formule de Sherman Morrison. Cette dernière solution s'avère pour l'instant la plus efficace.

4.3.1 Formulation de l'évolution de la matrice

Les formules précédentes s'appliquent lorsque la matrice B du nouveau système à résoudre s'écrit sous la forme $B = A + \mathbf{u}\mathbf{v}^\top$, oú A est la matrice d'un système dont on connait la solution avec $\mathbf{u}$ et $\mathbf{v}$ deux vecteurs donnés. Dans notre cas, lorsque la taille de la matrice augmente il faut écrire :

$$G^{new} = \begin{pmatrix} & & & 0 \\ & G^{old} & & \vdots \\ & & & 0 \\ 0 & \dots & 0 & 1 \end{pmatrix} + \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix} \begin{pmatrix} g_1^{new} \\ \vdots \\ g_n^{new} \\ -1 \end{pmatrix}^\top + \begin{pmatrix} g_1^{new} \\ \vdots \\ g_n^{new} \\ g_{n+1}^{new} \end{pmatrix} \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix}^\top$$

Pour diminuer la taille de la matrice on reprendra le même principe.

4.3.2 Factorisation incrémentale

Matlab propose une implémentation de la mise a jour dynamique de la factorisation de Cholesky. Elle n'est cependant pas proposée dans notre boîte à outils car elle s'est avérée peu efficace.
La décomposition QR a elle aussi sa version incrémentale. Il existe même deux méthodes pour arriver à mettre à jour les matrices Q et R. Matlab propose les fonctions *QRinsert* et *QRdelete* qui permettent la mise à jour de la décomposition quand on ajoute ou supprime des lignes ou colonnes, ainsi que la méthode *QRupdate*. La boîte à outils utilise cette dernière méthode de la manière suivante :

```
function [q,r] = sv_qrupdate(q,r,u,v,ind,dec)
if dec==0  q(ind,ind)=1;  end;
l = [zeros(ind-1,1);1;zeros(n-ind,1)];
[q,r] = qrupdate(q,r,u,l);           % mise à jour d'une colonne
if dec==0   v(ind) = 0;   end;
[q,r] = qrupdate(q,r,l,v);           % mise à jour d'une ligne
```

4.3.3 Sherman Morrison

La méthode de Sherman Morrison permet de calculer incrémentalement l'inverse d'une matrice. La boîte à outils utilise la formule de Sherman Morrison sur des matrices symétriques de la façon suivante :

```
function [Hi] = sherman_morrison(Hi,ind,u)
v = [zeros(1,ind-1) 1 zeros(1,n-ind)];
if Hi(ind,ind)==0                                                    % Hi matrice inversée
    Hi(ind,ind)=1;                                                % à l'étape précédente
    u(ind) = (u(ind)-1)/2;
else
    u(ind) = u(ind)/2;                          % u contient les valeurs à ajouter
end
B = (Hi - ((Hi*u)*(v*Hi))/(1+v*Hi*u));
Hi = (B - ((B*vt)*(ut*B))/(1+ut*B*vt));
```

La complexité de cette méthode est $3n^2$.

4.4 Conseil sur le choix de la méthode

Dans la boîte à outils que nous proposons, le choix de la méthode de résolution du système linéaire est un paramètre laissé à la préférence de l'utilisateur. De manière générale nous pouvons toutefois donner les cas d'utilisation majeurs :

- base de donnée de petite taille et noyau classique (semi-défini positif) : Cholesky est alors un bon choix pour sa stabilité numérique et sa rapidité en Matlab. Dans la version actuelle de la boîte à outils, cette version est la seule à utiliser du code compilé et donc plus rapide,
- base de donnée de grande dimension et noyau classique : Sherman Morrison est alors la méthode la plus adaptée car c'est une version incrémentale,
- noyau non semi-défini positif : QR est alors la seule méthode à utiliser quelles que soient les données, car elle ne requiert aucun à priori sur la matrice à décomposer. En revanche cette méthode est plus lente (de l'ordre de $26n^2$ contre $\frac{1}{3}n^2$ pour Cholesky en version incrémentale.)

4.5 Ajout d'un point dans la solution

Plusieurs stratégies peuvent être adoptées lors de la phase d'ajout d'un point dans le groupe des supports. La première, la plus proche de la formulation mathématique, consiste à prendre tous les points des groupes saturés et inactifs et vérifier que les contraintes sont bien satisfaites pour tous. Si l'on travaille sur une grande base de données, on se réalisera qu'aussi bien en mémoire qu'en calculs, cette opération demande beaucoup de ressources. Notre idée est de segmenter cet ensemble de points et de chercher un point à ajouter sur des sous-groupes de p points. S'il n'y a aucun point à ajouter dans le premier groupe, on regarde dans le suivant jusqu'à trouver un point. Si après le passage en revue de tous les points on n'a rien trouvé, alors nous avons la solution finale. Au contraire dès que l'on trouve un point, on le rajoute sans regarder plus loin et on revient à la première étape avec notre nouveau jeu d'actifs. Plus le pas de segmentation p est petit, plus le calcul de vérification de l'optimalité est rapide. En revanche plus le pas est petit, plus on choisit de points qui seront rejetés par la suite car on ne prend pas à chaque itération le pire de tous, mais le pire de quelques uns. On se retrouve donc à devoir choisir un compromis entre le nombre d'itérations et la taille du calcul à chaque itération. En pratique nous avons constaté qu'un pas de l'ordre d'une centaine de points est en général en bon compromis. Il est à noter ici que si l'on envisage de traiter des problèmes qui requièrent un traitement particulier à chaque point (par exemple si l'on veut intégrer des invariances, on regardera le point et toutes ses variations pour savoir si on l'ajoute) alors le pas devra être $p = 1$ et cela ralentira d'autant la résolution.

4.6 Critère d'arrêt

Nous avons vu que l'algorithme converge (3.3), mais pour des raisons de précision machine nous définissons une borne d'erreur ϵ de l'ordre de 10^{-5}. L'algorithme est alors arrêté quand chacun des ν_i et μ_i est positif ou nul à ϵ près. Nous admettons par là que l'algorithme à convergé lorsque chacun des points contribue pour moins de ϵ au saut de dualité (Schölkopf & Smola, 2002).

5 Expérimentations

Dans cette partie nous allons donner les résultats de différentes expériences visant à montrer les avantages de *SimpleSVM* en pratique. Pour des problèmes de taille réelle nous avons besoin d'outils rapides et dont la complexité permette d'utiliser des bases de données de grande dimension tout en restant dans des temps raisonnables. La première étude exposée ici est une étude pratique de complexité par rapport à SMO. La deuxième étude est la mise en œuvre de *SimpleSVM* sur la base de reconnaissance de caractère MNIST qui contient 60000 exemples de dix classes en dimension 576. La machine utilisée pour mener ces épreuves est un PC sous linux (pentium 4 3GHz avec 1Go de RAM).

5.1 Étude de complexité et comparaison avec SMO

Le but de cette expérience est d'avoir un aperçu concret du comportement de *SimpleSVM* indépendamment de la machine sur lequel il est mis en œuvre. notre démarche consiste à choisir un problème et de le résoudre pour différentes tailles. On peut ainsi suivre l'évolution du temps d'exécution. La pente de la droite ainsi obtenue en coordonnées logarithmiques, nous donne une estimation de la complexité de l'algorithme. Pour comparer les performances et la complexité de notre solution, nous avons choisi de faire la même expérience avec un solveur reconnu comme étant parmi les plus rapide, SMO. SMO est une méthode proposée par (Platt, 1999). Elle consiste à décomposer le problème en sous-problèmes de deux points et à optimiser ainsi les α deux à deux. De ce fait il n'y a plus de problème quadratique à résoudre. Le choix des couples à optimiser est fait selon différentes heuristiques.

5.1.1 Cadre expérimental

Cette étude consiste à comparer l'évolution des temps de calculs en fonction de la taille du jeu d'apprentissage. L'expérience est effectuée sur des données en damier (problème des *checkers*, voir figure 1).
Le code utilisé pour SMO est celui de la *toolbox* Matlab (Cawley, 2000). Le noyau et les paramètres sont identiques :

- Noyau Gaussien
- $\sigma = 0,25$
- $C = 500$

La méthode de résolution est QR dans sa version incrémentale ($QRupdate$).

5.1.2 Résultats et Discussion

La figure 1 montre que sur ce problème *SimpleSVM* est beaucoup plus efficace que SMO quand la taille du jeu d'apprentissage devient grand. On remarque par ailleurs que sur des petits jeux de données, SMO est plus rapide.
Sur ce problème concret nous voyons l'impact de la stratégie de *SimpleSVM* qui ne prend pas en compte les points non vecteurs supports lors de la résolution du problème. En effet en augmentant le nombre de points dans le damier, on ajoute très peu de points

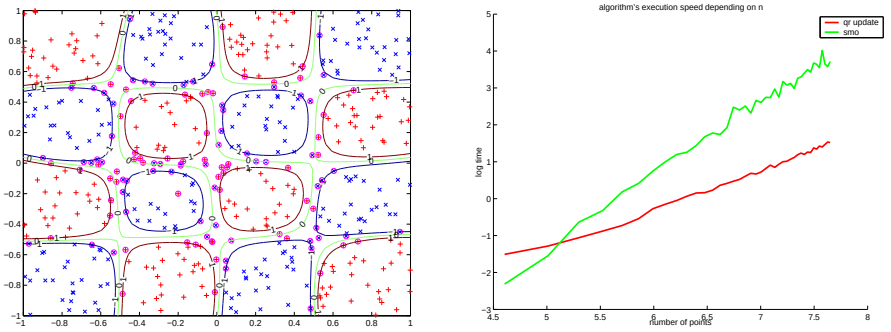

FIG. 1 – A gauche : forme des données utilisées pour comparer la complexité de SMO et *SimpleSVM* en fonction de la taille du jeu d'apprentissage. A droite : courbe de complexité de SMO et *SimpleSVM* sur le problème du damier. On représente ici le temps d'exécution (en log) en fonction du nombre de points en apprentissage (en log également). Les test sont effectués pour une taille d'apprentissage allant de 100 à 3000 points. Au delà de 200 points, SimpleSVM est plus rapide que SMO, avec une complexité de l'ordre de $n^{1,2}$ contre $n^{2,5}$ pour SMO.

sur les frontières et beaucoup à l'intérieur des cases. Alors que SMO doit comparer de plus en plus de couples, *SimpleSVM* ne se sert des points non vecteurs supports qu'au moment de l'ajout d'un nouveau point.
Il faut noter ici que si l'on se trouve face à un problème où le nombre de vecteurs supports augmente avec la taille de la base d'apprentissage, alors SimpleSVM sera pénalisé. La complexité de *SimpleSVM* dépend plus du nombre de vecteurs supports que du nombre de points d'apprentissage.

5.2 MNIST

Nous nous intéressons maintenant à l'utilisation de *SimpleSVM* dans le cadre de problèmes de taille réelle. Pour cela nous l'avons testé sur le problème de reconnaissance de caractères manuscrits MNIST. Ce jeu de données est constitué de 60000 images pour l'apprentissage et 10000 pour le test. Les images représentent les chiffres de 0 à 9 issus de codes postaux. Chaque image à une taille de 28×28 pixels dont la valeur est comprise entre 0 et 256.

5.2.1 Cadre expérimental

Tous les réglages ont été effectués sur l'ensemble d'apprentissage seul, celui-ci étant divisé en un ensemble d'apprentissage de 50000 points et un ensemble de validation de 10000 points. Les images ont été détourées de leur cadre blanc (on a donc retiré 2 pixels de chacun des 4 cotés). Les images sont ainsi réduites à 576 dimensions au lieu de 784. Par ailleurs la valeur des pixels a été translatée et réduite à l'intervalle $[-1, 1]$.

MNIST comprenant 10 classes, il a fallut choisir une méthode de classification multi-classes. Les deux principales, facilement mises en œuvre à partir de classifieurs binaires, sont le 1 contre 1 et le 1 contre tous. Dans le premier cas, nous obtenons 45 classifieurs binaires apprenant chacun sur une moyenne de 12000 points (l'équivalent de 2 classes), et la classification est donnée par un vote majoritaire sur l'ensemble des classifieurs. Dans l'autre cas nous obtenons 10 classifieurs qui apprennent chacun sur tous les points de la base d'apprentissage et la classification est donnée par le classifieur qui affiche la plus grande valeur de la fonction de décision. La complexité de SimpleSVM étant fortement liée au nombre de vecteurs supports et pour des raisons de capacité de mémoire, nous avons choisi d'implémenter le 1 contre 1.

Les résultats publiés pour MNIST avec des SVM utilisent le noyau gaussien ou le noyau polynomial. Nous avons choisi d'utiliser le noyau gaussien de la forme $k(\mathbf{x_i}, \mathbf{x_j}) = exp(-\sigma(\mathbf{x_i} - \mathbf{x_j})^2)$. Nous avons également utilisé un noyau polynomial (dont l'ordre est arbitrairement fixé à 5) de la forme $k(\mathbf{x_i}, \mathbf{x_j}) = (\sigma * \mathbf{x_i}^\top * \mathbf{x_j}) + 1)^5$. Dans tous les cas la méthode de résolution utilisée est Sherman-Morrison. Les deux paramètres essentiels à déterminer sont la largeur de bande du noyau (σ) et le paramètre de régularisation C. Le problème pour régler ces paramètres sur une base de données de cette dimension est le temps de calcul. Nous avons donc cherché à régler les paramètres sur un sous-ensemble des données et tenté d'extrapoler les valeurs optimales pour l'ensemble des données. Nous avons travaillé sur une grille de paramètres munie d'un masque (de façon à éviter les combinaisons de paramètres qui vont sur-régulariser (et mettre tous les points dans la marge) ou pour sur-apprendre (équivalent aux *hard margins* avec C inactif).

La figure 2 montre une carte obtenue pour un ensemble d'apprentissage de 5000 points et un ensemble de test de 10000 points, tirés au hasard dans le jeu d'apprentissage. Chaque point donne le taux de bonne classification du classifieur 1 contre 1 pour la combinaison de paramètres (σ, C). Les temps d'apprentissage dépendent des paramètres dans le sens où plus C est petit, plus il y a de vecteurs supports dont les multiplicateurs $\boldsymbol{\alpha}$ sont bloqués à la valeur de C et plus σ est grand, plus la largeur de bande est petite, plus on apprend par cœurs les données (ce qui donne plus de vecteur supports). Donc sur la carte 2, les temps d'apprentissage sont les plus courts pour σ petit et C grand, et les plus longs dans l'angle opposé. Pour avoir un ordre d'idée, l'apprentissage le plus long a duré 28 minutes et le plus court 5 minutes (pour les 45 classifieurs). Il est donc clair que le gain en performance ici est coûteux en temps.

5.2.2 Résultats

Les résultats présentés dans le tableau (2) sont obtenus sur le jeu de test. Celui-ci n'a été utilisé que pour l'obtention de ces résultats finaux.

Les performances en taux de classification n'atteignent pas la hauteur des meilleurs résultats publiés. Cela s'explique en partie par l'absence de pré-traitement sur les images. Car notre objectif nétait pas de donner la meilleure solution mais de montrer qu'en un temps raisonnable (une heure de demi de calcul) et avec un environnement de calcul convivial comme Matlab, il est résoudre un problème réel de grande taille. En revanche, les performances en vitesse d'exécution sont satisfaisantes compte tenu du fait que l'on traite la base complète sans stratégie de *chunking* ou autre. Il faut en moyenne 3 à 4

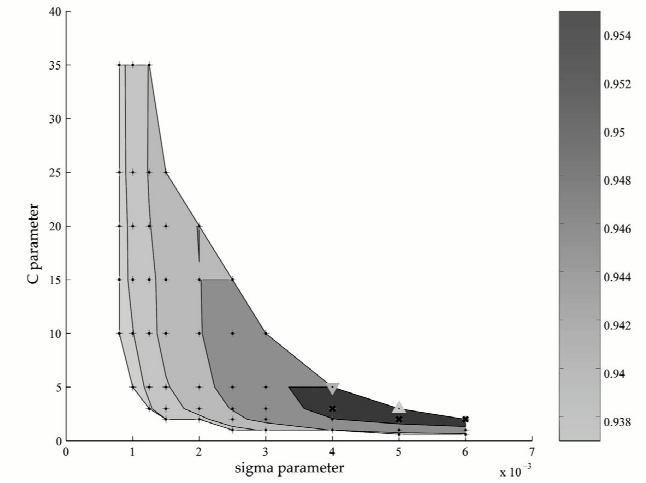

FIG. 2 – Carte obtenue pour un ensemble d'apprentissage de 5000 points et un ensemble de test de 10000 points, tirés au hasard dans le jeu d'apprentissage. Chaque point donne le taux de bonne classification du classifieur 1 contre 1 pour la combinaison de paramètres (σ, C). Le meilleur taux de classification est montré par le triangle pointant vers le haut, le meilleur compromis performance/temps de calcul par la triangle pointant vers le bas.

minutes pour entraîner un classifieur binaire avec 12000 points (ces temps dépendent des paramètres choisis dans le mesure où le temps de calcul dépend aussi du nombre de vecteurs supports).

6 Conclusion

Les SVM sont connus comme étant performants mais lents. Ici nous montrons qu'il est possible de résoudre le problème des SVM dans des temps tout à fait raisonnables. Dans cet article nous avons présenté les aspects liés à la mise en oeuvre d'un algorithme rapide. L'algorithme SimpleSVM est expliqué et l'implémentation en Matlab détaillée. Les principaux résultats de cette étude sont d'une part une boîte à outils disponible qui

Noyau	σ	C	temps d'apprentissage	erreur 1	erreur 2
Gaussien	0.002	20	1h45 (de 30s à 400s)	1.71%	0.50%
Gaussien	0.004	5	2h45 (de 60s à 720s)	1.48%	0.48%
polynomial ordre 5.	0.0005	20	1h15 (de 16s à 360s)	1.90%	0.69%

TAB. 2 – Résultat des SVM sur la base de donnée MNIST. Entre parenthèses est précisé en seconde les temps minimum et maximum pris pour estimer un SVM. L'erreur 2 correspond à la prise en compte de la deuxième classe proposée lors du vote.

fournit des méthodes rapides pour SVM et d'autre part des expérimentations montrant les capacités de cet algorithme en terme de temps de calcul et de complexité. En outre, le fait de pouvoir initialiser la méthode à partir d'une solution candidate arbitraire permet de facilement régler les hyper paramètres par des méthodes de validation croisée. Par ailleurs, de part la structure de l'algorithme, il est possible de traiter des problèmes variés qui feront l'objet de futurs travaux. Par exemple le traitement des invariances peut être intégré au moment de l'ajout d'un point dans la solution. De même il est envisageable de prendre en compte l'aspect multi-classes directement.

D'un point de vue plus général, nous nous intéressons ici à la question de la complexité dans les méthodes d'apprentissage et les SVM en particulier. Notre objectif est de profiter des particularités du problème des SVM pour diminuer la complexité de la résolution à des ordres allant de $\mathcal{O}(n)$ à $\mathcal{O}(n^2)$.

Remerciements

Nous tenons à remercier Jean-Pierre Yvon qui nous avait initialement donné l'idée d'utiliser une méthode de contraintes actives sur ce problème, Olivier Bodard qui nous a aidé à l'améliorer, ainsi que Alice Briant et Yannick Pencolé qui nous ont aidés lors de la finalisation de l'article.

National ICT Australia (NICTA) est financé par l'initiative du gouvernement australien *Backing Australia's Ability* et en partie par le Conseil de la Recherche Australienne (ARC, Australian Research Council). Ce travail a été soutenu par des financements de l'ARC.

Ce travail est financé en partie par le Programme IST de la Communauté Européenne, par le reseau d'excellence PASCAL, IST-2002-506778. Cette publication reflète seulement le point de vue des auteurs.

Références

ATTEIA M. & GACHES J. (1999). *Approxiation Hilbertienne*. Presses Universitaires de Grenoble.

CAWLEY G. C. (2000). MATLAB support vector machine toolbox (v0.55β). University of East Anglia, School of Information Systems, Norwich, Norfolk, U.K. NR4 7TJ. `http ://theoval.sys.uea.ac.uk/~gcc/svm/toolbox`.

GILL P. E., MURRAY W., SAUNDERS M. A. & WRIGHT M. H. (1991). Inertia-controlling methods for general quadratic programming. *SIAM Review*, **33**(1).

LECUN Y., BOTTOU L., BENGIO Y. & HAFFNER P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, **86**(11), 2278–2324. `http ://yann.lecun.com/exdb/mnist/`.

MINOUX M. (1983). *Programation mathématique. Théorie et algorithmes. Tome 1*. Dunod.

PLATT J. (1999). Fast training of support vector machines using sequential minimal optimization. In B. SCHOLKOPF, C. BURGES & A. SMOLA, Eds., *Advanced in Kernel Methods - Support Vector Learning*, p. 185–208 : MIT Press.

ROOBAERT D. (2000). DirectSVM : A simple support vector machine perceptron. In *Neural Network for Signal Processing X - Proceedings of the 2000 IEEE Workshop*, p. 356–365 : New York IEEE.

SCHÖLKOPF B. & SMOLA A. J. (2002). *Learning with Kernels*. MIT Press.

STEINWART I. (2003). Sparseness of support vector machine. *Journal of Machine Learning Research*, **4**, 1071–1105.

VISHWANATHAN S. V. N., SMOLA A. J. & MURTY M. N. (2003). SimpleSVM. In *Proceedings of the Twentieth International Conference on Machine Learning*.