Challenge Kaggle

Reconnaissance de chiffres manuscrits

Rapport de projet – "Méthodes à noyau"

Lilian Besson	Jaime Roquero Gimenez	Mathilde Ralle
ENS de Cachan	ENS d'Ulm	U. Paris-Sud
lilian.besson	roquero	mathilde.ralle
@ens-cachan.fr	@ens.fr	@u-psud.fr

Abstract

Ce court rapport de projet[0] présente la méthodologie choisie ainsi que les résultats obtenus pour le challenge Kaggle du cours de méthodes à noyaux du Master MVA, 2016. Notre code (Python, 2 ou 3) est fournit avec le rapport, avec des explications claires permettant la reproductibilité de nos résultats; le code est très détaillé[1].

Project Advisor: Julien Mairal (INRIA Grenoble).
Course: *"Kernel Methods"*, byJulien Mairal, in 2015–2016.
Master 2 program: Mathématiques, Vision, Apprentissage (MVA) at ENS de Cachan.
Grade: We got 14.6/20 for our project.

---

1 Introduction

Le défi consistait à classer des images représentant des chiffres (de 0 à 9), écrits à la main, en utilisant des méthodes à noyaux et le langage de notre choix. Chaque chiffre est une image, de 28×28 pixels, donnée par un vecteur de longueur 784 traduisant l'intensité de gris en chaque pixel. L'ensemble des images étaient fournis dans des fichiers au format `csv`. Nous avions en notre possession les données du Kaggle : 5000 données d'entraînement (fichier `Xtr.csv`) avec leur classe respective (fichier `Ytr.csv`) ; et 10000 données de test (fichier `Xte.csv`) à prédire. Le but était donc de produire 10000 prédictions, entre 0 et 9, de ces images de `Xte.csv`, dans un fichier de prédiction `Yte.csv` Nous avions le droit à soumettre nos prédictions au plus deux fois par jour, durant 3 semaines, ce qui nous fournissait un taux de réussite (correspondant à la vérification sur une moitié – inconnue – des données de test). La consigne du défi concernant la programmation était qu'il fallait implémenter nous-même tout ce qui a trait aux méthodes d'apprentissage statistique (SVM par exemple) ou au traitement préalable des images; ce que nous avons fait (cf. partie 3 pour quelques remarques sur l'implémentation).

[0] If needed, see on-line at `http://lbo.k.vu/kernel2016` for an e-version of this report, as well as additional resources (code, data, figures etc), open-sourced under the MIT License.

[1] Le code entier est très bien évalué, à 9.54/10, par `pylint`.

2 Notre approche

"Sanity check" Afin de s'assurer une chance de parvenir à nos fins avec des méthodes classiques, nous avons commencé par vérifier que les données d'entraînement étaient uniformément réparties selon les classes. C'est effectivement[2] le cas.

Visualisation Ensuite, nous avons affiché, grâce à la fonction implémentée dans `plot_digit.py`, quelques images pour avoir une idée de leur apparence. Contrairement aux images de la base de données MNIST, les images du défi sont légèrement bruitées (au sens où les traits dessinant le chiffre n'ont pas forcément des contours réguliers), et elles ont subi des rotations voire des symétries axiales (et elles n'ont pas toute reçues la même transformation).

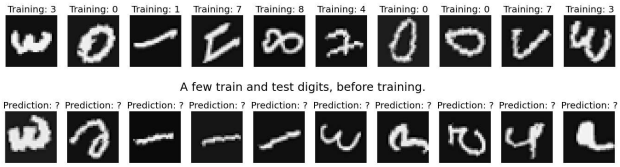

Figure 1: Exemple de 10 images d'entraînement $(3, 0, 1, 7, 8, 7, 0, 0, 7, 3)$, et de test (non étiquetées)

Chacun de nos essais a commencé sur un plus petit nombre de données (500), et de dimensions que ceux du défi (via ACP), afin de ne pas perdre inutilement du temps à tester une méthode qui, pour une raison quelconque, ne fonctionnerait pas. Notre script principal `start.py` est très modulable, pour chaque bloc, e.g., ces deux morceaux là peuvent être désactivé en changeant les valeurs booléennes `SUBSAMPLE`, `USE_PCA`.

Nettoyage des données Avant de considérer des méthodes de classification, nous avons procédé à un pré-traitement des données. La première étape consiste à centrer et réduire les données pour être en accord avec les hypothèses des modèles de classification utilisés, via la fonction `scale` (cf. `scale.py`).

Nous avons aussi essayé une réduction de dimension, en implémentant l'algorithme d'Analyse en Composante Principale (*PCA*) pour réduire la dimension des données (bien que cette méthode n'ait pas été retenue par la suite). Nos différents essais utilisait, aléatoirement, 1/3 des 5000 données d'entraînement pour l'apprentissage, et 2/3 de ces données[3] pour tester l'efficacité de nos méthodes; et cela nous a permis de constater

[2] Histogramme selon les 10 classes : 468, 553, 532, 532, 486, 432, 530, 482, 490, 495.

[3] Ce ratio 1/3 – 2/3 a été choisie pour préserver le ratio 5000 train – 10000 test.

qu'appliquer une ACP avait tendance à donner de moins bons résultats. Cela s'explique par le fait que le spectre des images est très "étalé", l'information des pixels bruts est répartie dans de nombreuses dimensions. Le fichier `pca.py` implémente la classe `myPCA`, un clone minimaliste de `sklearn.decomposition.PCA`, avec trois méthodes `fit`, `transform` et `fit_transform`.

Quel noyau utiliser ? Nous avons considéré les noyaux classiques suivants : linéaire, RBF (gaussien), de Laplace, polynomial, et sigmoïde. Le fichier correspondant est `kernels.py`, qui implémente chaque noyau. Les essais ont montré que le noyau polynomial (de degré 3 ou 4), et le RBF (avec $\gamma = 10^{-3}$) sont les plus performants.

Classification binaire et multi-classe Ces noyaux ont été utilisés dans une classification SVM implémentée dans `svm.py`. En commençant par un problème binaire (0 ou "non 0"), il fallait ensuite élargir pour considérer tous les chiffres, nous avons choisi d'utiliser l'approche "one vs. all" au lieu de "one vs. one", qui aurait été plus couteuse en temps sans forcément donner une meilleure performance. En effet, pour "one vs. all" on doit construire, entraîner et projeter à l'aide de K *classifieurs*[4] alors que "one vs. one" demande $K(K-1)/2$ classifieurs (où K désigne le nombre de classes, ici $K = 10$, soit 4.5 fois plus pour la seconde approche). Par ailleurs, un certain nombre d'études en reconnaissance de caractères (*OCR*) ont aussi montrées que l'approche "one vs. all" était préférable (et c'est désormais l'option par défaut dans `scikit-learn`). Cette transformation d'un classifieur binaire en un classifieur multi-classe est implémentée dans `svm.py` par deux classes (approche objet, imitant l'interface utilisateur des objets de `scikit-learn`, `BinarySVC` pour le classifieur binaire, et `mySVC` qui construit K instances de `BinarySVC`.

"Bagging" ou "Voting" ? Après de multiples essais (manuellement) avec chaque noyau, et en explorant les paramètres des algorithmes, nous n'étions toujours pas parvenu à un score correct, en restant en-dessous de 90%. Nous avons pensé que c'était dû à la difficulté[5] à bien séparer les chiffres 8 et 9. Pour y remédier, nous avons voulu, mettre en place un système d'agrégation ou de vote de classifieurs (Bagging ou Voting) : appliquer la méthode de classification sur plusieurs sous-ensemble des données d'entraînement, puis garder la prédiction qui apparaît le plus. Cependant cela n'a pas été fructueux (sous-ensembles insuffisants pour un bon entraînement probablement).

Problème quadratique, solveur QP ou SMO ? Aussi, un problème majeur de cette approche est la complexité en temps due au problème d'optimisation requis pour l'entraînement du classifieur SVM (méthode `fit`). Entraîner un SVM peut se faire de multiples façons, mais l'approche usuelle consiste à résoudre le problème dual, un problème quadratique (QP) contraint, généralement en haute dimension (ici, de taille $(2d)^2$, où $d = 768$, ou 1536 pour pixel+HoG). Typiquement, le SVM de `scikit-learn` mettait 8 minutes à s'entraîner sur 5000 données et prédire sur 10000; et notre première implémentation "naïve", utilisant `cvxopt.qp` pour résoudre le problème quadratique mettait environ 14 minutes. Pour tenter d'améliorer cet aspect, nous avons implémenté un algorithme SMO (cf. `SMO.py`), *théoriquement* bien plus efficace dans ce contexte. Cependant la vitesse de notre implémentation naïve de SMO, séquentielle en Python (langage interprété) et ni optimisée ni compilée, n'était pas du tout satisfaisante, et l'algorithme SMO n'est pas retenu pour la version finale. Finalement, notre implémentation d'un classifieur SVC multi-classe utilise donc un solveur de problème quadratique, fourni par le module libre `cvxopt`.

Meilleures "features" ? Sobel et HoG Nous avons cherché à utiliser des méthodes de détection de forme, notamment de contours (approche classique). Les gradients discrets en chaque pixels de l'image indiquent les changements brutaux d'intensité et permettent de détecter les contours. La première expérience a donc consisté à obtenir des "features" sur lesquelles appliquer le classifieur, à partir de filtres de Sobel directionnels, G_x et G_y. Nos tests numériques n'ont pas montré un gain en performance avec cette première approche. Nous avons essayé la méthode d'histogramme des gradients (HoG) qui permet d'extraire des features de chaque image sous forme d'histogrammes sur lesquels on applique ensuite notre classifieur SVM (RBF). La méthode HoG accepte aussi de nombreux paramètres, les principaux étant :

- le nombre d'orientations regardées (en testant $2, 4, 6, 8, 9, 12, 16$, la meilleure valeur semble 9);
- et le nombre de petits morceaux d'images à regarder, ou leur taille (`pixels_per_cell` dans `skimage.feature.hog`). En testant des patchs de tailles $(7, 7)$, $(14, 14)$ ou $(28, 28)$ (qui remplissent entièrement l'image de taille $28, 28$), ou d'autres tailles, la meilleure valeur

[4] On utilise l'anglicisme *"classifieur"* plutôt que classificateur, et quelques autres anglicismes, d'avance pardon pour les puristes.

[5] En effet, la matrice de confusion affichait un score faible pour $8, 9$; mais au dessus de 90% pour les autres chiffres.

semble $(14, 14)$, ce qui correspond à $2 \times 2 = 4$ histogrammes dans chaque image :

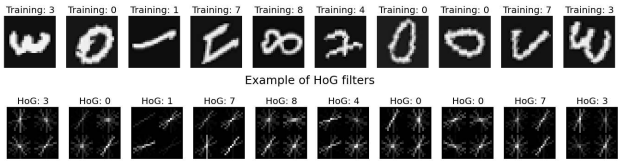

Figure 2: Exemples de "features" obtenues par HoG.

"Grid search" Cette méthode nous a permis d'atteindre le "score minimal attendu" de 93%.

Pour trouver les paramètres `C` (régularisation de la marge), les paramètres du noyau (`gamma` = γ, `degree` ou `coef0`) les plus appropriés, nous avons utilisé une recherche (non-exhaustive) dans une grille discrète de valeurs possibles des paramètres (*grid-search*, qui nous a fourni les valeurs optimales :

- Meilleur noyau : RBF (gaussien), $K(X, Y) = \exp(-\gamma \|X - Y\|^2)$;
- Meilleure constante `C` = 5.5 (sur une grille d'une trentaine de valeurs, allant de 10^{-3} à 10^4);
- Meilleur `gamma` : $\gamma = 10^{-3}$ (aussi sur une grille logarithmique d'une vingtaine de valeurs, 10^i et 5×10^i, pour $i = -6 \dots 4$).

Ces valeurs et notre approche finale nous ont permis d'améliorer le score jusqu'à 94.6% (sur notre test $1, 2/3$), et de produire les meilleurs fichiers `Yte.csv` de soumission.

2.1 Résultats

Durant nos essais, le test sur $2/3$ des données d'entraînement donnait entre 85% et 94%, selon les méthodes testées, et augmentait au fur et à mesure de l'optimisation des paramètres. Finalement, la méthode retenue est un noyau RBF pour le SVM (via un QP solveur), appliqué aux pixels brutes et aux features HoG, normalisées et concaténées, avec $\gamma = 10^{-3}$ et $C = 5.5$, Le test sur $2/3$ de `Xtr.csv` donnait 94.1%, et le score préliminaire sur la moitié de `Xte.csv` était de 94.35%, pour **un score final**[6] **de** 94.52%.

3 Détails sur l'implémentation

Les différents scripts Python sont tous valides Python 2.7+ et Python 3.4+, ils sont extrêmement détaillés (très commentés et clairs), très bien évalués par `pylint`, pour un score de $9.54/10$.

[6] Nous classant $46/71$ parmi les participants du challenge.

Pour une explication rapide du rôle de chaque fichier, voir cette page sur le dépôt git; et pour plus de détails, veuillez vous référer aux commentaires et à la documentation (consultable et générée avec `pydoc`) incluse dans chaque fichier (aussi en ligne).

4 Conclusion

Pour conclure, nous avons présenté en détail notre approche dans ce challenge de reconnaissance de chiffres manuscrits, en expliquant à la fois l'aspect temporel (tentatives successives) et l'aspect technique (quelle méthode retenue, quels grilles de paramètres explorés etc).
Quels détails concernant notre implémentation ont été donnés pour chaque morceau de notre analyse modulaire, et davantage d'information est disponible (dans le code, ou en ligne).

A Annexe

A.1 Merci

Nous tenons à remercier Julien Mairal, pour un challenge intéressant, et un cours que nous avons tous apprécié.

A.2 Références

Nous listons ici quelques références utilisées durant le challenge :

- Les slides du cours, pour les noyaux et les SVM.
- [M.Wu & Z.Zhang, 2014] et [CL.Liu et al., 2003] pour les "features" (Sobel et HoG) et des idées pour la méthodologies;
- [A.Ng, 2015] et [J.Platt, 1998] pour l'algorithme SMO;
- La documentation et exemples de `scikit-learn`;
- [S.Raschka, 2014] sur PCA, et [M.Blondel, 2010] sur les SVM;
- Et bien-sûr, Wikipedia, francophone et anglophone.

A.3 (Licence)